The SeaHorn Verification Framework*

Arie Gurfinkel[1], Temesghen Kahsai[2], Anvesh Komuravelli[3], and Jorge A. Navas[4]

[1] Software Engineering Institute / Carnegie Mellon University
[2] NASA Ames / Carnegie Mellon University
[3] Computer Science Department, Carnegie Mellon University
[4] NASA Ames / SGT

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 precision, (c) leverages the state-of-the-art in software model checking and abstract interpretation for verification, and (d) uses Horn-clauses as an intermediate language to represent verification conditions which simplifies interfacing with multiple verification tools based on Horn-clauses. SeaHorn provides users with a powerful verification tool and provides researchers with an extensible and customizable framework for experimenting with new software verification techniques. The effectiveness and scalability of SeaHorn are demonstrated using an experimental evaluation and the competition results of SV-COMP 2015 and avionics code.

1 Introduction

In this paper, we present SeaHorn, an LLVM-based [38] framework for verification of safety properties of programs. SeaHorn is a fully automated verifier that verifies user-supplied assertions as well as a number of built-in safety properties. For example, SeaHorn provides built-in checks for buffer and signed integer overflows. More generally, SeaHorn is a framework that simplifies development and integration of new verification techniques. Its main features are:

1. *It 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

* This material is based upon work funded and supported by NASA Contract No. NNX14AI09G, NSF Award No. 1422705 and by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense, NASA or NSF. This material has been approved for public release and unlimited distribution. DM-0002153

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 representation (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 [54], this provides us with a well-defined formal semantics. Finally, we use Constrained Horn Clauses (CHC) to logically represent the verification condition (VC).

2. *It provides an efficient and precise analysis of programs with procedure using new inter-procedural verification techniques.* SeaHorn summarizes the input-output behavior of procedures efficiently without inlining. The expressiveness of the summaries is not limited to linear arithmetic (as in our earlier tools) but extends to richer logics, including, for instance, arrays. Moreover, it 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.
3. *It allows developers to customize the verification semantics and offers users with verification semantics of various degrees of precision.* 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 [48], it can also automatically lift small-step semantics to large-step [7, 28] (a.k.a. Large Block Encoding, or LBE).
4. *It 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.
5. *It 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, 10, 24, 27]). 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 interpretation-based tool.
6. Finally, *it is implemented on top of the open-source LLVM compiler infrastructure.* The latter, is a well-maintained, well-documented, and continuously improving framework, it 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. Sea-

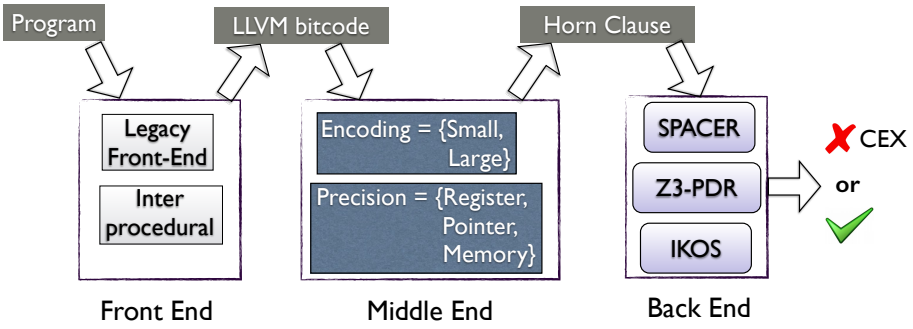


Fig. 1: Overview of SeaHorn architecture.

Horn 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 [38]. The overall approach is illustrated in Figure 1. SeaHorn has been developed in a modular fashion; its architecture is layered in three parts:

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 encoding of the VCs and the degree of precision. 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 several SMT-based model checking engines based on PDR/IC3 [13], including Spacer [35, 36] and GPDR [33]. Complementarly, SeaHorn uses the abstract interpretation-based analyzer IKOS (Inference Kernel for Open Static Analyzers) [14] for providing numerical invariants[5]. More details are reported in Section 4.

The effectiveness and scalability of SeaHorn are demonstrated by our extensive experimental evaluation in Section 5 and the results of SV-COMP 2015.

Related work. Automated analysis of software is an active area of research. There are a large number of tools with different capabilities and trade-offs [6,8,9,

[5] While conceptually, IKOS should run on CHC, currently it uses its own custom IR.

15–18, 20, 42]. Our approach on separating the program semantics from the verification engine has been previously proposed in numerous tools. In that, the tool SMACK [49] is closest to SeaHorn. Like SeaHorn, SMACK targets programs at the LLVM-IR level. However, SMACK targets Boogie intermediate verification language [22] 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 a critical for scaling verification tools and has been addressed by many researchers (e.g., [2,33,35,37,40,51]). Our approach of using mixed-semantics [30] as a source-to-source transformation has been also explored in [37]. While in [37], 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 recently proposed [11] 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., [41, 48]) and more recently adopted by an increasing number of solvers (e.g., [12,23,33,36,40]) as well as other verifiers such as UFO [4], HSF [26], VeriMAP [21], Eldarica [50], and TRACER [34].

