Proving Programs Robust *

Swarat Chaudhuri
Rice University
swarat@rice.edu

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Roberto Lublinerman
Pennsylvania State University
rluble@psu.edu

Sara Navidpour
Pennsylvania State University
ssn123@psu.edu

ABSTRACT

We present a program analysis for verifying *quantitative robustness properties* of programs, stated generally as: "If the inputs of a program are perturbed by an arbitrary amount ϵ, then its outputs change at most by $K\epsilon$, where K can depend on the size of the input but not its value." Robustness properties generalize the analytic notion of continuity—e.g., while the function e^x is continuous, it is not robust. Our problem is to verify the robustness of a function P that is coded as an imperative program, and can use diverse data types and features such as branches and loops.

Our approach to the problem soundly decomposes it into two subproblems: (a) verifying that the *smallest possible* perturbations to the inputs of P do not change the corresponding outputs significantly, even if control now flows along a different control path; and (b) verifying the robustness of the computation along each control-flow path of P. To solve the former subproblem, we build on an existing method for verifying that a program encodes a *continuous function* [5]. The latter is solved using a static analysis that bounds the magnitude of the slope of any function computed by a control flow path of P. The outcome is a sound program analysis for robustness that uses proof obligations which do not refer to ϵ-changes and can often be fully automated using off-the-shelf SMT-solvers.

We identify three application domains for our analysis. First, our analysis can be used to guarantee the predictable execution of embedded control software, whose inputs come from physical sources and can suffer from error and uncertainty. A guarantee of robustness ensures that the system does not react disproportionately to such uncertainty. Second, our analysis is directly applicable to *approximate computation*, and can be used to provide foundations for a recently-proposed program approximation scheme called *loop perforation*. A third application is in database privacy: proofs of robustness of queries are essential to *differential privacy*, the most popular notion of privacy for statistical databases.

Categories and Subject Descriptors

F.3.2 [**Logics and Meanings of Programs**]: Semantics of programming languages—*Program analysis*.; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification*; G.1.0 [**Numerical Analysis**]: General—*Error analysis*

General Terms

Theory, Verification

Keywords

Lipschitz, Continuity, Quantitative Program Analysis, Sensitivity, Robustness, Uncertainty, Perturbations, Program Approximation.

1. INTRODUCTION

Uncertainty in computation [12] has long been a topic of interest to computer science. Depending on the context, uncertainty in the operation of programs can be a curse or a blessing. On one hand, uncertain operating environments may cause system failures—consider, for example, an aircraft controller that reacts unpredictably to noisy sensor data and causes a crash. On the other hand, randomized and approximate algorithms deliberately inject uncertainty into their data to trade off quality of results for better performance. Uncertainty of both forms is rife in a world where cyber-physical systems are ubiquitous [15] and applications suited to approximation and randomization are ascendant. Love or hate uncertainty, you increasingly cannot ignore it.

Robustness is a system property critical to reasoning about program behavior under uncertainty. A program is robust (in the sense of this paper) if a perturbation to its inputs can only lead to proportional changes in its outputs. This means that a robust avionic controller reacts predictably to noise in the measurements made by the plane's sensors. Also, if a program P is robust, then the output $P(x)$ of P on an input x can be approximated "safely" by $P(x')$, where x' is a value close to x—as x and x' are close, so must be $P(x)$ and $P(x')$. If uncertainty is our enemy, a proof of robustness shows that our program is relatively safe from it; if we want to introduce uncertainty in our computation for performance gains, robustness ensures that it is safe to do so.

*This research was supported by NSF CAREER Award #0953507 ("Robustness Analysis of Uncertain Programs: Theory, Algorithms, and Tools").

```
DIJK(G : graph, src : node)
 1  for each node v in G
 2      d[v] :=⊥;    prev[v] := UNDEF ;
 3  d[src] := 0;    WL := set of all nodes in G;
 4  while WL ≠ ∅
 5      choose node w ∈ WL such that d[w] is minimal;
 6      remove w from WL;
 7      for each neighbor v of w
 8              z := d[w] + G[w, v];
 9              if z < d[v]
10                  then    d[v] := z;    prev[v] := w
```

Figure 1: Dijkstra's shortest-path algorithm

A system to formally verify the robustness of everyday programs would then seem to be of considerable practical importance. A step to this end was taken by Chaudhuri et al [5], who presented a program analysis to verify that a program encodes a *continuous function*. A function is continuous if *infinitesimal*—or arbitrarily small—changes to its inputs can only cause infinitesimal changes to its outputs. This makes such a function robust in a sense. Such a formulation of robustness is particularly valuable in the setting of programs, where violation of robustness is often due to discontinuities introduced by control constructs like branches and loops. A provably continuous program is free from violations of this sort. At the same time, continuity is too weak a robustness property for many settings, as a small but non-infinitesimal change to the inputs of a continuous function can create disproportionately large changes to its outputs. For example, while the function e^x is continuous, there is no bound on the change in its output on a small finite change to its input x.

In this paper, we investigate a stronger, quantitative formulation of robustness of programs that does not suffer from this limitation. We believe that this formulation, based on the analytic notion of *Lipschitz continuity*, is a canonical notion of robustness for programs. By this definition, a program is robust if a change of $\pm\epsilon$ to its inputs, for any ϵ, results in a change of $\pm K\epsilon$ to its outputs, where K does not depend on the values of the input variables. The multiplier K—known as the *robustness parameter*—quantifies the extent of this robustness.

For example, consider the implementation *Dijk* of Dijkstra's shortest-paths algorithm in Fig. 1—here G is a graph with real-valued edge-weights and N edges, and *src* is the source node. The output of the program is the table d of shortest-path distances in G. We note that *Dijk* is robust with a robustness parameter N (from now on, *N-robust*): if each edge-weight in G changes by $\pm\epsilon$, then each output $d[i]$ changes at most by $\pm N\epsilon$. But how do we verify the above robustness property from the text of programs like *Dijk*, which use features like branches, loops, and arrays?

One way is to first prove that *Dijk* computes shortest paths, and then to establish that the costs of these paths change proportionally on changes to the edge-weights of G. Such a proof, however, would be highly specialized and impossible to automate. Our goal, instead, is to develop a proof system that reasons about robustness without having to prove full functional correctness, is applicable to a wide range of algorithms, and is mostly automated.

To see how we achieve this goal, consider programs like *Dijk* whose inputs and outputs are from dense domains. Key to our analysis of robustness for such programs P is the following metatheorem: P is K-robust if, first, P encodes a continuous function, and, second, each control flow path of P computes a function that is linear in the values (but not the size) of the input, and the magnitude of the slope of this line is bounded by K. The first property accounts for the possibility of different control flow on the perturbed and unperturbed input, and can be verified using an existing analysis [5]. As for the latter property, we offer a new static analysis for it. The final outcome is a program analysis that can verify the robustness of programs over continuous data types, can be automated using off-the-shelf SMT-solvers (we provide an implementation on top of the Z3 solver), and can verify the robustness of many everyday programs. For example, we establish the property for the program *Dijk* by showing that: (1) The effect of each loop iteration on the array d can be written as $d[v_1] := c_1 \cdot d[v_2] + c_2 \cdot a + c_3$, where a is the weight of some edge of G, $d[v_1]$ and $d[v_2]$ are elements of d, and c_1, c_2, c_3 are constants with $|c_1| \leq 1$ and $|c_2| \leq 1$; and (2) Each edge-weight of the input graph G is used *only once* as an operand during an addition.

We identify three application domains for our analysis. First, our analysis can be used to guarantee the predictable execution of embedded control programs, whose inputs come from physical sources and can therefore be noisy or uncertain. Second, we investigate the application of our system in approximate computations that trade off accuracy of results for resource savings. In particular, we demonstrate that a robustness analysis can be used to provide foundations for a recently-proposed program approximation heuristic called *loop perforation* [20]. Third, our analysis can be used in the synthesis of information release mechanisms that satisfy *differential privacy* [10], perhaps the most popular definition of privacy for statistical databases.

