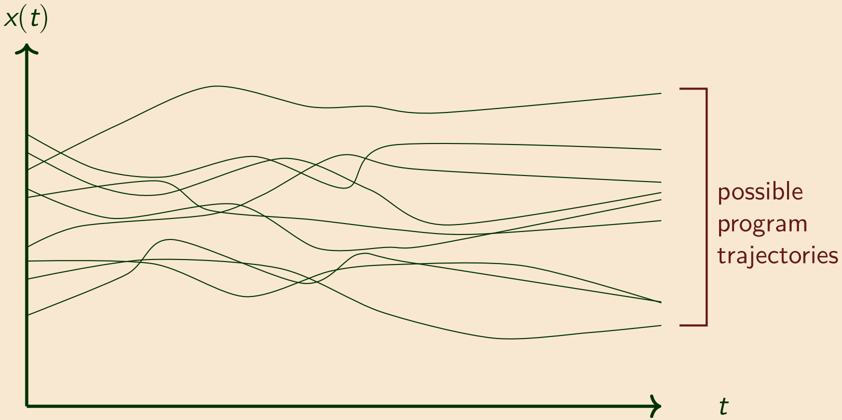
Abstract interpretation

Jeremy Yallop
jeremy.yallop@cl.cam.ac.uk

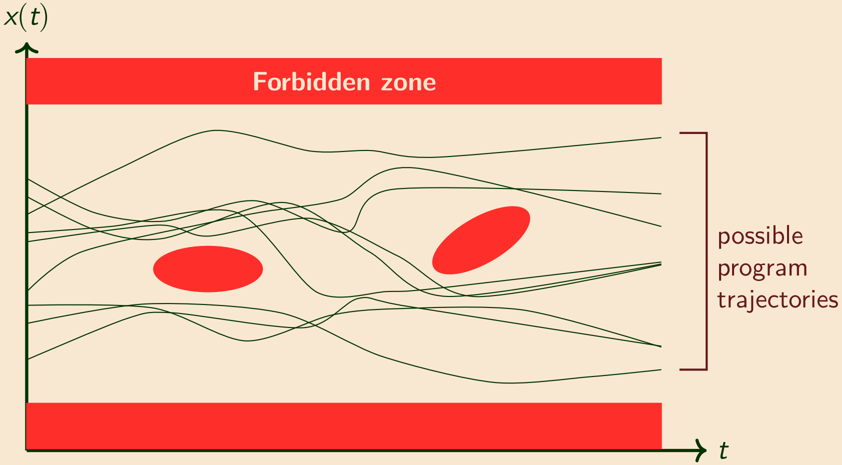
Overview[1]

[1] Based on Patrick Cousot's *Abstract Interpretation in a Nutshell*

Possible program executions

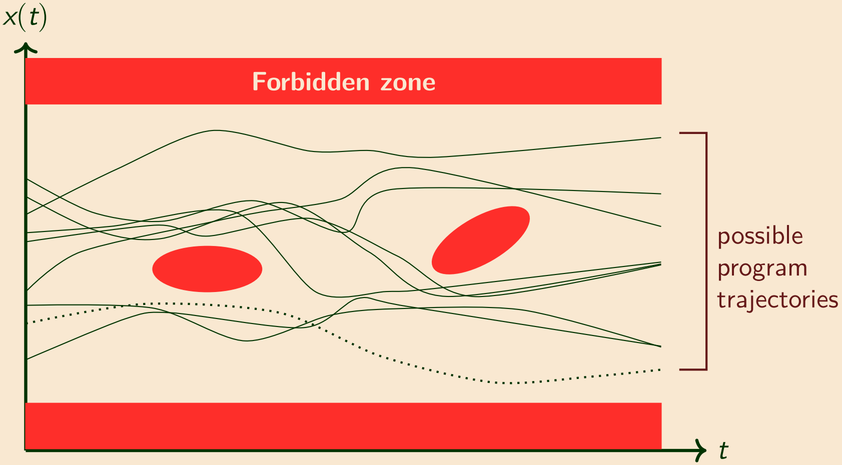


Testing cannot (generally) ensure complete absence of errors.

