Proving Programs Robust

Swarat Chaudhuri  
Pennsylvania State University  
swarat@cse.psu.edu

Sumit Gulwani  
Microsoft Research  
sumitg@microsoft.com

Roberto Lublinderman  
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 Kε, where K can depend on the size of the input, but is independent of 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.

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 computation is increasingly intertwined with sensor-derived perceptions of the physical world [15], and applications suited to approximation and randomization are ascendant. Love or hate uncertainty, you can increasingly ill afford to 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 safely approximated 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 safe from it; if we want to introduce uncertainty in our computation for performance gains, robustness ensures that it is safe to do so.

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. 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 ±ε to its inputs, for any ε, results in a change of ±Kε 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 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. Edge-weights in the input G are prone to changes, and the output of the program is the table d of shortest-path distances.
The effect of each loop iteration on the array \( Dijk \) establishes the property for the program bounded by size of the input, and the magnitude of the slope of this line is \( G \) of \( m \) encodes a continuous function, and, second, each control flow path whose inputs and outputs are from dense (e.g., real or real arrays) can be written as \( a \in \mathbb{R} \). An \( e \)-change to a value \( x \) of type \( \tau \) and size \( N \) is assumed to result in a value \( y \) of type \( \tau \) and size \( N \), such that \( d_{\tau,N}(x, y) = e \).

In particular, the type \( \text{real} \) (size is immaterial here, as all reals have size 1) is associated with the Euclidean metric, defined as \( d_{\text{real}}(x, y) = |x - y| \). The type \( \text{int} \) has a similar metric. We let an \( e \)-change to an array consist of \( e \)-changes to each of its elements. Formally, the metric over arrays of length \( N \) whose elements are of type \( \tau \) is the \( L_\infty \)-norm:

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

Now we offer the syntax of arithmetic expressions \( e \), boolean expressions \( b \), and programs \( P \) in IMP:

\[
e  ::=  x \mid c \mid e_1 + e_2 \mid e_1 - e_2 \mid A[i]
\]

\[
b  ::=  e \geq 0 \mid b_1 \land b_2 \mid \neg b
\]

\[
P  ::=  \text{skip} \mid x := e \mid A[i] := e \mid \text{if } b \text{ then } P_1 \text{ else } P_2
\]

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; also, expressions and assignments in the program are assumed to be well-typed.

For semantics, let us consider programs \( P \) that terminate on all inputs. The semantics of \( P \) is the standard denotational semantics [25] for imperative programs (except as mentioned earlier, we assume unit-time operations on reals). Formally, let us associate with each variable \( x \) a set \( \text{Cloc}(x) \) of concrete memory locations. A state of \( P \) is a map \( \sigma \) that assigns a value in \( \text{Val}_\tau \) to each program variable \( x \) of type \( \tau \). 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 \( y \in \text{Cloc}(x) \), for each variable \( x \). We use the notation \( \sigma(y) \) to denote the content of location \( y \) at state \( \sigma \).

The semantics of the program \( P \), and an expression \( e \) of type \( \tau \) appearing in it, are now defined by two functions \([P] : \Sigma(P) \rightarrow \Sigma(P)\) and \([e] : \Sigma(P) \rightarrow \text{Val}_\tau\), where \( \text{Val}_\tau \) is the set of values of type \( \tau \). Intuitively, \([e]\)(\(\sigma\)) is the value of \( e \) at state \( \sigma \), and \([P](\sigma) \) is the state at which \( P \) terminates after starting execution from \( \sigma \). 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.

\[1\] Recall that a \textit{metric} over a set \( S \) is a function \( d : S \times S \rightarrow \mathbb{R} \cup \{\infty\} \) 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) \( d(x, y) + d(y, z) \geq d(x, z) \).

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 to only use four data types that illustrate the challenges of the analysis: the best: reals (\( \text{real} \)), integers (\( \text{int} \)), arrays of reals (\( \text{realarray} \)), and arrays of integers (\( \text{intarray} \)). Other popular types such as records, tuples, and functional lists/trees can be added without changing the analysis significantly.

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 [3]. 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 an integer or real, and \( N \) if it is an array of length \( N \). We assume, for each type \( \tau \) and size \( N \), a metric \( d_{\tau,N} \). An \( e \)-change to a value \( x \) of type \( \tau \) and size \( N \) is assumed to result in a value \( y \) of type \( \tau \) and size \( N \), such that \( d_{\tau,N}(x, y) = e \).