Summary of contributions and organization

Now we summarize this paper's contributions:

- We give a canonical, quantitative definition of robustness of programs (Sec. 2).
- We present a sound program analysis that can be used to verify the robustness of a given program (Sec. 3). (Sec. 5).
- We present a prototype implementation of our proof system, built on top of the Z3 SMT-solver (Sec. 5).
- We identify three application domains for our analysis. (Sec. 4)

2. ROBUSTNESS OF PROGRAMS

Now we formalize our notion of robustness of programs. We begin by fixing a language IMP of imperative arithmetic programs. For simplicity, we allow IMP only two data types: reals (`real`) and arrays of reals (`realarr`). Other popular types such as records, tuples, and functional lists/trees can be added without changing the analysis significantly. Our robustness analysis can be extended to programs with discrete typed inputs such as integers. We intend to present this extension in a future work. Also, we assume that reals in IMP are infinite-precision rather than floating-point, and treat arithmetic and comparison operations on them as unit-time oracles. Thus, our programs are equivalent to Blum-Shub-Smale Turing machines [2]. While this idealized semantics rules out reasoning about floating-point rounding errors, we can use it to prove the absence of robustness bugs

due to flawed logic (arguably, it is this semantics that forms the mental model of programmers as they design numerical algorithms). We intend to pursue a floating-point modeling of continuous data in future work.

As for perturbations, they can change the *value* of a datum but not its type or size, the latter being 1 if the datum is a real, and N if it is an array of length N. We assume, for each type τ and size N, a *metric*[1] $d_{\tau,N}$. An ϵ-change to a value x of type τ and size N is assumed to result in a value y, of type τ and size N, such that $d_{\tau,N}(x,y) = \epsilon$. In particular, the type `real` is associated with the Euclidean metric, defined as $d_{\texttt{real},1}(x,y) = |x-y|$. We let an ϵ-change to an array consist of ϵ-changes to any number of its elements. Formally, the metric over arrays of length N whose elements are of type τ is the *L_∞-norm*:

$$d_{\text{array of } \tau,N}(A,B) = \max_i \{d_\tau(A[i],B[i])\}.$$

Now we offer the syntax of arithmetic expressions e, boolean expressions b, and programs P in Imp:

$$\begin{array}{lcl} e & ::= & x \mid c \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid A[i] \\ b & ::= & e > 0 \mid e = 0 \mid b_1 \wedge b_2 \mid \neg b \\ P & ::= & \textbf{skip} \mid x := e \mid A[i] := e \mid \textbf{if } b \textbf{ then } P_1 \textbf{ else } P_2 \\ & & \mid \textbf{while } b \textbf{ do } P_1 \mid P_1; P_2 \end{array}$$

Here x is a variable, c is a constant, A is an array variable, i an integer variable or constant, and the arithmetic and boolean operators are as usual. We let each statement be annotated with a distinct *label*.

As for semantics, let us only consider programs P that *terminate on all inputs*. The semantics of P is the standard denotational semantics [27] for imperative programs (except as mentioned earlier, we assume unit-time operations on reals). Formally, let us associate with each variable x a set $Cloc(x)$ of *concrete memory locations*. A *state* of P is a map σ that assigns a value in Val_τ to each program variable x of type τ. We denote the set of all states of P by $\Sigma(P)$. Each state induces, in the usual way, an assignment of *contents* to each location $y \in Cloc(x)$, for each variable x. We use the notation $\sigma(y)$ to denote the content of location y at state σ. The semantics of the program P, and an expression e of type τ appearing in it, are now defined by two functions $[\![P]\!] : \Sigma(P) \to \Sigma(P)$ and $[\![e]\!] : \Sigma(P) \to Val_\tau$, where Val_τ is the set of values of type τ. Intuitively, $[\![e]\!](\sigma)$ is the value of e at state σ, and $[\![P]\!](\sigma)$ is the state at which P terminates after starting execution from σ. In addition, we assume definitions of *control flow paths* (sequences of labels) and *executions* (sequences of states) of P. These definitions are all standard, and hence omitted.

Robustness of programs. Our definition of robustness of programs is based on the analytic notion of *Lipschitz continuity* [1]. Intuitively, a program P is K-robust if any additive ϵ-change to the input of P can only change the output of P by $\pm K\epsilon$. Note that ϵ is arbitrary, so that the output of a K-robust program changes proportionally on *any* change to the inputs, and not just small ones.

As a program can have multiple inputs and outputs, we define robustness with respect to an *input variable* x_{in} and an *output variable* x_{out}. If P is robust with respect to x_{in} and x_{out}, a change to the initial value of any x_{in}, while keeping the remaining variables fixed, must only cause a proportional change to the final value of x_{out}. Variables other than x_{out} can change arbitrarily.

Also, we allow robustness parameters that are not just constants, but depend on the *size of the input*. For example, suppose the size of x_{in} is N, and an ϵ-change to it changes the output by $N\epsilon$. Then P is *N-robust* with respect to x_{in}. We model this by letting a robustness parameter K be a function of type $\mathbb{N} \to \mathbb{R}$, rather than just a real.

Finally, our definition allows a program to be robust only within a certain subset Σ' of the input space, without assertions about the effect of perturbations on states outside Σ' (capturing the fact that many realistic programs are robust only within certain regions of their input space).

Formally, for a variable x (say of type τ) and a state σ, let $Size(x,\sigma)$ be the size of the value of x at σ. Now let $\epsilon \in \mathbb{R}^+$; also let $\sigma' \in \Sigma(P)$ such that $\sigma(x,\sigma) = Size(x,\sigma') = N$. The state σ' is an *(ϵ,x)-perturbation* of σ, and is denoted by $Pert_{\epsilon,x}(\sigma,\sigma')$, if $d_{\tau,N}(\sigma(x),\sigma'(x)) < \epsilon$, and for all other variables y, we have $\sigma(y) = \sigma'(y)$. The states σ and σ' are *(ϵ,x)-close* (written as $\sigma \approx_{\epsilon,x} \sigma'$) if $d_{\tau,m}(\sigma(x),\sigma'(x)) < \epsilon$. Now we define:

Definition 1 (Robustness of programs). *Consider a function $K : \mathbb{N} \to \mathbb{R}$ and a set of states $\Sigma' \subseteq \Sigma$. The program P is* K-robust *within Σ' with respect to the input x_{in} and the output x_{out} if for all $\sigma, \sigma' \in \Sigma'$ and $\epsilon \in \mathbb{R}^+$, we have*

$$Pert_{\epsilon,x_{in}}(\sigma,\sigma') \Longrightarrow [\![P]\!](\sigma) \approx_{m,x_{out}} [\![P]\!](\sigma')$$

where $m = K(Size(x_{in},\sigma)) \cdot \epsilon$. □

Definition 2 (Continuity of programs [5]). *The program P is continuous within $\Sigma' \subseteq \Sigma$ with respect to the input x_{in} and the output x_{out} if for all $\epsilon \in \mathbb{R}^+, \sigma \in \Sigma'$, there exists a $\delta \in \mathbb{R}^+$ such that for all $\sigma' \in \Sigma'$,*

$$Pert_{\delta,x_{in}}(\sigma,\sigma') \Longrightarrow [\![P]\!](\sigma) \approx_{\epsilon,x_{out}} [\![P]\!](\sigma').$$ □

If P is continuous by the above, then *infinitesimal* perturbations to x_{in} (that keep the state within the set Σ') can only cause infinitesimal changes to x_{out}. Not all continuous programs are robust. For example, a program computing x^2, given arbitrary $x \in \mathbb{R}$, is continuous but non-robust. Now we consider a few everyday programs that are robust or continuous by the above definitions:

Example 1 (Sorting). *Consider a correct implementation P of a sorting algorithm that takes in an array A_{in} of reals, and returns a sorted array A_{out}. The program is 1-robust, with respect to input A_{in} and output A_{out}, within $\Sigma(P)$: for any $\epsilon > 0$, if each element of A_{in} is perturbed at most by $\pm\epsilon$, then the maximum change to an element of the output A_{out} is $\pm\epsilon$ as well. Note that this observation is not at all obvious, as we are speaking of* arbitrary *changes to A_{in} here, and as even the minutest change to A_{in} can alter the position of a given item in A_{out} arbitrarily.*

Example 2 (Shortest paths, MSTs). *Let SP be a correct implementation of a shortest-path algorithm (e.g., Dijk; Fig. 1). We view the graph G on which SP operates as a perturbable array of reals such that $G[i]$ is the weight of the i-th edge. An ϵ-change to G thus amounts to a maximum*

[1]Recall that a *metric* over a set S is a function $d : S \times S \to \mathbb{R} \cup \{\infty\}$ such that for all x, y, z, we have: (1) $d(x,y) \geq 0$, with $d(x,y) = 0$ iff $x = y$; (2) $d(x,y) = d(y,x)$; and (3) $d(x,y) + d(y,z) \geq d(x,z)$.

```
KRUSKAL(G : graph)
1  for each node v in G do  C[v] := {v};
2  WL := set of all edges in G; cost := 0; T := ∅;
3  while WL ≠ ∅
4      choose edge (v, w) ∈ WL
           such that G(v, w) is minimal;
5      remove (v, w) from WL;
6      if C[v] ≠ C[w] then
7          add edge (v, w) to T;
8          cost := cost + G(v, w);
9          C[v] := C[w] := C[v] ∪ C[w];
```

Figure 2: Kruskal's algorithm

change of $\pm\epsilon$ to any edge-weight of G, while keeping the node and edge structure intact.