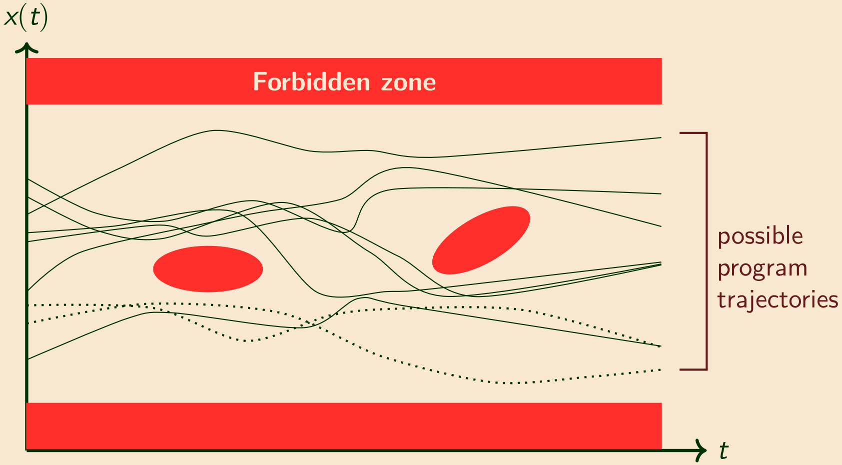


Erroneous executions & testing

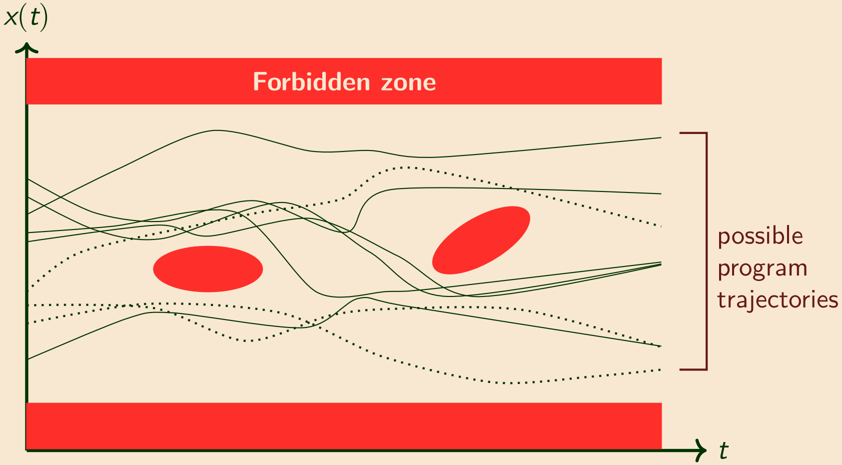
Testing cannot (generally) ensure complete absence of errors.



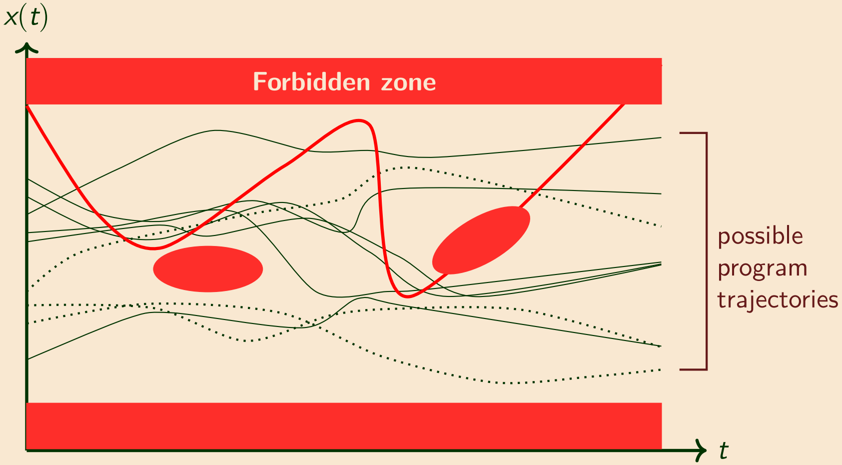
Testing cannot (generally) ensure complete absence of errors.