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. SeaHorn provides two front-ends: a *legacy* front-end and an *inter-procedural* front-end.

The legacy front-end. This front-end has been used by SeaHorn for the SV-COMP 2015 competition [29] (for C programs). It was originally developed for UFO [3]. First, the input C program is pre-processed with CIL [46] to insert line markings for printing user-friendly counterexamples, define missing functions that are implicitly defined (e.g., malloc-like functions), and initialize all local variables. Moreover, it creates stubs for functions whose addresses can be taken and replaces function pointers to those functions with function pointers to the stubs. Second, the result is translated into LLVM-IR bitcode, using `llvm-gcc`. After that, it performs compiler optimizations and preprocessing to simplify the verification task. As a preprocessing step, we further initialize any uninitialized registers using non-deterministic functions. This is used to bridge the gap between the verification semantics (which assumes a non-deterministic assignment) and the compiler semantics, which tries to take advantage of the undefined behavior of uninitialized variables to perform code optimizations. We perform

a number of program transformations such as function inlining, conversion to static single assignment (SSA) form, dead code elimination, peephole optimizations, CFG simplifications, etc. We also internalize all functions to enable global optimizations such as replacement of global aggregates with scalars.

The legacy front-end is very effective for solving SV-COMP (2013, 2014, and 2015) problems. However, it has its own limitations: its design is not modular and it relies on multiple unsupported legacy tools (such as `llvm-gcc` and LLVM versions 2.6 and 2.9). Thus, it is difficult to maintain and extend.

The inter-procedural front-end. In this new front-end, SeaHorn can take any input program that can be translated into LLVM bitcode. For example, SeaHorn uses `clang` and `gcc` via `DragonEgg` [6]. Our goal is to make SeaHorn not to be limited to C programs, but applicable (with various degrees of success) to a broader set of languages based on LLVM (e.g., C++, Objective C, and Swift).

Once we have obtained LLVM bitcode, the front-end is split into two main sub-components. The first one is a pre-processor that performs optimizations and transformations similar to the ones performed by the legacy front-end. Such pre-processing is optional as its only mission is to optimize 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.

Although this front-end can optionally inline functions similarly to the legacy front-end, its major feature is a transformation that can significantly help the verification engine to produce procedure summaries.

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. To mitigate this problem, the front-end provides a transformation based on the concept of *mixed semantics*[7] [30, 37]. 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

[6] `DragonEgg` (`http://dragonegg.llvm.org/`) is a GCC plugin that replaces GCC's optimizers and code generators with those from LLVM. As result, the output can be LLVM bitcode.

[7] The term *mixed* semantics refers to a combination of small- with big-step operational semantics.

```
main ()                    main_new ()                p1_entry :              p1_new ()
    p1 (); p1 ();              if (*) goto p1_entry;      if (*) goto p2_entry;      p2_new ();
    assert (c1);               else p1_new ();            else p2_new ();            assume (c2);
p1 ()                          if (*) goto p1_entry;      if (¬c2) goto error;   p2_new ()
    p2 ();                     else p1_new ();          p2_entry :                   assume (c3);
    assert (c2);               if (¬c1) goto error;       if (¬c3) goto error;
p2 ()                          assume (false);            assume (false);
    assert (c3);                                        error : assert (false);
```

Fig. 2: A program before and after mixed-semantics transformation.

inlined only if it fails and hence, its call stack can be ignored. A key property of this transformation is that it preserves reachability and non-termination properties (see [30] for details). Since this transformation is not very common in other verifiers, we illustrate its working 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 Flexible Semantics for Developers

SeaHorn provides out-of-the-box verification semantics with different degrees of precision. Furthermore, to accommodate a variety of applications, SeaHorn is designed 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 below in Figure 3) and (b) a large-block encoding (LBE) [7]. A user can choose the encoding depending on the particular application. In practice, LBE is often more efficient but small-step might be more useful

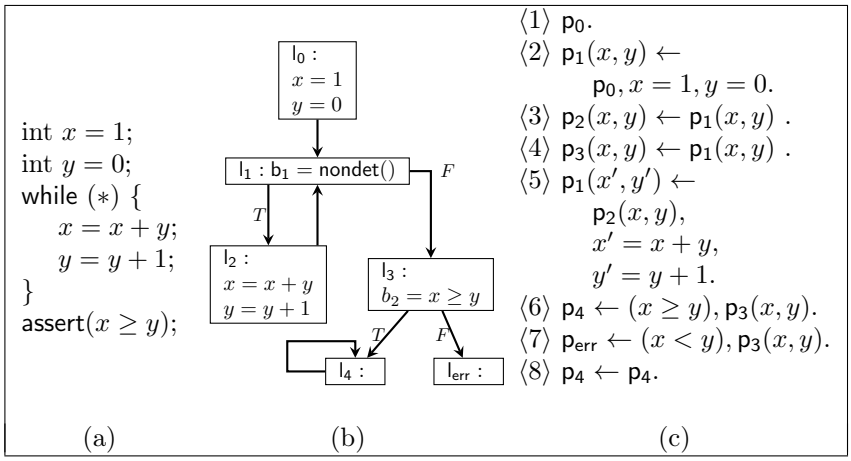


Fig. 3: (a) Program, (b) Control-Flow Graph, and (c) Verification Conditions.

if a fine-grained proof or counterexample is needed. For example, SeaHorn used the LBE encoding in SV-COMP [29].