One output of SP is the array d of shortest-path distances in G—i.e., $d[i]$ is the length of the shortest path from the source node src to the i-th node u_i of G. A second output is the array π whose i-th element is a sequence of nodes forming a minimal-weight path between src and u_i. Let the distance between two elements of π be 0 if they are identical, and ∞ otherwise.

As it happens, SP is N-robust everywhere within $\Sigma(P)$ with respect to the output d—if each edge weight in G changes by an amount ϵ, a shortest path weight can change at most by $(N\epsilon)$. However, an ϵ-change to G may add or subtract elements from π—i.e., perturb π by the amount ∞. Therefore, SP is not K-robust with respect to the output π for any K.

Similar arguments apply to a program MST computing minimum spanning trees in a graph G (Kruskal's algorithm; Fig. 2). Suppose the program has two outputs: a sequence T of edges forming a minimum spanning tree, and the cost of this tree. MST is N-robust within $\Sigma(MST)$ if the output is cost, but not robust if the output is T.

3. VERIFYING ROBUSTNESS

In this section, we present our program analysis for robustness. The inputs of the analysis are a program P, symbolic encodings of a set Σ' of $\Sigma(P)$ and a function $K : \mathbb{N} \to \mathbb{R}$, an input variable x_{in}, and an output variable x_{out}. Our goal is to soundly judge P K-robust within Σ' with respect to x_{in} and x_{out}.

3.1 Piecewise K-robustness and K-linearity

Consider, first, the simple scenario where P has a single real-valued variable x. Note that each control flow path of P computes a differentiable function over the inputs. Now suppose we can show that each control flow path of P represents a robust computation (in this case, P is said to be *piecewise K-robust*). Piecewise robustness does not entail robustness: a perturbation to the initial value of x can cause P to execute along a different control flow path, leading to a completely different final state. However, if P is *continuous* as well as piecewise K-robust, then P is K-robust as well, e.g. the function $abs(x) = |x|$, where $x \in \mathbb{R}$, is continuous as well as piecewise 1-robust—hence 1-robust. On the other hand, the continuous function "$\mathbf{if}(x > 0)\mathbf{then}x^2\mathbf{else}x$" is nonrobust because x^2 is not piecewise robust within $x \in \mathbb{R}$.

The above observation can be generalized to settings where P has multiple variables of different types. Our analysis exploits it to decompose the problem of robustness analysis into two independent subproblems: that of verifying continuity and piecewise K-robustness of P.

For any program P and any set of states Σ' of P, let $\Sigma'_{(i)}$ denote the set of states $\sigma \in \Sigma'$ such that starting from σ, P executes along its i-th control flow path (we assume a global order on control flow paths). Let us now define:

DEFINITION 3 (PIECEWISE K-ROBUSTNESS). *Let P be a program, $\Sigma' \subseteq \Sigma(P)$ a set of states, K a function of type $\mathbb{N} \to \mathbb{R}$, and $x_{in}, x_{out} \in Var(P)$. P is* piecewise K-robust *within $\Sigma' \subseteq \Sigma(P)$ with respect to input x_{in} and output x_{out} if for all i, P is K-robust within $\Sigma_{(i)}$ with respect to x_{in} and x_{out}.*

We establish piecewise robustness using the weaker property of *piecewise K-linearity*, which says that the function computed by each control flow path of P is a linear function, and that the absolute value of its slope is bounded by K:

DEFINITION 4 (PIECEWISE K-LINEARITY). *Let P be a program, $\Sigma' \subseteq \Sigma(P)$ a set of states, K a function of type $\mathbb{N} \to \mathbb{R}$, and $x_{in}, x_{out} \in Var(P)$. P is* K-linear *within Σ' w.r.t. input x_{in} and output x_{out} if for each $z \in Cloc(x_{in})$, $y \in Cloc(x_{out})$, $\sigma \in \Sigma'$, we have the relationship*

$$([\![P]\!](\sigma))(y) \quad = \quad \left(\sum_{z \in Cloc(x_{in})} c_{z,y} \cdot z \right) + \tau,$$

where τ is an expression whose free variables range over the set $\bigcup_{x' \neq x_{in}} Cloc(x')$, and $\sum_z |c_{z,y}| \leq K(Size(x_{in}, \sigma))$. P is piecewise K-linear *within $\Sigma' \subseteq \Sigma(P)$ with respect to input x_{in} and output x_{out} if for all i, P is K-linear within $\Sigma_{(i)}$ with respect to x_{in} and x_{out}.*

It is not hard to see that:

THEOREM 1. *If P is piecewise K-linear, then P is piecewise K-robust.*

EXAMPLE 3 (DIJKSTRA'S ALGORITHM). *Consider, once again, the procedure Dijk in Fig. 1. While the dependence between G and d is complex, Dijk is piecewise N-robust in G and d. To see why, consider any control flow path π of Dijk and view it as a straight-line program. Suppose the addition operation in Line 8 of Dijk is executed M times in this program. As we only remove elements from the worklist WL, a specific edge $G[w, v]$ is used at most once as an operand of this addition. Consequently, we have $M \leq N$, where N is the size of G. Let M' be the number of times Line 10 assigns the result of this addition to an element of d. We have $M' \leq M \leq N$. It is easy to see that this means that π is N-linear with respect to input G and output d. As the set of control flow paths in Dijk is countable and each path is N-linear with respect to G and d, Dijk is piecewise N-robust within $\Sigma(Dijk)$ with respect to G and d.*

3.2 Robustness

Now we apply the notion of piecewise robustness and piecewise linearity in the analysis of robustness.

THEOREM 2. *Let P be a program, $\Sigma' \subseteq \Sigma(P)$, and x_{in} and x_{out} be variables of dense types. P is K-robust within*

```
INSERTION-SORT(A : realarr)
1  for i := 1 to (|A| − 1)
2      z := A[i]; j := i − 1;
3      while j ≥ 0 and A[j] > z
4          A[j + 1] := A[j]; j := j − 1;
5      A[j + 1] := z;
```

Figure 3: Insertion sort

Σ' with respect to input x_{in} and output x_{out} if and only if: (1) P is continuous within Σ' w.r.t. input x_{in} and output x_{out}; and (2) P is piecewise K-robust within Σ' with respect to input x_{in} and output x_{out}.

By Theorem 2, the problem of robustness analysis can be decomposed soundly and completely into the problems of verifying continuity and piecewise robustness. We establish these conditions independently. The first criterion is proved using a sound program analysis due to Chaudhuri et al [5] (from now on, we call this system CONT). To prove the second property, we prove P to be piecewise K-linear, and use Theorem 1. However, for reasons outlined later, no existing sound abstraction that we know of is suited to precise and efficient analysis of piecewise linearity—a new solution is needed.