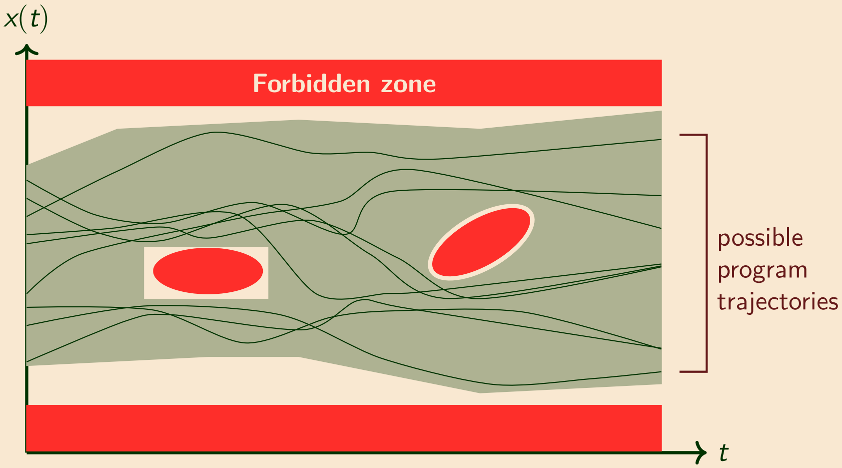
Testing cannot (generally) ensure complete absence of errors.



Testing cannot (generally) ensure complete absence of errors.



Idea: over-approximate all traces to **ensure absence of errors**.



The AI recipe[2]

[2] Adapted from Isil Dillig's *Abstract Interpretation* slides

Three-part recipe

1. An **abstract domain** that captures some aspect of program invariants
   (e.g. $n \leq x \leq m$ (x always lies within some *interval*))

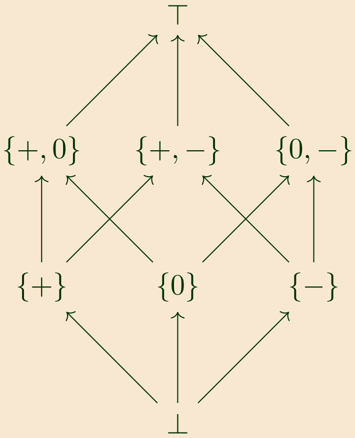
2. An **abstract semantics** that symbolically interprets each program construct
   (e.g. given invariants on x and y, what are the invariants on $x + y$?)

3. Iterate until **fixed point**

Example: *sign* abstract domain

Functions: **concretization** (γ) and **abstraction** (α) map between abstract values & sets of concrete values:

$$
\begin{aligned}
\gamma(\{+,-\}) &= \{x \in \mathbb{Z} \mid x \neq 0\} \\
&\dots \\
\alpha(\{-1,-2,4\}) &= \{+,-\} \\
&\dots
\end{aligned}
$$



Abstract semantics for $+$

$+$	$\bot$	$\{+\}$	$\{0\}$	$\{-\}$	$\{+,0\}$	$\{+,-\}$	$\ldots$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\ldots$
$\{+\}$	$\bot$	$\{+\}$	$\{+\}$	$\top$	$\{+\}$	$\top$	$\ldots$
$\{0\}$	$\bot$	$\{+\}$	$\{0\}$	$\{-\}$	$\{+,0\}$	$\{+,-\}$	$\ldots$
$\{-\}$	$\bot$	$\top$	$\{-\}$	$\{-\}$	$\top$	$\top$	$\ldots$
$\{+,0\}$	$\bot$	$\{+\}$	$\{+,0\}$	$\top$	$\{+,0\}$	$\top$	$\ldots$
$\{+,-\}$	$\bot$	$\top$	$\{+,-\}$	$\top$	$\top$	$\top$	$\ldots$
$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$	$\ldots$