Regardless of the encoding, SeaHorn uses CHC to encode the VCs. Given the sets $\mathcal{F}$ of function symbols, $\mathcal{P}$ of predicate symbols, and $\mathcal{V}$ of variables, a *Constrained Horn Clause (CHC)* is a formula

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

where: ϕ is a constraint over $\mathcal{F}$ and $\mathcal{V}$ with respect to some background theory; $X_i, X \subseteq \mathcal{V}$ 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 $p \in \mathcal{P}$ for first-order terms t_i constructed from $\mathcal{F}$ and X_i; and $h[X]$ is either defined analogously to p_i or is $\mathcal{P}$-free (i.e., no $\mathcal{P}$ 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 $\mathcal{P}$-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 $h[X] \leftarrow \phi, p_1[X_1], \ldots, p_k[X_k]$.

A set of CHCs is satisfiable if there exists an interpretation $\mathcal{I}$ of the predicate symbols $\mathcal{P}$ such that each constraint ϕ is true under $\mathcal{I}$. Without loss of generality, to check if a program $\mathcal{A}$ satisfies a safety property α_{safe} amounts to establishing the (un)satifiability of CHCs encoding the VCs of $\mathcal{A}$, as described next.

Example 2 (Small-step encoding of VCs using Horn clauses). Figure 3(a) shows a program which increments two variables x and y within a non-deterministic loop. After the loop is executed we would like to prove that x cannot be less than y. Ignoring wraparound issues, 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 Horn clause. A basic block label l_i in the CFG is translated into $\mathsf{p_i}(X_1, \ldots, X_n)$ such that $\mathsf{p_i} \in \mathcal{P}$ and $\{X_1, \ldots, X_n\} \subseteq \mathcal{V}$ is the set of live variables at the entry of block l_i. A Horn clause can model both the control flow and data of each block in a very 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 unsatisfiability any safe final state can be represented by an infinite loop (e.g., clause (8)).

SeaHorn middle-end offers a very simple interface for developers to implement an encoding of the verification semantics that fits their need. 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 to 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.

To model heap, SeaHorn uses heap analysis called *Data Structure Analysis (DSA)* [39]. In general, DSA is a context-sensitive, field-sensitive heap analysis that builds an explicit model of the heap. However, in SeaHorn, we use a simpler context-insensitive variant that is similar to Steensgaard's pointer analysis [52].

In DSA, 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

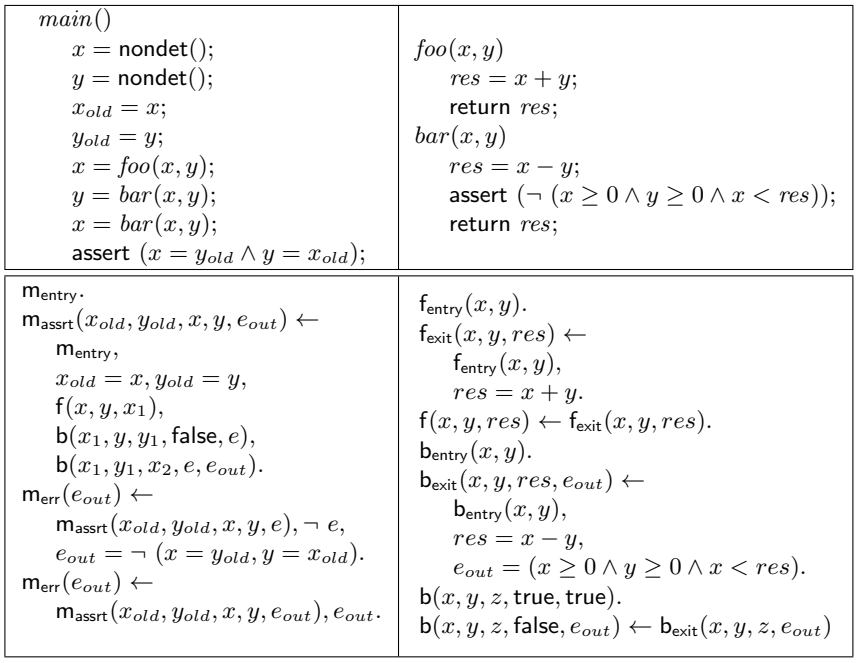


Fig. 4: A program with procedures (upper) and its verification condition (lower).

a potentially infinite set of memory objects and distinct *DSA nodes* express disjoint sets of objects. Edges in the graph represents *points-to* relationships between DS 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 for scalability and it is preserved by *merging* nodes whenever it is violated. A DS graph contains also *call nodes* representing the effect of function calls.