Robustness ensures that such uncertainty only causes proportional changes at most by \( \pm \epsilon \). We pursue a floating-point modeling of continuous data in future work.
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 $\epsilon$-change to the input of $P$ can only change the output of $P$ by $\pm K \epsilon$. Note that $\epsilon$ 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.

Second, 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 $N$ before $P$ starts executing, and an $\epsilon$-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 $\Sigma'$ of the space of input states, not making assertions about the effect of perturbations on states outside $\Sigma'$. This captures the fact that many realistic programs are robust only within certain regions of their input space.

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

$$\text{Perm}_{\epsilon,x_{in}}(\sigma, \sigma') \Rightarrow [P]\sigma \approx_{m,x_{out}} [P]\sigma'. $$

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

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 $\sigma' \in \Sigma'$,

$$\text{Perm}_{\delta,x_{in},x_{out}}(\sigma, \sigma') \Rightarrow [P]\sigma \approx_{x_{out}} [P]\sigma'. $$

If $P$ is continuous by the above, then infinitesimal perturbations to $x_{in}$ (that keep the state within the set $\Sigma'$) 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—there is no bound on the change to $x^2$ on an $\epsilon$-change to $x$. On the other hand, one can verify that if $x \in [0, 1]$, then the above program is 1-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, minimum spanning trees). Let $SP$ be a correct implementation of a shortest-path algorithm (e.g., Dijkstra; 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 $\epsilon$-change to $G$ thus amounts to a maximum 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 $s$ to the $i$-th node $v_i$. Of $G$. A second output is the array $\pi$ whose $i$-th element is a sequence of nodes forming a minimum-weight path between $s$ and $v_i$. Let the distance between two elements of $\pi$ be 0 if they are identical, and $\infty$ 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 $\epsilon$, a shortest path weight can change at most by $(N \epsilon)$. However, an $\epsilon$-change to $G$ may add or subtract elements from $\pi$—i.e., perturb $\pi$ by the amount $\infty$. Therefore, $SP$ is not $K$-robust with respect to the output $\pi$ for any $K$.

Similar arguments apply to a program $MST$ computing minimum spanning trees in a graph $G$ (Kruskal’s minimum spanning tree 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 $\Sigma' \subseteq \Sigma(P)$ 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 $\Sigma'$ 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.

For example, the function $\text{abs}(x) = |x|$, where $x \in \mathbb{R}$, is continuous as well as piecewise 1-robust—hence it is 1-robust. On the other hand, the continuous function “if ($x > 0$) then $x^2$ 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 $P$. 

---

**Figure 2.** Kruskal’s algorithm

1. for each node $v$ in $G$ do $C[v] := \{v\}$;
2. $WL := \text{set of all edges in } G$; $\text{cost} := 0$; $T := \emptyset$;
3. while $WL \neq \emptyset$ do
4. choose edge $(v, w) \in WL$ such that $G(v, w)$ is minimal;
5. remove $(v, w)$ from $WL$;
6. if $\exists C[v'] \neq C[w']$ then
7. add edge $(v, w)$ to $T$;
8. $\text{cost} := \text{cost} + G(v, w)$;
9. $C[v] := C[w'] := C[v] \cup C[w']$;
For any program $P$ and any set of states $\Sigma'$ of $P$, let $\Sigma'_{(i)}$ denote the set of states $\sigma \in \Sigma'$ such that starting from $\sigma$, $P$ 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 $\mathbb{N} \rightarrow \mathbb{R}$, and $x_{in}, x_{out} \in \text{Var}(P)$. $P$ 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 $\mathbb{N} \rightarrow \mathbb{R}$, and $x_{in}, x_{out} \in \text{Var}(P)$. $P$ is K-linear within $\Sigma'$ w.r.t. input $x_{in}$ and output $x_{out}$ if for each $z \in \text{Clo}(x_{in}), y \in \text{Clo}(x_{out}), \sigma \in \Sigma'$, we have the relationship

$$([P](\sigma))(y) = \left( \sum_{z \in \text{Clo}(x_{in})} c_{x_{in},y} z \right) + \tau,$$

where $\tau$ is an expression whose free variables range over the set $\bigcup_{z \neq x_{in}} \text{Clo}(z)$, and $\sum_{z \neq x_{in}} |c_{x_{in},y}| \leq K(\text{Size}(x_{in},\sigma))$.

$P$ 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; let $G$ (viewed as an array of real weights) be the input variable and $d$ the output. 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 $\pi$ 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 $\pi$ 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 input $G$ and output $d$.