Fixed points

```
int x = 2;
int y = 0;
while (y != z) {
   if (f y) x = x + 1;
   y = y + x
}
/* What do we know
 about x and y here? */
```

Evolution of x and y:

	x	y
0	$\{+\}$	$\{0\}$
1	$\{+\}$	$\{+,0\}$
2	$\{+\}$	$\{+,0\}$

(Generally: fixed point calculation may not terminate; we may need **widening**.)

Reading

Reading 1: Octagons

The Octagon Abstract Domain

Antoine Miné

École Normale Supérieure de Paris, France,
mine@di.ens.fr,
http://www.di.ens.fr/~mine

***Abstract*— This article presents a new numerical abstract domain for static analysis by abstract interpretation. It extends a former numerical abstract domain based on Difference-Bound Matrices and allows us to represent invariants of the form $(\pm x \pm y \leq c)$, where x and y are program variables and c is a real constant.**

We focus on giving an efficient representation based on Difference-Bound Matrices—$\mathcal{O}(n^2)$ memory cost, where n is the number of variables—and graph-based algorithms for all common abstract operators—$\mathcal{O}(n^3)$ time cost. This includes a normal form algorithm to test equivalence of representation and a widening operator to compute least fixpoint approximations.

***Index Terms*— abstract interpretation, abstract domains, linear invariants, safety analysis, static analysis tools.**

I. Introduction

This article presents practical algorithms to represent and manipulate invariants of the form $(\pm x \pm y \leq c)$, where x and y are numerical variables and c is a numeric constant. It extends the analysis we previously proposed in our PADO-II article [1]. Sets described by such invariants are special kind of polyhedra called *octagons* because they feature at most eight edges in dimension 2 (Figure 2). Using abstract interpretation, this allows discovering automatically common errors, such as division by zero, out-of-bound array access or deadlock, and more generally to prove safety properties for programs.

Our method works well for reals and rationals. Integer variables can be assumed, in the analysis, to be real in order to find approximate but safe invariants.

Example. The very simple program described in Figure 1 simulates M one-dimensional random walks of m steps and stores the hits in the array *tab*. Assertions in curly braces are discovered automatically by a simple static analysis using our octagonal abstract domain. Thanks to the invariants discovered, we have the guarantee that the program does not perform out-of-bound array access at lines 2 and 10. The difficult point in this example is the fact that the bounds of the array *tab* are not known at the time of the analysis; thus, they must be treated symbolically.

For the sake of brevity, we omit proofs of theorems in this article. The complete proof for all theorems can be found in the author's Master thesis [2].

II. Previous Work

A. Numerical Abstract Domains.

Static analysis has developed a successful methodology, based on the abstract interpretation framework—see Cousot and Cousot's POPL'77 paper [3]—to build analyzers that discover invariants automatically: all we need is an *abstract domain*, which is a practical representation of the invariants we want to study, together with a fixed set of operators and transfer functions (union, intersection, widening, assignment, guard, etc.) as described in Cousot and Cousot's POPL'79 article [4].

There exists many *numerical abstract domains*. The most used are the lattice of *intervals* (described in Cousot and Cousot's ISOP'76 article [5]) and the lattice of *polyhedra* (described in Cousot and Halbwachs's POPL'78 article [6]). They represent, respectively, invariants of the form $(v \in [c_1; c_2])$ and $(\alpha_1 v_1 + \cdots + \alpha_n v_n \leq c)$, where $v, v_1, \ldots, v_n$ are program variables and $c, c_1, c_2, \alpha_1, \ldots, \alpha_n$ are constants. Whereas the interval analysis is very efficient—linear memory and time cost—but not very precise, the polyhedron analysis is much more precise (Figure 2) but has a huge memory cost—in practice, it is exponential in the number of variables.