Given a DS graph we can map each DS node to an array. Then each memory load (read) and store (write) in the LLVM bitcode can be associated with a particular DS node (i.e., 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 [29] achieved full context-sensitivity by inlining all program functions. Although in-lining 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 tackle this problem in SeaHorn, by providing an encoding that allows inter-procedural proofs. We illustrate this procedure via the 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 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 can be, in theory, avoided if the mixed-semantics transformation described in Section 2 is applied. However, this transformation assumes that all functions can be inlined in order to raise all assertions to the main procedure. However, recursive functions and functions that contain LLVM indirect branches (i.e., branches that can jump to a label within the current function specified by an address) are not currently inlined in SeaHorn. For these reasons, our inter-procedural encoding must always consider the propagation of error across Horn clauses.

4 Verification Engines

In principle, SeaHorn can be used with any Horn clause-based verification tool. In the following, we describe two such tools developed recently 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 based on an efficient SMT-based algorithm for model checking procedural programs [35]. Compared to existing SMT-based algorithms (e.g., [2, 26, 31, 40]), the key distinguishing characteristic of Spacer is its compositionality. That is, to check safety of an input program, the algorithm iteratively creates and checks *local* reachability queries for individual procedures (or the unknown predicates of the Horn-clauses). This is crucial to avoid the exponential growth in the size of SMT formulas present in approaches based on monolithic Bounded Model Checking (BMC). To avoid redundancy and enable reuse, we maintain two kinds of summaries for each procedure: *may* and *must*. A may (must) summary of a procedure is a formula over its input-output parameters that over-approximates (under-approximates) the set of all feasible pairs of pre- and post-states.

However, the creation of new reachability queries and summaries involves existentially quantifying auxiliary variables (e.g., local variables of a procedure). To avoid dependencies on such auxiliary variables, we use a technique called *Model Based Projection* (MBP) for lazily and efficiently eliminating existential quantifiers for the theories of Linear Real Arithmetic and Linear Integer Arithmetic. At a high level, given an existentially quantified formula $\exists \overline{x} \cdot \varphi(\overline{x}, \overline{y})$, where φ is quantifier-free, it is expensive to obtain an equivalent quantifier-free formula $\psi(\overline{y})$. Instead, MBP obtains a quantifier-free under-approximation $\eta(\overline{y})$ of $\exists \overline{x} \cdot \varphi(\overline{x}, \overline{y})$. To ensure that η is a useful under-approximation, MBP uses a model-based approach such that given a model $M \models \varphi(\overline{x}, \overline{y})$, it ensures that $M \models \eta(\overline{y})$.

As mentioned in Section 3, SeaHorn models memory operations using the extensional theory of arrays (ARR). To handle the resulting Horn clauses, we have recently developed an MBP procedure for ARR. First of all, given a quantified formula $\exists a \cdot \varphi(a, \overline{y})$ where a is an array variable with index sort I and value sort V and φ is quantifier-free, one can obtain an equivalent formula $\exists \bar{i}, \overline{v} \cdot \varphi(\bar{i}, \overline{v}, \overline{y})$ where $\bar{i}$ and $\overline{v}$ are fresh variables of sort I and V, respectively. This can be achieved by a simple modification of the decision procedure for ARR by Stump et al. [53] and we skip the details in the interest of space.[8] We illustrate our MBP procedure below using an example, which is based on the above approach for eliminating existentially quantified array variables.

Let φ denote $(b = a[i_1 \leftarrow v_1]) \vee (a[i_2 \leftarrow v_2][i_3] > 5 \wedge a[i_4] > 0)$, 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 $\exists x \cdot \varphi[b[i_1 \leftarrow x]/a]$.

[8] The authors thank Nikolaj Bjørner and Kenneth L. McMillan for helpful discussions.

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.

The MBP procedure outlined above for ARR is implemented in Spacer. Additionally, the version of Spacer used in SeaHorn contains numerous enhancements compared to [35].

4.2 Abstract Interpretation with Ikos

Ikos [14] is an open-source library of abstract domains with a state-of-the-art fixed-point algorithm [5]. Available abstract domains include: intervals [19], reduced product of intervals with congruences [25], DBMs [43], and octagons [44].

SeaHorn users can choose Ikos 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, we might lose some precision during the verification. As a consequence, Ikos alone might not be sufficient as a back-end engine. Instead, a more suitable job for Ikos is to supply program invariants to the other engines (e.g. Spacer).