Now we apply the notion of piecewise robustness and piecewise linearity in the analysis of robustness. The analysis differs slightly depending on whether the input $x_{in}$ and the output $x_{out}$ belong to dense and discrete data types. Let us define:

**Definition 5** (Dense/discrete types). A type $\tau$ is dense if for all values $x, y$ of type $\tau$ such that $x \neq y$ and $x$ and $y$ are of the same size, there exists $z$ of type $\tau$ such that $z \neq x, z \neq y$, $d(x, z) < d(x, y)$, and $d(y, z) < d(x, y)$. A type $\tau$ that is not dense is discrete.

In particular, real and realarr are dense types, while int and inat are discrete. The two most interesting scenarios are when the input variable $x_{in}$ and the output variable $x_{out}$ are both of dense types or both of discrete types—we restrict ourselves to these.

**Insertion-Sort**($A$ : realarr)

1. for $i := 1$ to $(|A| - 1)$
2. do $z := A[i]; j := i - 1$;
3. while $j \geq 0$ and $A[j] > z$
5. $A[j + 1] := z$

**Figure 3.** Insertion sort

### 3.2 Robustness in dense domains

We begin by considering the case when $x_{in}$ and $x_{out}$ are both dense types. Using the reasoning outlined in Sec. 3.1, we have:

**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 $\Sigma'$ with respect to input $x_{in}$ and output $x_{out}$ if and only if: (1) $P$ is continuous within $\Sigma'$ w.r.t. input $x_{in}$ and output $x_{out}$; and (2) $P$ is piecewise K-robust within $\Sigma'$ 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 (defined in the usual way) from $x_{in}$ to $x_{out}$ are free of arithmetic operations. A program can be shown to be arithmetic-free in this sense using standard slicing techniques—essentially, we can use a program like $Q$ above as an abstraction of $P$. Applications in the verification of piecewise robustness stem 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 $\Sigma'$ 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:

1. if $(x + y > 0)$ then $z := x + y$ else $z := -x - y$
To understand the interpretation of this matrix, we recall the classic definition of a Jacobian from vector calculus. The Jacobian of a function \( f : \mathbb{R}^n \to \mathbb{R}^m \) with inputs \( x_1, \ldots, x_n \in \mathbb{R} \) and outputs \( x_1', \ldots, x_m' \in \mathbb{R} \) is the matrix

\[
J_f = \begin{pmatrix}
\frac{\partial x_1}{\partial x_1} & \cdots & \frac{\partial x_1}{\partial x_n} \\
\vdots & \ddots & \vdots \\
\frac{\partial x_m}{\partial x_1} & \cdots & \frac{\partial x_m}{\partial x_n}
\end{pmatrix}
\]

Now, if \( f \) is a differentiable function, then for each \( x_i \) and \( x_i' \), it is \( K \)-robust with respect to input \( x_i \) and output \( x_i' \), where \( K \) is any upper bound on \( \| \frac{\partial f}{\partial x_i} \|_\infty \). 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 \( \frac{\partial x_i}{\partial x_i'} \)). 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 \( \nabla \) and \( \nabla' \); 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 if we use 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 \( \text{CONT} \).

The goal of our analysis—call it ROB\MAT—is to syntactically derive facts of the form \( P \vdash \nabla \), read as: \( \nabla \) is the robustness matrix for the subprogram \( P' \). The structural rules for the analysis are shown in Fig. 4. Here, \( I \) is the identity matrix. The assertion \( \text{Bound}(P', M) \) states that the while-loop \( P' \) terminates in \( M \) or less steps—this condition can be established either via an auxiliary checker or by manual annotation (here \( M \) is a constant; in settings involving arrays, \( M \) can be a function of the size of the input).

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 generalized into the rule for while-loops. We have:

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

Example 5. Fig. 5 shows the result of applying a dataflow analysis based on ROB\MAT to a simple program. The annotations depict the robustness matrices \( \nabla \) propagated to the various program points—we use the more readable notation \( \nabla_{y,x} \) to refer to the matrix entry \( \nabla_{ij} \) 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 \( x_i \), as resources, and to establish an assertion
Lastly, observe the premises \(PL\text{in}(R, 1, x_{in}[\theta], x_{out})\) and \(PL\text{in}(R, 1, x_{out}, x_{out})\). These are typically derived using one of our other abstractions—e.g., robustness matrices.

To see the intuition behind the rule \(\text{LinLoop}\), 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 \(|c_2| \leq 1\). As each location in \(x_{in}\) is used only once and our norm over arrays is \(L_\infty\), this means the complete loop is piecewise \(N\)-linear with respect to output \(x_{out}\).