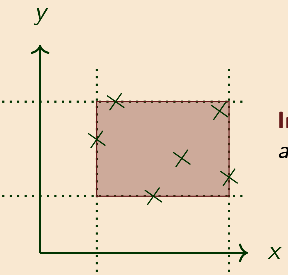
Remark that the correctness of the program in Figure 1 depends on the discovery of invariants of the form $(a \in [-m, m])$ where m must not be treated as a constant, but *as a variable*—its value is not known at analysis time. Thus, this example is beyond the scope of interval analysis. It can be solved, of course, using polyhedron analysis.

B. Difference-Bound Matrices.

Several satisfiability algorithms for set of constraints involving only two variables per constraint have been proposed in order to solve *Constraint Logic Programming (CLP)* problems. Pratt analyses, in [7], the simple case of constraints of the

```
1   int tab[−m ... m];
2   for i = −m to m   tab[i] = 0;   {−m ≤ i ≤ m}
3   for j = 1 to M do
4     int a = 0;
5     for i = 1 to m
6       { 1 ≤ i ≤ m;  −i + 1 ≤ a ≤ i − 1 }
7       if rand(2) = 0
8           then a = a + 1;   { −i + 1 ≤ a ≤ i }
9           else a = a − 1;   { −i ≤ a ≤ i − 1 }
10    tab[a] = tab[a] + 1;   { −m ≤ a ≤ m }
11  done;
```

Fig. 1. Simulation of a random walk. The assertions in curly brackets $\{\ldots\}$ are discovered automatically and prove that this program does not perform index out of bound error when accessing the array *tab*.

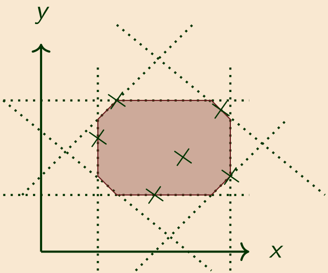


Interval domain

$a \leq x \leq b, c \leq x \leq d$

Octagon domain

$\pm x + \pm y < c$



AI^2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation

Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri*, Martin Vechev
Department of Computer Science
ETH Zurich, Switzerland

***Abstract*—We present AI^2, the first sound and scalable analyzer for deep neural networks. Based on overapproximation, AI^2 can automatically prove safety properties (e.g., robustness) of realistic neural networks (e.g., convolutional neural networks).**

The key insight behind AI^2 is to phrase reasoning about safety and robustness of neural networks in terms of classic abstract interpretation, enabling us to leverage decades of advances in that area. Concretely, we introduce abstract transformers that capture the behavior of fully connected and convolutional neural network layers with rectified linear unit activations (ReLU), as well as max pooling layers. This allows us to handle real-world neural networks, which are often built out of those types of layers.

We present a complete implementation of AI^2 together with an extensive evaluation on 20 neural networks. Our results demonstrate that: (i) AI^2 is precise enough to prove useful specifications (e.g., robustness), (ii) AI^2 can be used to certify the effectiveness of state-of-the-art defenses for neural networks, (iii) AI^2 is significantly faster than existing analyzers based on symbolic analysis, which often take hours to verify simple fully connected networks, and (iv) AI^2 can handle deep convolutional networks, which are beyond the reach of existing methods.

***Index Terms*—Reliable Machine Learning, Robustness, Neural Networks, Abstract Interpretation**

I. Introduction

Recent years have shown a wide adoption of deep neural networks in safety-critical applications, including self-driving cars [2], malware detection [44], and aircraft collision avoidance detection [21]. This adoption can be attributed to the near-human accuracy obtained by these models [21], [23].