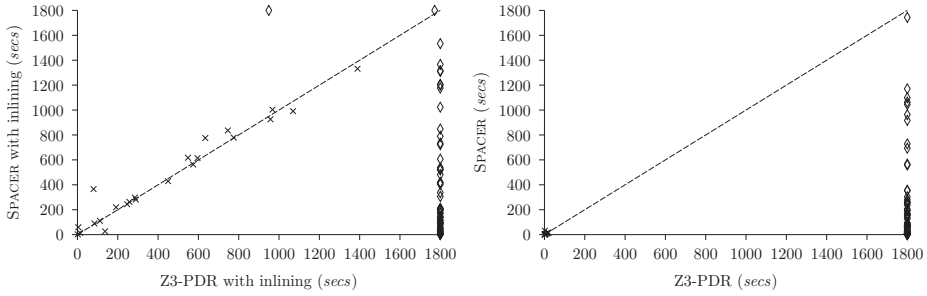
To exemplify this, let us come back to the example of Figure 3. Spacer alone can discover $x \geq y$ but it misses the vital invariant $y \geq 0$. Thus, it does not terminate. On the contrary, Ikos alone with the abstract domain of DBMs can prove safety immediately. Interestingly, Spacer populated with invariants supplied by Ikos using intervals proves safety even faster.

Although we envision Ikos to be part of the back-end it is currently part of the middle-end translating bitcode to its own custom IR. Note that there is no technical impediment to move Ikos to the back-end. Abstract interpretation tools over Horn clauses have been previously explored successfully, e.g., [32].

5 Experimental Evaluation

In this section, we describe the results of our evaluation on various C program benchmarks. First, we give an overview of SeaHorn performance at SV-COMP 2015 that used the legacy non-inter-procedural front-end. Second, we showcase the new inter-procedural verification flow on the hardest (for SeaHorn) instances from the competition. Finally, we illustrate a case study of the use of SeaHorn built-in buffer overflow checks on autopilot control software.

Results of SV-COMP 2015. For the competition, we used the legacy front-end described in Section 2. The middle-end was configured with the large step semantics and the most precise level of small-step verification semantics (i.e., registers, pointers, and heap). Note, however, that for most benchmarks the heap



(a) (b)

Fig. 5: Spacer vs. Z3-PDR on hard benchmarks (a) with and (b) without inlining

is almost completely eliminated by the front-end. Ikos with interval abstract domain and Z3-PDR were used on the back-end. Detailed results can be found at `http://tinyurl.com/svcomp15`.

Overall, SeaHorn won one gold medal in the *Simple* category – benchmarks that depend mostly on control-flow structure and integer variables – two silver medals in the categories *Device Drivers* and *Control Flow*. The former is a set of benchmarks derived from the Linux device drivers and includes a variety of C features including pointers. The latter is a set of benchmarks dependent mostly on the control-flow structure and integer variables. In the device drivers category, SeaHorn was beaten only by BLAST [8] – a tool tuned to analyzing Linux device drivers. Specifically, BLAST got 88% of the maximum score while SeaHorn got 85%. The *Control Flow* category, was won by CPAChecker [9] getting 74% of the maximum score, while SeaHorn got 69%. However, as can be seen in the quantile plot reported in the Appendix A, SeaHorn is significantly more efficient than most other tools solving most benchmarks much faster.

Results on Hard Benchmarks. SeaHorn participated in SV-COMP 2015 with the *legacy* front-end and using Z3-PDR as the verification back-end. To test the efficiency of the new verification framework in SeaHorn, we ran several experiments on the 215 benchmarks that we either could not verify or took more than a minute to verify in SV-COMP. All experiments have been carried out on an Ubuntu machine with a 2.2 GHz AMD Opteron(TM) Processor 6174 and 516GB RAM with resource limits of 30 minutes and 15GB for each verification task. In the scatter plots that follow, a diamond indicates a *time-out*, a star indicates a *mem-out*, and a box indicates an anomaly in the back-end tool.

For our first experiment, we used inlining in the front-end and Figure 5a shows a scatter plot comparing Z3-PDR and Spacer in the back-end. The plot clearly shows the advantages of the various techniques we developed in Spacer, and in particular, of Model Based Projection for efficiently and lazily eliminating existential quantifiers for integers and arrays.

Figure 5b compares the two back-end tools when SeaHorn is using inter-procedural encoding. As the plot shows, Z3-PDR runs out of time on most of the benchmarks whereas Spacer is able to verify many of them.

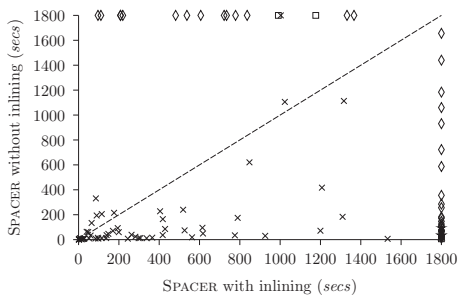


Fig. 6: Advantage of inter-procedural encoding using SPACER.

	SPACER	SPACER BMC	Z3-PDR	Total verified
SAFE	21	–	3	21
UNSAFE	74	76	7	81

Table 1: Number of hard benchmarks that are verified as safe/unsafe by SPACER in its normal and BMC mode, and Z3-PDR, with inlining disabled.