Piecewise linearity using arithmetic-freedom

It is sometimes possible to establish piecewise linearity using traditional dataflow analysis. Let a program Q be free of arithmetic operations—i.e., if $x := e$ is an assignment in the program, then the evaluation of e does not require arithmetic. In this case, each control flow path in Q encodes a 1-linear function, which means that Q is piecewise 1-linear. Generalizing, let a program P be *arithmetic-free* with respect to input x_{in} and outputs x_{out} if all data flows from x_{in} to x_{out} are free of arithmetic operations. A program can be shown arithmetic-free in this sense using standard slicing techniques. We can use a program like Q above as an abstraction of P. Application in the verification of piecewise robustness stems from the fact that:

THEOREM 3. *If a program P is arithmetic-free within $\Sigma' \subseteq \Sigma(P)$ with respect to input x_{in} and output x_{out}, then P is piecewise 1-linear within Σ' with respect to input x_{in} and output x_{out}.*

EXAMPLE 4 (SORTING). *The seemingly trivial abstraction of arithmetic-freedom can in fact be used to prove the robustness of several challenging, array-manipulating algorithms. Consider Insertion Sort (Fig. 3), where the array A is the input as well as the output. A lightweight analysis can prove this algorithm arithmetic-free, at all input states, with respect to the input A and output A. While arithmetic does occur in the program, it only updates the index i, whose value does not depend on the original contents of A. Other algorithms like Mergesort and Bubblesort can be proved arithmetic-free in the same way. Separately, we prove these algorithms continuous using* CONT*, which gives us a proof of 1-robustness.*

Piecewise linearity with robustness matrices

In most realistic programs, however, arithmetic-freedom will not suffice, and some form of quantitative reasoning will be necessary. A natural first question is: can we use a traditional numerical abstract domain—such as polyhedra [9]—for such reasoning? The answer, unfortunately, seems to be no. Consider the program

if $(x + y > 0)$ **then** $z := x + y$ **else** $z := -x - y$

Our goal is to prove this program piecewise 1-linear with respect to input x and output y. Unfortunately, the best invariant that we can establish at the end of the branch using the polyhedra domain is $(z \geq x + y) \wedge (z \geq -x - y)$. These constraints permit z to be a linear function of x with slope equal to ∞—hence the best we can say is that the program is ∞-robust! The key issue here is that we need a form of *disjunctive* reasoning to track quantities (absolute values of slopes) computed along different paths. While there is a plethora of abstract domains for disjunctive reasoning about programs [6, 7], none of them, so far as we know, is suitable for this purpose. Now we present a simple static analysis that fulfils our needs.

Here, an abstract state—known as a *robustness matrix*—tracks, for each pair of memory locations x and y, a bound on the slope of the expression relating the current value of x to the initial value of y. This information is propagated through the program using an abstract interpretation.

For brevity, we make a few simplifying assumptions in this presentation. First, we assume that all variables of P are real valued. The analysis that we have actually implemented can handle arrays by abstracting each unbounded array using a finite number of abstract memory locations. As this array abstraction is standard but adds significantly to the notation, we omit its details. Also, we only show how to derive piecewise linearity judgments holding over the entire space $\Sigma(P)$. A generalization to judgments conditioned by $\Sigma' \subseteq \Sigma(P)$ is, however, easy to construct. Finally, we view a program state not as a map but as a vector $\langle r_1, \ldots, r_n \rangle$, where r_i is the value of the i-th variable x_i. The denotational semantics of any program Q is thus a function

$$\text{(Skip)}\ \frac{}{\mathbf{skip} \vdash \mathbf{I}}$$

$$\text{(Assign)}\ \frac{}{x_i := e \vdash \nabla} \quad \text{where}$$

$$\forall j,k : \nabla_{jk} = \begin{cases} \left|\frac{\partial e}{\partial x_k}\right| & \text{if } j = i \text{ and } \frac{\partial e}{\partial x_k} \text{ is constant} \\ \infty & \text{if } j = i \text{ and } \frac{\partial e}{\partial x_k} \text{ depends on the } x_m\text{-s} \\ 1 & \text{if } j = k \neq i \\ 0 & \text{otherwise} \end{cases}$$

$$\text{(Weaken)}\ \frac{P' \vdash \nabla \qquad \forall i,j : \nabla_{ij} \leq \nabla'_{ij}}{P' \vdash \nabla'}$$

$$\text{(Sequence)}\ \frac{P_1 \vdash \nabla_1 \qquad P_2 \vdash \nabla_2}{P_1; P_2 \vdash \nabla_2 \cdot \nabla_1}$$

$$\text{(Ite)}\ \frac{P_1 \vdash \nabla_1 \qquad P_2 \vdash \nabla_2}{\mathbf{if}\ b\ \mathbf{then}\ P_1\ \mathbf{else}\ P_2 \vdash \max(\nabla_1, \nabla_2)}$$

$$\text{(While-1)}\ \frac{P' = \mathbf{while}\ b\ \mathbf{do}\ P'' \qquad P'' \vdash \nabla'' \qquad \mathit{Bound}^+(P', M) \qquad \forall i,j : (\nabla''_{ij} = 0 \vee \nabla''_{ij} \geq 1)}{P' \vdash (\nabla'')^M}$$

$$\text{(While-2)}\ \frac{P' = \mathbf{while}\ b\ \mathbf{do}\ P'' \qquad P'' \vdash \nabla'' \qquad \mathit{Bound}^-(P', M) \qquad \forall i,j : \nabla''_{ij} < 1}{P' \vdash (\nabla'')^M}$$

Figure 4: System ROBMAT for propagating robustness matrices

1 $x := a$ $\{\nabla_{x,a} = 1\}$
2 $y := b$ $\{\nabla_{x,a} = 1, \nabla_{y,b} = 1\}$
3 **if** $(x \leq y)$
4 **then** $z := -2x + y$
$\{\nabla_{x,a} = 1, \nabla_{y,b} = 1, \nabla_{z,x} = 2, \nabla_{z,y} = 1, \nabla_{z,a} = 2, \nabla_{z,b} = 1\}$
5 **else** $z := 3y + x$; $\{\nabla_{x,a} = 1, \nabla_{y,b} = 1, \nabla_{z,x} = 1, \nabla_{z,y} = 3, \nabla_{z,a} = 1, \nabla_{z,b} = 3\}$
$\{\nabla_{x,a} = 1, \nabla_{y,b} = 1, \nabla_{z,x} = 2, \nabla_{z,y} = 3, \nabla_{z,a} = 2, \nabla_{z,b} = 3\}$

Figure 5: Piecewise robustness using robustness matrices

$[\![Q]\!] : \mathbb{R}^n \to \mathbb{R}^n$. Now we define a *robustness matrix* ∇ for P to be an $n \times n$ matrix whose elements ∇_{ij} are non-negative reals.

To understand the interpretation of this matrix, we recall the classic definition of a *Jacobian* from vector calculus. The Jacobian of a function $f : \mathbb{R}^n \to \mathbb{R}^n$ with inputs $x_1^{in}, \ldots, x_n^{in} \in \mathbb{R}$ and outputs $x_1, \ldots, x_n \in \mathbb{R}$ is the matrix

$$J_f = \begin{pmatrix} \frac{\partial x_1}{\partial x_1^{in}} & \cdots & \frac{\partial x_1}{\partial x_n^{in}} \\ \cdots & & \cdots \\ \frac{\partial x_n}{\partial x_1^{in}} & \cdots & \frac{\partial x_n}{\partial x_n^{in}} \end{pmatrix}$$

If f is a differentiable function, then for each x_i and x_j^{in}, it is K-robust with respect to input x_j and output x_i, where K is any upper bound on $|\frac{\partial x_i}{\partial x_j^{in}}|$. In our setting, the expression relating the inputs and outputs of a single control flow path is differentiable; consequently, we can verify the robustness of this expression by propagating a Jacobian along it (strictly, entries in the actual matrix that we carry are constant upper bounds on the absolute values of the terms $\partial x_i / \partial x_j^{in}$). It is possible to carry out this propagation using the chain rule of differentiation. Of course, due to branches, a program P need not be differentiable. This is where abstract interpretation comes handy—we *merge* multiple Jacobians propagated along different paths into a *robustness matrix* that overestimates the robustness parameter of P. Such a merge demands an abstract join operator $\sqcup$: for robustness matrices ∇ and ∇', we define $(\nabla \sqcup \nabla')$ to be the matrix M such that for all i, j, we have $M_{ij} = \max(\nabla_{ij}, \nabla'_{ij})$.