Despite their success, a fundamental challenge remains: to ensure that machine learning systems, and deep neural networks in particular, behave as intended. This challenge has become critical in light of recent research [40] showing that even highly accurate neural networks are vulnerable to *adversarial examples*. Adversarial examples are typically obtained by slightly perturbing an input that is correctly classified by the network, such that the network misclassifies the perturbed input. Various kinds of perturbations have been shown to successfully generate adversarial examples (e.g., [3], [12], [14], [15], [17], [18], [29], [30], [32], [38], [41]). Fig. 1 illustrates two attacks, FGSM and brightening, against a digit classifier. For each attack, Fig. 1 shows an input in the Original column, the perturbed input in the Perturbed column, and the pixels that were changed in the Diff column. Brightened pixels are marked in yellow and darkened pixels are marked in purple. The FGSM [12] attack perturbs an image by adding to it a particular noise vector multiplied by a small number ϵ (in Fig. 1, $\epsilon = 0.3$). The brightening attack (e.g., [32]) perturbs an image by changing all pixels above the threshold $1 - \delta$ to the brightest possible value (in Fig. 1, $\delta = 0.085$).

Attack	Original	Perturbed	Diff
FGSM [12], $\epsilon = 0.3$			
Brightening, $\delta = 0.085$			

Fig. 1: Attacks applied to MNIST images [25].

Adversarial examples can be especially problematic when safety-critical systems rely on neural networks. For instance, it has been shown that attacks can be executed physically (e.g., [9], [24]) and against neural networks accessible only as a black box (e.g., [12], [40], [43]). To mitigate these issues, recent research has focused on reasoning about neural network robustness, and in particular on *local robustness*. Local robustness (or robustness, for short) requires that all samples in the neighborhood of a given input are classified with the same label [31]. Many works have focused on designing *defenses* that increase robustness by using modified procedures for training the network (e.g., [12], [15], [27], [31], [42]). Others have developed approaches that can show non-robustness by underapproximating neural network behaviors [1] or methods that decide robustness of small fully connected feedforward networks [21]. However, no existing sound analyzer handles convolutional networks, one of the most popular architectures.

Key Challenge: Scalability and Precision. The main challenge facing sound analysis of neural networks is scaling to large classifiers while maintaining a precision that suffices to prove useful properties. The analyzer must consider all possible outputs of the network over a prohibitively large set of inputs, processed by a vast number of intermediate neurons. For instance, consider the image of the digit 8 in Fig. 1 and suppose we would like to prove that no matter how we brighten the value of pixels with intensity above $1-0.085$, the network will still classify the image as 8 (in this example we have 84 such pixels, shown in yellow). Assuming 64-bit floating

*Rice University, work done while at ETH Zurich.

"Based on overapproximation, AI^2 can automatically prove safety properties (e.g., robustness) of realistic neural networks (e.g., convolutional neural networks).

"Our results demonstrate that:

i. AI^2 is precise enough to prove useful specifications (e.g., robustness),
ii. AI^2 can be used to certify the effectiveness of state-of-the-art defenses for neural networks,
iii. AI^2 is significantly faster than existing analyzers based on symbolic analysis, which often take hours to verify simple fully connected networks, and
iv. AI^2 can handle deep convolutional networks, which are beyond the reach of existing methods."

Reading 3: Verasco

A Formally-Verified C Static Analyzer

Jacques-Henri Jourdan
Inria Paris-Rocquencourt
jacques-henri.jourdan@inria.fr

Vincent Laporte
IRISA and U. Rennes 1
vincent.laporte@irisa.fr

Sandrine Blazy
IRISA and U. Rennes 1
sandrine.blazy@irisa.fr

Xavier Leroy
Inria Paris-Rocquencourt
xavier.leroy@inria.fr

David Pichardie
IRISA and ENS Rennes
david.pichardie@irisa.fr

Abstract

This paper reports on the design and soundness proof, using the Coq proof assistant, of Verasco, a static analyzer based on abstract interpretation for most of the ISO C 1999 language (excluding recursion and dynamic allocation). Verasco establishes the absence of run-time errors in the analyzed programs. It enjoys a modular architecture that supports the extensible combination of multiple abstract domains, both relational and non-relational. Verasco integrates with the CompCert formally-verified C compiler so that not only the soundness of the analysis results is guaranteed with mathematical certitude, but also the fact that these guarantees carry over to the compiled code.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—*Assertion checkers*, *Correctness proofs*; F.3.1 [*Logics and meanings of programs*]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*