As mentioned in Section 2, inter-procedural encoding is advantageous from a usability point of view. It turns out that it also makes verification easier over-all. To see the advantage of inter-procedural encoding, we used the same tool SPACER in the back-end and compared the running times with and without inlining in the front-end. Figure 6 shows a scatter plot of the running times and we see that SPACER takes less time on many benchmarks when inlining is disabled.

SPACER also has a compositional BMC mode (see Section 4.1 for details), where no additional computation is performed towards invariant generation after checking safety for a given value of the bound. This helps SPACER show the failure of safety in two additional hard benchmarks, as shown in Table 1. The figure also shows the number of benchmarks verified by Z3-PDR, the back-end tool used in SV-COMP, for comparison.

Case Study: Checking Buffer Overflow in Avionics Software. We have evaluated the SEAHORN built-in buffer overflow checks on two autopilot control software. To prove absence of buffer overflows, we only need to add in the front-end a new LLVM transformation pass that inserts the corresponding checks in the bitcode. The middle-end and back-end are unchanged. If SEAHORN proves the program is safe then it guarantees that the program is free of buffer overflows. Details of the instrumentation are given in Appendix B.

Table 2 shows the results of our evaluation comparing SEAHORN with an abstract interpretation-based static analyzer using IKOS (labelled ANALYZER) developed at NASA Ames [14]. We have used two open-source autopilot control software mnav [45] (160K LOC) and paparazzi [47] (20K LOC). Both are versatile autopilot control software for a fixed-wing aircrafts and multi-copters. For each benchmark, we created two versions: one inlining all functions (inlined) and the other applying the mixed-semantics transformation (mixed). SEAHORN front-end instruments the programs with the buffer overflow and underflow checks. In the middle-end, we use large-step encoding and the inter-procedural encoding

Program	#C	ANALYZER		SEAHORN					
		%W	T	T_F	T_M	T_{SPACER}	T_{FMS}	$T_{\text{SPACER}} + T_{\text{IKOS}}$	T_{FMSI}
mnav.inlined	607	4.7%	36	2	18	744	764	$116 + 52$	187
mnav.mixed	815	8.2%	10	1	8	278	287	$139 + 5$	153
paparazzi.inlined	343	0%	85	2	1	–	3	–	3
paparazzi.mixed	684	43%	15	1	2	3	6	$2 + 1$	6

Table 2: A comparison between SEAHORN and ANALYZER on autopilot software.

(for mixed). For mnav, we had to model the heap, while for paparazzi, modeling registers and pointers only was sufficient. For ANALYZER, we neither inline nor add the checks explicitly as these are handled internally. Both SEAHORN and ANALYZER used intervals as the abstract domain.

In Table 2, $\#C$ denotes the number of overflow and underflow checks. For ANALYZER, we show the warning rate $\%W$ and the total time of the analysis T. For SEAHORN, we show the time spent by the front-end (T_F) and the middle-end (T_M). All times are in seconds. For the back-end, we record the time spent when SPACER alone is used (T_{SPACER}), and the time spent when both SPACER and IKOS are used ($T_{\text{SPACER}} + T_{\text{IKOS}}$). The column T_{FMS} and T_{FMSI} denote the total time, from front-end to the back-end, when SPACER alone and SPACER together with IKOS are used, respectively. SEAHORN proves absence of buffer overflows for both benchmarks, while ANALYZER can only do it for paparazzi; although, for mnav the number of warnings was low (4%). To the best of our knowledge, this is the first time that absence of buffer overflows has been proven for mnav. For the inlined paparazzi benchmark, SEAHORN was able to discharge the proof obligations using front-end only (probably because all global array accesses were lowered to scalars and all loops are bound). The performance of SEAHORN on mnav reveals that the inter-procedural encoding significantly better than the inlined version. Furthermore, IKOS has a significant impact on the results. Specially, SEAHORN with IKOS dramatically helps when the benchmark is inlined. The best configuration is the inter-procedural encoding with IKOS.

6 Conclusion

We have presented SEAHORN, a new software verification framework with a modular design that separates the concerns of the syntax of the language, its operational semantics, and the verification semantics. Building a verifier from scratch is a very tedious and time-consuming task. We believe that SEAHORN is a versatile and highly customizable framework that can help significantly the process of building new tools by allowing researchers experimenting only on their particular techniques of interest. To demonstrate the practicality of this framework, we shown that SEAHORN is a very competitive verifier for proving safety properties both for academic benchmarks (SV-COMP) and large industrial software (autopilot code).