We can show that:

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

Example 6 (Kruskal’s algorithm). For an application of the rule \(\text{LinLoop}\), 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 challenging examples, including Dijkstra’s and Bellman-Ford’s shortest-path algorithms.

### 3.3 Robustness in discrete domains

Now we consider robustness analysis with respect to input and output variables of discrete types. For example, consider Dijkstra’s algorithm once again, except this time, let its input \(G\) be a graph with integer-valued weights.

The problem with this setting is that Theorem 2 no longer applies. If \(x_{in}\) has a discrete type, then every nonzero change to \(x_{in}\) is non-infinitesimal, so that the continuity requirement “Infinitesimal changes to \(x_{in}\) cause infinitesimal changes to \(x_{out}\)” is vacuously true. Fortunately, the theorem and the analysis can be adapted to this setting with a small modification. Let us define:

**Definition 6** (Discrete continuity). The program \(P\) is discretely \(K(N)\)-continuous within \(\Sigma' \subseteq \Sigma\) with respect to the discrete-typed input \(x_{in}\) and the discrete-typed output \(x_{out}\) if for all unit changes to \(\sigma, \sigma' \in \Sigma'\),

\[
\text{Pert}_{x_{in}}(\sigma, \sigma') \implies [P](\sigma) \approx_{m, x_{in}} [P](\sigma')
\]

where \(m = K(\text{Size}(x_{in}, \sigma))\).

Now we can show that:

**Theorem 6.** Let \(P\) be a program, \(\Sigma' \subseteq \Sigma(P)\), and \(x_{in}\) and \(x_{out}\) be variables of discrete types. \(P\) is \(K\)-robust within \(\Sigma'\) with respect to \(x_{in}\) and output \(x_{out}\) if: (1) \(P\) is discretely \(K\)-continuous within \(\Sigma'\) with respect to input \(x_{in}\) and output \(x_{out}\); and (2) \(P\) is piecewise \(K\)-linear within \(\Sigma'\) with respect to input \(x_{in}\) and output \(x_{out}\).

Our analysis proves robustness by proving conditions (1) and (2) in Theorem 6. To prove piecewise linearity, we “cast” all integers in \(P\) to reals and all integer arrays to real arrays, then apply the method of Sec. 3.2. It is easy to see that this approach is sound.

To verify discrete continuity, we modify the system \(\text{CONT}\) [5]. The changes turn out to be fairly simple—here we only describe the most interesting of the needed changes. For simplicity, we assume that \(K\) is a constant; also, we only consider unconditional judgments where \(\Sigma' = \Sigma(P)\).

---

2Interestingly, if the metric for arrays were the \(L_1\)-norm rather than the \(L_\infty\)-norm, then the rule \(\text{LinLoop}\) would sound even if we changed its conclusion to \(PL\text{in}(Q, 1, x_{in}, x_{out})\).
Zeroes boolean expressions—e.g., if \( b \) the formula \( x \) in states that agree on the value of \( \sigma \) \( K \) \( P \) \( x \) can be represented symbolically. Finally, the assertion \( \Sigma^1 \vdash P_1 \equiv_{x_{out}} P_2 \) says that if \( P_1 \) and \( P_2 \) start executing from the same initial state \( \sigma \in \Sigma \), then they terminate in states that agree on the value of \( x_{out} \).

Intuitively, the rule ITE-CONT says that “if \( b \) then \( P_1 \) else \( P_2 \)” is continuous provided \( P_1 \) and \( P_2 \) are individually continuous, and that if \( P_1 \) and \( P_2 \) start executing from a state that can be affected “boundary” of \( b \), then the final value of \( x_{out} \) is the same no matter which branch is executed.

Let us now adapt this rule to the setting of discrete continuity. Let the judgment \( DCon(P_1, K, x_{in}, x_{out}) \) state that “\( P \) is discretely \( K \)-continuous w.r.t. input \( x_{in} \) and output \( x_{out} \).” For any boolean condition \( b \), \( Zeros_{\Sigma \sigma}(b) \) denotes the set of states where the truth value of \( b \) change on an infinitesimal perturbation to \( x_{in} \). A key observation is that the set \( Zeros_{\Sigma \sigma}(b) \) can be represented symbolically for many interesting boolean expressions—e.g., if \( b \) is the expression \( x_{in} \geq y + 1 \), then the formula \( x_{in} = y + 1 \) is symbolically represents \( Zeros_{\Sigma \sigma}(b) \). Finally, an assertion \( \Sigma^1 \vdash P_1 \equiv_{x_{out}} P_2 \) says that if \( P_1 \) and \( P_2 \) start executing from the same initial state \( \sigma \in \Sigma \), then they terminate in states that agree on the value of \( x_{out} \).