Note that with the above strategy, we will infer robustness matrices even for discontinuous programs. This is, of course, acceptable, as the present analysis only verifies piecewise robustness—continuity is judged separately by Cont.

The goal of our analysis—call it Robmat—is to syntactically derive facts of the form $P' \vdash \nabla$, read as: "∇ is the robustness matrix for the subprogram P'." The structural rules for the analysis are shown in Fig. 4. Here, $\mathbf{I}$ is the identity matrix. The assertion $Bound^+(P', M)$ states that M is an upper bound on the number of iterations of the while-loop P'; likewise, $Bound^-(P', M)$ states that the symbolic or numeric constant M is a lower bound on the number of iterations for which P' executes. These conditions can be established either via an auxiliary checker or by manual annotation.

We observe that the robustness matrix for $P_1; P_2$ obtained by multiplying the matrices for P_1 and P_2—this rule follows from the chain rule of calculus. This rule is now generalized into the rule While-1 for while-loops. As the loop may terminate after $M' < M$ iterations, we require the following extra condition for the rule to be sound: $(\nabla'')^i \leq (\nabla'')^{i+1}$ for all $i < M$. This property is ensured by the condition $\forall i, j : (\nabla''_{ij} = 0 \vee \nabla''_{ij} \geq 1)$. Note that in the course of a proof, we can weaken *any* robustness matrix that does not satisfy this condition to one that does, using the Weaken rule. On the other hand, the While-2 rule applies to the special case when all matrix elements are less than 1—that is, the matrix represents a contraction.

P ::= ⟨all syntactic forms in Imp⟩ | ⟨the form Q below⟩

l: **while** b
$\theta :=$ nondeterministically chosen $u \in U$;
$R(U, \theta, x_{in}[\theta], x_{out})$

Figure 6: The language Limp (P represents programs).

We have:

Theorem 4. *If the system* Robmat *derives the judgment $P \vdash \nabla$, then for all i, j, P is piecewise ∇_{ij}-linear within $\Sigma(P)$ with respect to the input x_j and the output x_i.*

Example 5. *Fig. 5 shows the result of applying a dataflow analysis based on* Robmat *to a simple program. The annotations depict the robustness matrices ∇ propagated to the various program points—we use the more readable notation $\nabla_{y,z}$ to refer to the matrix entry ∇_{ij} if y and z are respectively the i-th and j-th variables. Observe, in particular, how the robustness matrices from the two branches of the program are merged.*

Piecewise linearity with linear loops

A problem with the robustness matrix abstraction is that it does not satisfactorily handle loops iterating over unbounded data structures. For example, let us try to use it to prove Dijkstra's algorithm (Fig. 1) piecewise linear w.r.t. input G and output d. Here, each iteration makes multiple assignments to d and is consequently piecewise K-linear for $K \geq 2$. As the main loop iterates N times, the complete algorithm is then piecewise $O(2^N)$-robust. However, by the reasoning in Ex. 3, Dijkstra's algorithm is piecewise N-linear. Now we present an abstraction that can establish this and similar facts. A key insight here is to treat the locations of the input variable x_{in} as *resources*, and to establish an assertion of the form "Each location of x_{in} is used at most once during the execution of the loop."

To express our conservative abstractions, we extend the language Imp with a syntactic form for loops with restricted nondeterministic choice. We call this extended language Limp. Its syntax is as in Figure 6. Here:

- U is a set—the *iteration space* for the loop in the syntactic form Q. Its elements are called *choices*.
- x_{in} is an unbounded data structure (e.g. an array).
- θ is a special variable, called the *current choice variable*. Every iteration starts by picking an element of U and storing it in θ.
- x_{out} is the output variable.
- $R(U, \theta, x_{in}[\theta], x_{out})$ (henceforth R) is an Imp program that does not write to θ, or use elements of x_{in} other than $x_{in}[\theta]$, but can read θ and read or update the iteration space U, element $x_{in}[\theta]$, and the output x_{out}.

$$\text{(LinLoop)}\ \frac{\begin{array}{c} UseOnce(\theta, U, Q) \\ PLin(R, k, x_{in}[\theta], x_{out}) \qquad PLin(R, 1, x_{out}, x_{out}) \end{array}}{PLin(Q, k.N, x_{in}, x_{out})}$$

Figure 7: Proof rule LinLoop (Q is an abstract loop)

We call a program of form Q an *abstract loop*—henceforth, Q denotes an arbitrary, fixed abstract loop. For simplicity, we only consider the analysis of abstract loops—an extension to all Limp programs involves a combination of the present abstraction with the previous abstractions. As before, we restrict ourselves to programs that terminate on all inputs.

The main loops in Figs. 1 and 2 are abstract loops. For example, the workset WL, the graph G, the node u, and the array d in Figure 1 respectively correspond to the iteration space U, the input variable x_{in}, the choice variable θ, and the output variable x_{out}.

Now we present a rule (Fig. 7) for proving piecewise N-linearity of Limp programs. As before, for brevity, we only derive robustness judgments that hold over the entire $\Sigma(P)$. We denote by $PLin(P, k.N, x_{in}, x_{out})$ the judgment "P is piecewise $k.N$-linear within $\Sigma(P)$ with respect to x_{in} and x_{out}."

A key premise for the rule is the assertion $UseOnce(\theta, U, Q)$, which states that the *values of θ chosen during a complete execution of Q are all distinct.* As the variable θ is used to index the data array x_{in}, we can also see this assertion to mean that each memory location in x_{in} is used at most once during a run of Q. We establish the property $UseOnce(\theta, U, Q)$ using a few syntactic sufficient conditions that apply to several common classes of programs. For example, if Q is a for-loop over the indices of x_{in}, and θ is the (monotonically increasing or decreasing) index variable, then we have $UseOnce(\theta, U, Q)$. For another example, consider Dijkstra's algorithm (Fig. 1), where an element w is removed from the workset WL in each iteration, never to be re-inserted again. In this case, letting Q_{Dijk} be the main loop (Lines 4-10) of the program, we have $UseOnce(w, WL, Q_{Dijk})$. Another example in this class is Kruskal's algorithm (Fig. 2), where we iterate over a workset that initially contains all edges of the input graph. Because edges are removed from but never added to this set, we have a use-once invariant similar to that in *Dijk*. Lastly, observe the premises $PLin(R, k, x_{in}[\theta], x_{out})$ and $PLin(R, 1, x_{out}, x_{out})$. These are typically derived using one of our other abstractions—e.g., robustness matrices.

To see the intuition behind the rule LinLoop, consider the simple case where x_{out} is a `real`. By the premises, the effect of each iteration on x_{out} can be summarized by assignments

$$x_{out} := c_1 \cdot x_{out} + c_2 \cdot x_{in}[\theta] + c_3$$

where c_1, c_2, and c_3 are constants with $|c_1| \leq 1$ and $|c_2| \leq k$. As each location in x_{in} is used only once and our norm over arrays is L_∞, this means the complete loop is piecewise $k.N$-linear with respect to output x_{out}.[2] We can show that:

Theorem 5 (Soundness). *If the rule* LinLoop *infers the judgment $PLin(Q, N, x_{in}, x_{out})$, then the abstract loop Q is piecewise N-linear within $\Sigma(Q)$ with respect to input x_{in} and output x_{out}.*

[2]Interestingly, if the metric for arrays were the L_1-norm rather than the L_∞-norm, then the rule LinLoop would be sound even if we changed its conclusion to $PLin(Q, k, x_{in}, x_{out})$.

```
CALC_TRANS_SLOW_TORQUES(angle : real, speed : real)
1  limit := 29; pressure1 := 0; pressure2 := 0;
2  if (angle ≥ 27∧ angle < 35) then limit :=41;
3  elseif (angle ≥ 35∧ angle < 50) then limit :=63;
4  elseif (angle ≥ 50∧ angle < 65) then limit :=109;
5  elseif (angle ≥ 65) then limit := 127;
6  if (3 * speed ≤ limit)
7      then    gear := 3; pressure1 := 1000;
8      else    gear := 4; pressure1 := 1000;
9  if (gear ≥ 3) then pressure2 := 1000
```

Figure 8: Code from a car transmission controller

Example 6 (Kruskal's algorithm). *For an application of the rule* LinLoop*, consider Kruskal's algorithm (2), whose main loop can be abstracted using an abstract loop. To analyze this loop, we establish the use-once property as discussed earlier. All that is left is to show that the loop body (Line 5-9) is piecewise 1-linear. This is easy to do using the robustness matrix abstraction. A similar strategy applies to several other examples, such as Dijkstra's or Bellman-Ford's shortest-path algorithms.*

4. APPLICATIONS

In this section, we identify three motivating application domains for our analysis. As this paper is primarily a foundational contribution, our discussions here use small, illustrative code fragments. The challenges of scaling to large real-world benchmarks is left for future work.