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig Interpretation. In: SAS. pp. 300–316 (2012)
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Whale: An interpolation-based algorithm for inter-procedural verification. In: VMCAI. pp. 39–55 (2012)
3. Albarghouthi, A., Gurfinkel, A., Li, Y., Chaki, S., Chechik, M.: UFO: Verification with interpolants and abstract interpretation - (competition contribution). In: TACAS. pp. 637–640 (2013)
4. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: Ufo: A framework for abstraction- and interpolation-based software verification. In: CAV. pp. 672–678 (2012)
5. Amato, G., Scozzari, F.: Localizing widening and narrowing. In: SAS. pp. 25–42 (2013)
6. Arlt, S., Rubio-González, C., Rümmer, P., Schäf, M., Shankar, N.: The gradual verifier. In: NFM. pp. 313–327 (2014)
7. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software model checking via large-block encoding. In: FMCAD. pp. 25–32 (2009)
8. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker blast. STTT 9(5-6), 505–525 (2007)
9. Beyer, D., Keremoglu, M.E.: Cpachecker: A tool for configurable software verification. In: CAV. pp. 184–190 (2011)
10. Bjørner, N., Gurfinkel, A.: Property directed polyhedral abstraction. In: Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015. pp. 263–281 (2015)
11. Bjørner, N., McMillan, K.L., Rybalchenko, A.: Program verification as satisfiability modulo theories. In: SMT. pp. 3–11 (2012)
12. Bjørner, N., McMillan, K.L., Rybalchenko, A.: On solving universally quantified horn clauses. In: SAS. pp. 105–125 (2013)
13. Bradley, A.R.: IC3 and beyond: Incremental, inductive verification. In: Computer Aided Verification - 24th International Conference, CAV. p. 4 (2012)
14. Brat, G., Navas, J.A., Shi, N., Venet, A.: IKOS: A framework for static analysis based on abstract interpretation. In: SEFM. pp. 271–277 (2014)
15. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamaric, Z.: A reachability predicate for analyzing low-level software. In: TACAS. pp. 19–33 (2007)
16. Clarke, E., Kroening, D., Lerda, F.: A Tool for Checking ANSI-C Programs. In: TACAS. pp. 168–176 (2004)
17. Cohen, E., Dahlweid, M., Hillebrand, M.A., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: Vcc: A practical system for verifying concurrent c. In: TPHOL. pp. 23–42 (2009)
18. Cordeiro, L., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ANSI-C software. IEEE Trans. Software Eng. 38(4), 957–974 (2012)
19. Cousot, P., Cousot, R.: Static Determination of Dynamic Properties of Programs. In: Proceedings of the second international symposium on Programming, Paris, France. pp. 106–130 (1976)
20. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-c: A software analysis perspective. In: SEFM. pp. 233–247 (2012)
21. De Angelis, E., Fioravanti, F., Pettorossi, A., Proietti, M.: VeriMAP: A tool for verifying programs through transformations. In: TACAS. pp. 568–574 (2014)

22. DeLine, R., Leino, K.R.M.: BoogiePL: A typed procedural language for checking object-oriented programs. Tech. Rep. MSR-TR-2005-70, Microsoft Research (2005)
23. Gange, G., Navas, J.A., Schachte, P., Søndergaard, H., Stuckey, P.J.: Failure tabled constraint logic programming by interpolation. TPLP 13(4-5), 593–607 (2013)
24. Garoche, P., Kahsai, T., Tinelli, C.: Incremental invariant generation using logic-based automatic abstract transformers. In: NASA Formal Methods, 5th International Symposium, NFM 2013. pp. 139–154 (2013)
25. Granger, P.: Static analysis of arithmetical congruences. International Journal of Computer Mathematics (1989)
26. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI. pp. 405–416 (2012)
27. Gurfinkel, A., Chaki, S.: Combining predicate and numeric abstraction for software model checking. STTT 12(6), 409–427 (2010)
28. Gurfinkel, A., Chaki, S., Sapra, S.: Efficient Predicate Abstraction of Program Summaries. In: NFM. pp. 131–145 (2011)
29. Gurfinkel, A., Kahsai, T., Navas, J.A.: SeaHorn: A framework for verifying C programs - (competition contribution). In: To appear in TACAS (2015)
30. Gurfinkel, A., Wei, O., Chechik, M.: Model checking recursive programs with exact predicate abstraction. In: ATVA. pp. 95–110 (2008)
31. Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., Podelski, A.: Ultimate Automizer with SMTInterpol - (Competition Contribution). In: TACAS. pp. 641–643 (2013)
32. Hermenegildo, M.V., Puebla, G., Bueno, F., López-García, P.: Program development using abstract interpretation (and the ciao system preprocessor). In: SAS. pp. 127–152 (2003)
33. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: SAT. pp. 157–171 (2012)
34. Jaffar, J., Murali, V., Navas, J.A., Santosa, A.E.: TRACER: A symbolic execution tool for verification. In: CAV. pp. 758–766 (2012)
35. Komuravelli, A., Gurfinkel, A., Chaki, S.: SMT-based model checking for recursive programs. In: CAV. pp. 17–34 (2014)
36. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in smt-based unbounded software model checking. In: CAV. pp. 846–862 (2013)
37. Lal, A., Qadeer, S.: A program transformation for faster goal-directed search. In: FMCAD. pp. 147–154 (2014)
38. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO. pp. 75–88 (2004)
39. Lattner, C., Adve, V.S.: Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. In: PLDI. pp. 129–142 (2005)
40. McMillan, K., Rybalchenko, A.: Solving Constrained Horn Clauses using Interpolation. Tech. rep., MSR-TR-2013-6 (2013)
41. Méndez-Lojo, M., Navas, J.A., Hermenegildo, M.V.: A flexible, (c)lp-based approach to the analysis of object-oriented programs. In: LOPSTR. pp. 154–168 (2007)
42. Merz, F., Falke, S., Sinz, C.: LLBMC: bounded model checking of C and C++ programs using a compiler IR. In: VSTTE. pp. 146–161 (2012)
43. Miné, A.: A few graph-based relational numerical abstract domains. In: SAS. pp. 117–132 (2002)
44. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)