Keywords static analysis; abstract interpretation; soundness proofs; proof assistants

1. Introduction

Verification tools are increasingly used during the development and validation of critical software. These tools provide guarantees that are always independent from those obtained by more conventional means such as testing and code review; often stronger; and sometimes cheaper to obtain (rigorous testing can be very expensive). Verification tools are based on a variety of techniques such as static analysis, model checking, deductive program proof, and combinations thereof. The guarantees they provide range from basic memory safety to full functional correctness. In this paper, we focus on static analyzers for low-level, C-like languages that establish the absence of run-time errors such as out-of-bound array accesses, null pointer dereference, and arithmetic exceptions. These basic properties are essential both for safety and security. Among the various verification techniques, static analysis is perhaps the one that scales best to large existing code bases, with minimal intervention from the programmer.

Static analyzers can be used in two different ways: as sophisticated bug finders, discovering potential programming errors that are hard to find by testing; or as specialized program verifiers, establishing that a given safety or security property holds with high confidence. For bug-finding, the analysis must be precise (too many false alarms render the tool unusable for this purpose), but no guarantee is offered nor expected that all bugs of a certain class will be found. For program verification, in contrast, *soundness* of the analysis is paramount: if the analyzer reports no alarms, it must be the case that the program is free of the class of run-time errors tracked by the analyzer; in particular, all possible execution paths through the program must be accounted for.

To use a static analyzer as a verification tool, and obtain certification credit in regulations such as DO-178C (avionics) or Common Criteria (security), evidence of soundness of the analyzer must therefore be provided. Owing to the complexity of static analyzers and of their input data (programs written in "big" programming languages), rigorous testing of a static analyzer is very difficult. Even if the analyzer is built on mathematically-rigorous grounds such as abstract interpretation [14], the possibility of an implementation bug remains. The alternative we investigate in this paper is *deductive formal verification of a static analyzer*: we apply program proof, mechanized with the Coq proof assistant, to the implementation of a static analyzer in order to prove its soundness with respect to the dynamic semantics of the analyzed language.

Our analyzer, called Verasco, is based on abstract interpretation; handles most of the ISO C 1999 language, with the exception of recursion and dynamic memory allocation; combines several abstract domains, both non-relational (integer intervals and congruences, floating-point intervals, points-to sets) and relational (convex polyhedra, symbolic equalities); and is entirely proved to be sound using the Coq proof assistant. Moreover, Verasco is connected to the CompCert C formally-verified compiler [26], ensuring that the safety guarantees established by Verasco carry over to the compiled code.

Mechanizing soundness proofs of verification tools is not a new idea. It has been applied at large scale to Java type-checking and bytecode verification [25], proof-carrying code infrastructures [1, 12], and verification condition generators for C-like languages [20, 23], among other projects. The formal verification of static analyzers based on dataflow analysis or abstract interpretation is less developed. As detailed in section 10, earlier work in this area either

POPL '15, January 15–17, 2015, Mumbai, India.

ACM 978-1-4503-3300-9/15/01…$15.00.
http://dx.doi.org/10.1145/2676726.2676966

"Verasco, a static analyzer based on abstract interpretation for most of the ISO C 1999 language (excluding recursion and dynamic allocation).

"Verasco establishes the absence of run-time errors in the analyzed programs. It enjoys a modular architecture that supports the extensible combination of multiple abstract domains, both relational and non-relational."

Writing suggestions

Abstract interpretation vs types

What are the relative benefits of AI and types?
(Are they in some sense the same thing?)

Cost vs precision

What is the tradeoff?

Widening and narrowing

What role do they play in convergence and precision?

Applicability

How widely applicable is abstract interpretation? How well does it scale up?

Relational and non-relational domains