4.1 Robustness of embedded control software

Robustness is a critical system property for many embedded control systems. The sensor data that drives these systems is often prone to noise and errors, and unpredictable changes to system behavior due to this sort of uncertainty can have catastrophic consequences. A proof that the system reacts predictably to perturbations in its inputs is therefore of crucial practical importance. Unsurprisingly, control theorists have studied the problem of robust controller design thoroughly [28]. However, approaches to the problem in control theory are concerned with deriving abstractly defined laws for robust control, rather than proving the robustness of the software that ultimately implements them. This is a gap that a program analysis of robustness a la our paper can fill.

As an example of how to apply our analysis to this space, we consider the code fragment in Fig. 8, derived from a software implementation of a transmission shift control system [3]. Robustness of this fragment, under a different definition of robustness, was previously studied by Majumdar and Saha [16]. Given the car speed and the throttle angle, the operator `calc_trans_slow_torques` computes a pair of pressure values `pressure1` and `presure2`, which are applied to actuators related to the car transmission system. A careful analysis reveals that the output `pressure1` is constant, which means the function is 1-robust in that output. On the other hand, it is not continuous in the second output `pressure2`, hence is not K-robust for any K. Sec. 5 reports on the results of our implementation on this example.

4.2 Robustness in approximate computation

Another application for our analysis is in *approximate computation*, where the goal is to trade off the accuracy of a

```
1  while (H(i))
2     if (y ↦ P(y)), where y ∈ [x[i] − ϵ, x[i] + ϵ], is in table
3           then   t := look up P(y); G(y, z, t, i)
4           else   t := P(x[i]);
5                  tabulate (x[i] ↦ P(x[i])); G(x[i], z, t, i)
```

Figure 9: Approximate memoization for loops

computation for resource savings. Rather than approximate solutions for specific problems, language-based program approximation [20] involves program transformations that are just like traditional compiler optimizations but, when applied to a program, lead to an *approximately* equivalent program. Such approaches are especially applicable to domains like image and signal processing, where programs compute continuous values with negotiable accuracy.

While such approximation schemes have been much discussed of late, very little is understood at this time about their theoretical foundations. Now we show that a robustness analysis such as ours can provide foundations for one such scheme, called *loop perforation* [20, 19]. The loop perforation optimization is applicable to expensive computational loops over large datasets; what it does is simply skip every alternate iteration in certain long-running loops. The empirical observation in many cases, this bizarre and obviously unsound optimization does not significantly affect the accuracy of the final output. In [20], a profiling compiler is proposed that exploits this fact and identifies loop iterations that can be skipped.

As for the theoretical foundations of loop perforation, a recent paper [19] shows that if the loop under consideration follows certain computational patterns and the dataset on which the loop operates follows certain probability distributions, then loop perforation is "probabilistically sound"—i.e., the output of loop perforation is guaranteed to be within reasonable bounds with high probability. Now we show that such a guarantee is available for *any* loop that satisfies certain robustness requirements of the sort studied in this paper. This means that our analysis can be used to identify loops that can be correctly perforated, and can serve as a static guidance mechanism for a perforating compiler.

Approximate memoization. Before we show how robustness relates to loop perforation, let us offer a more general approximation scheme. Consider a loop Q of form

$$\textbf{while } (H(i)) \ \{\ t := P(x[i]); G(x[i], z, t, i)\}.$$

Here x is a large, read-only array that is the "input variable" for the loop (for simplicity, we let elements of x to be reals). The variable z is the "output variable," t is a temporary variable, H and P are side-effect-free computations, and $G(\dots)$ is an imperative procedure that can only read the element $x[i]$ out of x, but can have other effects. The function P is expensive, and we would like to eliminate calls to it.

Our first approximation of Q consists of a sort of *approximate memoization* of P. Suppose that, in a loop iteration, prior to making a call to $P(x[i])$, we find that P was previously evaluated on some $x[j]$ such that $x[j] \approx x[i]$. Then rather than evaluating $P(x[i])$, we simply use the (cached) value $P(x[j])$. The pseudocode for the optimized loop is given in Fig. 9.

Robustness of Q is needed for this approximation to be "sound." Denote by $Q(x)$ the final value of z on input x; also, suppose Q is K-robust with respect to the input x and output z. In that case, the scheme in Fig. 9 is equivalent to a transformation that replaces Q by a program Q' that, on any input x: (1) perturbs x by an amount $\delta \leq \epsilon$, resulting in an array x'; and (2) computes $Q(x')$.

By the robustness of Q, the outputs $Q(x)$ and $Q(x')$ on x and x' differ at most by $K \cdot \epsilon$. If this value is suitably small (and we can make it be, by selecting ϵ suitably), the optimization approximately preserves the semantics of Q. But if Q is non-robust, the outputs of the optimization may be very different from the idealized output.

Loop perforation. One obvious objection to the above scheme is that the complexity of table lookup will make it impractical. However, we now demonstrate that under some extra assumptions, this scheme reduces to a scheme that is, almost exactly, loop perforation. First, let us restrict ourselves to for-loops iterating over an array x—i.e., we assume Q to have the form

$$\textbf{for } 0 \leq i < N \textbf{ do } t := P(x[i]); G(x[i], z, t)$$

where $N = |x|$, and the imperative procedure G does not modify the induction variable i of the loop.

Second, let us assume that the input dataset x exhibits *locality*—i.e., any two successive elements in x are approximately equal with high probability. Note that this property holds in most multimedia datasets—for example, in most images neighboring pixels, for the most part, have similar colors. We formalize the above using a model defined in Misailovic et al [19] that views x as a random variable that is generated by a random walk with independent increments. In more detail, the variable $x[0]$ is a fixed constant, and $z_i = x[i+1] - x[i]$ follows a normal distribution $\mathcal{N}$ with mean μ and variance σ^2.

As successive elements in x are likely to be close in value, we can replace Q by a program $\widehat{Q}$ that uses $x[i]$ as a proxy for the value $x[i+1]$, and the cached value $P(x[i])$ as a proxy for $P(x[i+1])$, in the iteration $(i+1)$. Observe that we have now arrived at an approximation scheme that is quite like loop perforation!

There is, however, an important distinction between this scheme and the version of loop perforation presented in prior work [20, 19]. As the latter scheme skips loop iterations entirely, it is not applicable when the loop body performs discrete computations, pointer updates, etc., in addition to calling P (these computations are encapsulated within the routine G). Perforating these loops may lead not only to inaccurate results but to system crashes. Our approach, on the other hand, does not skip any iterations, but only the call to P inside the iterations, and can be viewed as executing Q on a perturbed input. Therefore, if the original program Q does not crash on any input, then neither does $\widehat{Q}$.