45. Micro NAV Autopilot Software. Available at: `http://sourceforge.net/projects/micronav/`
46. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: CC. pp. 213–228 (2002)
47. Paparazzi Autopilot Software. Available at: `http://wiki.paparazziuav.org/wiki/Main_Page`
48. Peralta, J.C., Gallagher, J.P., Saglam, H.: Analysis of imperative programs through analysis of constraint logic programs. In: SAS. pp. 246–261 (1998)
49. Rakamaric, Z., Emmi, M.: SMACK: decoupling source language details from verifier implementations. In: CAV. pp. 106–113 (2014)
50. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for horn-clause verification. In: CAV. pp. 347–363 (2013)
51. Sinha, N., Singhania, N., Chandra, S., Sridharan, M.: Alternate and learn: Finding witnesses without looking all over. In: Computer Aided Verification - 24th International Conference, CAV 2012. pp. 599–615 (2012)
52. Steensgaard, B.: Points-to analysis in almost linear time. In: POPL. pp. 32–41 (1996)
53. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.R.: A Decision Procedure for an Extensional Theory of Arrays. In: LICS. pp. 29–37 (2001)
54. Zhao, J., Nagarakatte, S., Martin, M.M.K., Zdancewic, S.: Formalizing the LLVM intermediate representation for verified program transformations. In: POPL. pp. 427–440 (2012)

A SV-COMP 2015 Quantile plot

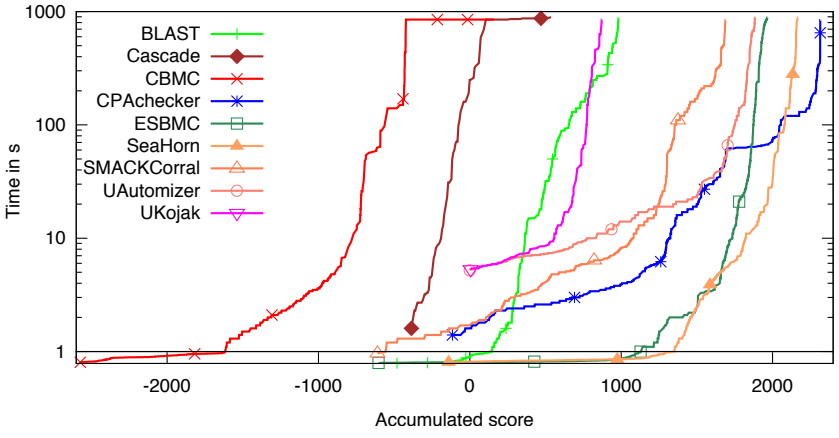


Fig. 7: Quantile graph of the results for the *Control Flow* category.

B Buffer Overflow Instrumentation

The instrumentation for inserting buffer overflow and underflow checks is as follows. For each pointer dereference `*p` we add two shadow registers: `p.offset` and `p.size`. The former is the offset from the base address of the object that contains `p` while the latter is the actual size of the allocated memory for `p` (including padding and alignment). Note that for stack and static allocations `p.size` is statically know. However, for `malloc`-like allocations `p.size` may only be known dynamically. For each pointer dereference `*p`, we add two assertions:

`assert` ($\mathsf{p.offset} \geq 0$) (Underflow)
`assert` ($\mathsf{p.offset} < \mathsf{p.size}$) (Overflow)

Then, we need also to add instructions to propagate the values of the shadow variables along the program including across procedure boundaries. More specifically, for every instruction that performs pointer arithmetic we add arithmetic operations over the shadow offset to compute its value.

For instrumenting a function `f` we add for each dereferenceable formal parameter `x` two more shadow formal parameters `x.offset` and `x.size`. Then, at a call site of `f` and for a dereferenceable actual parameter `y` we add its corresponding `y.offset` and `y.size`. Moreover, for each function that returns a pointer we add two more shadow formal parameters to represent the size and offset of the returned value. The difference here is that these two shadow variables must be passed by reference at the call site so the continuation can use those. Thus, rather than using registers we allocate them in the stack and pass their addresses to the callee.