As for the analysis of error produced by the above transformation, let us define the *output error* for the transformation to be $Err_Q(x) = d(Q(x), \widehat{Q}(x))$. Now observe that $\widehat{Q}(x) = Q(x')$ where for all i, $(x[i] - x'[i]) \sim \mathcal{N}(\mu, \sigma^2)$. Because we use the L_∞ norm as the distance measure over arrays, we have, for every $a > 0$, the property

$$\begin{aligned}\mathbb{P}[d(x, x') > a] &= \mathbb{P}[\exists i : |x[i] - x'[i]| > a] \\ &\leq N \cdot \mathbb{P}[|x[i] - x'[i]| > a] \\ &\leq N \cdot (1 - (\Phi_{\mathcal{N}}(a) - \Phi_{\mathcal{N}}(-a)))\end{aligned}$$

where $\Phi_{\mathcal{N}}$ is the cumulative distribution function of $\mathcal{N}$.

```
InsideError(FGmap: BinaryImage)
 1  samples :=0; error := 0;
 2  for cam:= 1 to nCams
 3     for cylndr := 0 to nParts
 4           s1,s2 := vectors on sides of body part;
 5           m := vector connecting midpoints of s1 and s2;
 6           n1 := |m| / vStep; n2 := |m| / hStep;
 7           for i := 1 to n1
 8              δ1 = i/n1;
 9              p1 := cylndr[0] + (s1.x * δ1, s1.y * δ1);
10              p2 := cylndr[3] + (s2.x * δ1, s2.y * δ1);
11              m := p2 - p1;
12              for j := 1 to n2
13                    δ2 := j/n2;
14                    p3 := p1 + (m.x * δ2, m.y * δ2);
15                    error := error + (1-FGmap(p3));
16                    samples++;
17  return error/samples;
```

Figure 10: Bodytrack's `InsideError` function

Let $\chi = 1-(\Phi_{\mathcal{N}}(a)-\Phi_{\mathcal{N}}(-a))$. Note that for χ decreases as a increases, and is approximately 0.03 for $a = 3 \cdot \sigma$. By the K-robustness of Q, we have:

$$\mathbb{P}[Err_Q(x) > K \cdot a] \leq N \cdot \chi.$$

The above can be seen as a "probabilistic" soundness result for loop perforation: if the loop Q is robust with a low robustness parameter, then the probability of loop perforation introducing a significant error is low. Indeed, one can give an intuitive interpretation to perforation here: it amounts to sampling the dataset x at a lower frequency, which is acceptable if the process is robust. What all of this means is that one can use a robustness analysis such as ours to determine whether a given loop is suitable for perforation.

For a concrete example, consider the code fragment in Fig. 10, from the computer vision application called *Bodytrack* in the Parsec benchmarks [21]. (Perforation of this application was studied in [20].) The goal of this application is to track the major body components of a moving subject; the code in Fig. 10 performs a sampling computation inside a cylinder (a projected body part). In this code, the loops at lines 2, 3, 7, and 12 can be perforated with good results [20]. However, we observe that the reason behind this is that the sampling process performed between lines 4 to 17 is robust. This robustness property can be proved by our analysis.

4.3 Differential privacy

Robustness analysis can also be helpful in guaranteeing privacy in statistical databases, where a trusted party wants to disseminate aggregate data about a population while preserving the privacy of individual members of the population. The dominant notion of privacy in this setting is *differential privacy* [10], which asserts that the result of a statistical query should not be affected substantially by the presence or absence of a single individual. A known strategy to "privatize" statistical queries is to add some noise to the result; the amount of noise needed is related to how sensitive is the query to individual changes of the data set. Our notion of K-robustness can be used to establish the sensitivity of the query. Suppose we have a query `over_six_feet` (Fig. 11) which returns the number of individuals that are over six feet tall in a population. Let us represent the set of rows in the database as two arrays of the same size: `heights`, an array of reals and `rows`, an array of reals whose element values will range over [0,1] representing its presence or absence in the populations. The first array, `heights`, contains the height of the individual while the second, `rows` specifies whether a certain row is present or not in the set. The row is present in the database if its corresponding array element (in the first array representing the rows) is 1 and absent if it is 0; note that we allow all real values between 0 and 1.

To compute the amount of noise to be used for ϵ-differential privacy, one first needs to determine the robustness parameter of the query with respect to a suitable metric. In differential privacy, we are only interested in determining the robustness of `over_six_feet` with respect to the first parameter, `rows`. Note that for this case we will need to use the L_1-norm on the type `realarray`. With this norm we can prove that `over_six_feet` is 1-robust, as removing or adding an element to the set will imply a change to only one of the elements of the `rows` array by 1, making the L_1 norm of the difference to be 1. The rules needed for this proof can be obtained by a simple modification of the rules that we have presented (which assume the L_∞ norm). According to [10] random noise with variance $1/\epsilon$ will be needed so that the query `over_six_feet` is ϵ-differentially private. Thus, our analysis can guide the amount of noise that needs to be added to ensure the differential privacy guarantee.

```
over_six_feet(rows : realarray, heights : realarray)
1  result := 0;
2  for i := 0 to n
3     if (heights[i] > 6)
4           then    result := result + row[i];
```

Figure 11: A 1-robust query

5. EXPERIMENTS

We implemented our robustness analysis on top of the Z3 SMT-solver, and used the tool to verify the robustness of several classic algorithms from an undergraduate computer science textbook, as well as the code fragments in Section 4. Now we report on some experiments using this tool.

Sorting Algorithms. Our tool was able to verify the robustness of several classic sorting algorithms that take an array A_{in} and return the sorted array A_{out}. In particular, we considered *InsertionSort, BubbleSort, SelectionSort* and *MergeSort*. In [5] those four algorithms where proved continuous with respect to A_{in} and A_{out} using proof system Cont. This time we proved that the computation of A_{out} is arithmetic free, hence 1-robust due to continuity.

Shortest Path Algorithms. We verified the robustness of several shortest path algorithms. Recall that single source shortest path is N-robust with respect to the input array of edge weights. The proof for the particular shortest path algorithms such as *Dijk* and *Bellman-Ford*, consists of the following. In [5] we have proved these algorithms continuous on the input array. To prove piecewise N-robustness of the loop we prove that the loop body is piecewise 1-robust with respect to the array variable, and with respect to output variable. This is done using the rule RobMat on an abstraction of loop body. It follows from our method that

after executing the loop, the output variable d is N-linear with respect to the array of edge waits. Given that it is piecewise N-linear and continuous we can conclude that it is N-robust. It must be noted that the nested loop structure in *Dijk* can be abstracted as one single loop in LIMP. Our proof system is unable to prove N-robustness for some shortest path algorithms such as *Floyd-Warshall*.

Minimum Spanning Tree Algorithms. The minimun spanning tree problem is N-robust, as explained in Example 2. The proofs for the particular spanning tree algorithms, *Kruskall* and *Prim*, are similar to that of the shortest path algorithms. Continuity is proved using the CONT proof system of [5]. Piecewise N-linearity follows by expressing the loop in LIMP and proving piecewise 1-robustness of the loop body. Now we conclude that the aforementioned algorithms are N-robust. Again our proof system can not prove N-robustness for some spanning tree algorithms like *Boruvka's*.

Knapsack Algorithms. The integer-knapsack algorithm takes as input a weight array c and a value array v, and a knapsack capacity *Budget* and returns the set of items with maximum combined value tot_v such that their combined weight is less than the knapsack capacity. Clearly value of tot_v is N-robust in v. To prove N-robustness of our recursive *Knapsack* implementation we first prove the algorithm continuous using CONT [5]. The prover uses a fixpoint procedure to prove piecewise K-robustness assuming K-robustness for the recursive function calls, while probing for different values of K (0,1 and n). In order to prove piecewise N-linearity, the function was manually rewritten to make it explicit the array partitioning operation at each recursive call, where the input arrays are partitioned in two, one containing only the first element, and the other containing the rest. The tool keeps track of this partitioning to prove N-linearity for the addition operation in line 5. At each fixpoint iteration linearity of the function body is established using proof system ROBMAT.

Car Transmission Controller Example (Fig. 8). The algorithm produces two outputs in two variables, pressure1 and pressure2. The tool determines 1-robustness on pressure1, and non robust on pressure2. To arrive to that conclusion the tool uses the proof system ROBMAT in conjunction to the SMT solver to discharge proof obligations for each discontinuity. These proof obligations arise due to the proof rule ITE-CON presented in continuity analysis [5].

Loop perforation examples. Our proof rules are able to establish robustness proofs for several other code segments from the Parsec Benchmark Suite [21] where loop perforation is empirically successful [20].

In particular, our rules can prove the robustness of the loops for the ImageMeasurement class in the Bodytrack application (including the one shown in Fig. 10), where loop perforation is reported to work. A second example comes from `x264`, a media application that performs H.264 encoding on a video stream. Two outermost loops in function `pixel_satd_wxh` are claimed to be amenable to perforation with good results by [20]. We looked at the code snippet inside the body of the nested loops: similar to the example in bodytrack, perforating these loops will result in sampling less points in a coarser way, and the computation inside the body of the loop is a robust computation.

However, not all loops that are empirically amenable to loop perforation are provably robust by our analysis. Examples include the loop in the `pFL()` routine in the StreamCluster application, and a loop in the `ParticleFilter::Update` method in Bodytrack. We leave for future work a proof principle that can explain why these (and similar) loops can be successfully perforated.

Example	Time	# SMT proofs	Proof method
BubbleSort	0.250	1	continuity + arithmetic freedom
InsertionSort	0.098	1	continuity + arithmetic freedom
SelectionSort	0.293	3	continuity + arithmetic freedom
MergeSort	0.560	3	continuity + arithmetic freedom + array partitioning
Dijkstra	0.454	3	continuity + robustness matrix + linear loops
Bellman-Ford	0.316	1	continuity + robustness matrix + linear loops
Kruskal	1.198	1	continuity + robustness matrix + linear loops
Prim	0.547	3	continuity + robustness matrix + linear loops
Knapsack	1.480	3	continuity + robustness matrix + array partitioning
Controller	8.770	60	continuity + robustness matrix

Table 1: Benchmark Examples

Implementation and Experimental Setup. Our tool is implemented in C#, relying on the Z3 SMT solver to discharge proof obligations and the Phoenix Compiler Framework to process the input program. The bulk of the new analysis computes facts about linear dependences between variables and parameters and is implemented as a fixpoint computation that finds the solution of dataflow equations derived from the proof rules. Some proof obligations are discharged in the process by the SMT-solver. In the future we plan to use an optimization toolbox to have better guesses for the minimum bound. Some proofs involving arrays, e.g. *MergeSort*, requires to keep track of the accesses to the array, with purpose to ensure disjoint access to elements of the array. Finally, as mentioned earlier, we manually rewrote some of the programs to fit the abstraction language LIMP. The performance results reported in table 1 were obtained on a Core2 Duo 2.53 Ghz with 4GB of RAM.

6. RELATED WORK

Robustness is a standard correctness property in control theory [22, 23], and there is an entire subfield of control theory studying the design and analysis of robust (control) systems. However, the systems studied by this literature methods are abstractly defined using differential equations and hybrid automata, rather than programs. As far as we know, the only effort to generally define and study the robustness of embedded control *software* can be found in [16, 17] on test generation for robustness. There, robustness is formulated as: "If the input of the program P changes by an amount less than ϵ, where ϵ is a *fixed* constant, then the output changes by only slightly." In contrast, we verify the stronger property that *any* perturbation to the inputs will change the output proportionally. Many applications (e.g., differential privacy) demand this stronger formulation.

In addition, there are many efforts in the abstract interpretation literature that, while not verifying robustness explicitly, reason about the uncertainty in a program's behav-

ior due to floating-point rounding and sensor errors [11, 18, 8, 6, 7]. Several of these approaches have been successfully applied to large embedded code bases. However, none of them reason systematically about divergent control flow caused due to uncertainty, which we can thanks to our continuity analysis. Also, none of the abstractions developed in this space seem suitable for an analysis of piecewise robustness that is needed to verify robustness.

So far as we know, Hamlet [13] was the first to argue for a testing methodology for Lipschitz-continuity of software. However, he failed to offer new program analysis techniques. Reed and Pierce [25] have since given a type system that can verify the Lipschitz-continuity of functional programs, as a component of a new language for differential privacy [24]. While the system can seamlessly handle functional data structures such as lists and maps, it does not, unlike our analysis, handle control flow, and would deem any program containing a conditional branch to be nonrobust. Also, this work does not consider any application other than differential privacy.

Robustness and stability of numerical algorithms are also well-studied topics in the numerical analysis literature [14]. However, the proofs studied there are manual, and specialized to specific numerical algorithms. Other related literatures include that on automatic differentiation (AD) [4], where the goal is to transform a program P into a program that returns the derivative of P where it exists. But AD does not attempt verification—no attempt is made to certify a program as differentiable or Lipschitz.

Finally, language-based approaches to program approximation are still very new [20, 19, 26]. In particular, there is only one existing paper [19] studying the theory of language-based program approximation.

7. CONCLUSION

We have presented a program analysis to quantify the *robustness* of a program to uncertainty in its inputs. Our analysis is sound, and decomposes the verification of robustness into the independent subproblems of verifying continuity and piecewise robustness.

In future work, we intend to extend our robustness analysis to programs that manipulate discrete data types like integers and boolean arrays in addition to continuous ones (like the ones studied here). This extension is especially important as discrete types are often used in real-world applications, in particular embedded control code. Another interesting question is the generation of test inputs that trigger robustness bugs—i.e. pairs of inputs that are close in value, but on which the program behaves very differently. A third interesting direction is a notion of robustness for *reactive programs*, where we must consider perturbations not only in the program inputs, but also in the environment with which the program interacts.

8. REFERENCES

[1] L. Ambrosio and P. Tilli. *Topics on analysis in metric spaces*. Oxford University Press, 2004.

[2] L. Blum, M. Shub, and S. Smale. On a theory of computation over the real numbers; np completeness, recursive functions and universal machines (extended abstract). In *FOCS*, 1988.

[3] D. Bostic, W. P. Milam, Y. Wang, and J. A. Cook. Smart vehicle baseline report. http://vehicle.berkeley.edu/mobies, 2001.

[4] M. Bucker, G. Corliss, P. Hovland, U. Naumann, and B. Norris. *Automatic differentiation: applications, theory and implementations*. Birkhauser, 2006.

[5] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, pages 57–70, 2010.

[6] L. Chen, A. Miné, J. Wang, and P. Cousot. Interval polyhedra: An abstract domain to infer interval linear relationships. In *SAS*, pages 309–325, 2009.

[7] L. Chen, A. Miné, J. Wang, and P. Cousot. An abstract domain to discover interval linear equalities. In *VMCAI*, pages 112–128, 2010.

[8] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *ESOP*, pages 21–30, 2005.

[9] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[10] C. Dwork. Differential privacy. In *ICALP*, 2006.

[11] E. Goubault. Static analyses of the precision of floating-point operations. In *SAS*, pages 234–259, 2001.

[12] J. Halpern. *Reasoning about uncertainty*. The MIT Press, 2003.

[13] D. Hamlet. Continuity in sofware systems. In *ISSTA*, 2002.

[14] F. Iavernaro, F. Mazzia, and D. Trigiante. Stability and conditioning in numerical analysis. *Journal of Num. Analysis, Industrial and Applied Math.*, 1(1):91–112, 2006.

[15] E. Lee. Cyber physical systems: Design challenges. In *ISORC*, pages 363–369, 2008.

[16] R. Majumdar and I. Saha. Symbolic robustness analysis. *Real-Time Systems Symposium, IEEE International*, 0:355–363, 2009.

[17] R. Majumdar, I. Saha, and Zilong Wang. Systematic testing for control applications. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*, pages 1 –10, 2010.

[18] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, pages 3–17, 2004.

[19] S. Misailovic, D. M. Roy, and M. Rinard. Probabilistically accurate program transformations. In *SAS*, 2011.

[20] S. Misailovic, S. Sidiroglou, H. Hoffman, and M. Rinard. Quality of service profiling. In *ICSE*, 2010.

[21] The Parsec benchmark suite. http://parsec.cs.princeton.edu, 2009.

[22] S. Pettersson and B. Lennartson. Stability and robustness for hybrid systems. In *Decision and Control*, pages 1202–1207, Dec 1996.

[23] Andreas Podelski and Silke Wagner. Model checking of hybrid systems: From reachability towards stability. In *HSCC*, pages 507–521, 2006.

[24] J. Reed, A. Aviv, D. Wagner, A. Haeberlen, B. Pierce,

and J. Smith. Differential privacy for collaborative security. In *Eurosec*, 2010.

[25] J. Reed and B. Pierce. Distance makes the types grow stronger: A calculus for differential privacy. In *ICFP*, 2010.

[26] S. Sidiroglou, S. Misailovic, H. Hoffman, and M. Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *FSE*, 2011.

[27] G. Winskel. *The formal semantics of programming languages*. The MIT Press, 1993.

[28] G. Zames. Input-output feedback stability and robustness, 1959-85. *IEEE Control Systems*, 16(3), 1996.