Figure 8. Continuity vs. discrete continuity

Of the rules used by the continuity analysis in [5], the rule ITE-CONT in Fig. 8 is one of the most interesting. Here, \( Cont(P, x_{in}, x_{out}) \) denotes the judgment “\( P \) is continuous (at all states) w.r.t. input \( x_{in} \) and output \( x_{out} \).” For any boolean condition \( b \), \( Zeros_{\Sigma \sigma}(b) \) denotes the set of states where the truth value of \( b \) change on an infinitesimal perturbation to \( x_{in} \). A key observation is that the set \( Zeros_{\Sigma \sigma}(b) \) can be represented symbolically for many interesting boolean expressions—e.g., if \( b \) is the expression \( x_{in} \geq y + 1 \), then the formula \( x_{in} = y + 1 \) is symbolically represents \( Zeros_{\Sigma \sigma}(b) \). Finally, an assertion \( \Sigma^1 \vdash P_1 \equiv_{x_{out}} P_2 \) says that if \( P_1 \) and \( P_2 \) start executing from the same initial state \( \sigma \in \Sigma \), then they terminate in states that agree on the value of \( x_{out} \).

Intuitively, the rule ITE-CONT says that “if \( b \) then \( P_1 \) else \( P_2 \)” is continuous provided \( P_1 \) and \( P_2 \) are individually continuous, and that if \( P_1 \) and \( P_2 \) start executing from a state that can be affected “boundary” of \( b \), then the final value of \( x_{out} \) is the same no matter which branch is executed.

Let us now adapt this rule to the setting of discrete continuity. Let the judgment \( DCon(P_1, K, x_{in}, x_{out}) \) state that “\( P \) is discretely \( K \)-continuous w.r.t. input \( x_{in} \) and output \( x_{out} \).” Second, let \( d(Zeros_{\Sigma \sigma}(b)) \leq 1 \) denote the set of states \( \sigma \) such that by perturbing the value of \( x_{in} \) in \( \sigma \) by \( \pm 1 \), we obtain a state in \( Zeros_{\Sigma \sigma}(b) \). For many interesting classes of expressions, this set can be represented symbolically. Finally, the assertion \( \Sigma^1 \vdash P_1 \equiv_{x_{out}} P_2 \) means that if \( P_1 \) and \( P_2 \) start executing from the same initial state \( \sigma \in \Sigma \), then they terminate in states that agree on the value of \( x_{out} \) with an error of \( \pm K \).

The modified rule ITE-DIS-CON is shown in Fig. 8. It is not hard to see that this rule is sound. Empirically speaking, the formulas that this rule dispatches during automatic verification are not too much more complex than those arising in continuity analysis. While the generation of these formulas raises some interesting technical issues, we omit further discussion for lack of space.

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 [26]. 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.

Figure 9. Code from a car transmission controller

As an example of how to apply our analysis to this space, we consider the code fragment in Fig. 9, derived from a software implementation of a transmission shift control system. (Robustness of this fragment, under a simpler 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 pressure2. These pressures are applied to some 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 1000-robust in output pressure2, which may or may not meet the robustness specification for the operator. Sec. 5 reports on the results of our implementation on this example.

4.2 Robustness in approximate computation

A second application for our analysis lies in approximate computation, where the goal is to trade off the accuracy of results computed by a program for performance or energy gains. Rather than approximate solutions for specific problems, language-based approaches to approximation involve generic program optimizations 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 rather than discrete values.

Now we show that a robustness analysis such as ours can provide foundations for a recently-proposed program approximation scheme, called loop perforation [19], for expensive computational loops over large datasets. The empirical observation in loop perforation is that in many such loops, it is possible to skip a large number of loop iterations without significantly affecting the accuracy of the final output. A profiling compiler is now proposed that can exploit this fact and identify loop iterations that can be skipped. While the compiler has been successfully applied to several applications from the PARSEC benchmark suite [20], it is fairly classified as black magic—no argument has been given as to why it “works.”

To see how robustness relates to loop perforation, let us start by considering a more general approximation scheme. Let \( Q \) be a program of the form “while \( \ldots \) do \( R \)” whose input is a large, read-only data set (represented as an array) \( x \), and where the element \( x[i] \) is the input to the loop body in the \( i \)-th iteration. Now suppose \( R \) includes a call to an expensive procedure \( P \). Our approximation of \( Q \) consists of a sort of approximate memoization of \( P \): if we find in the \( i \)-th iteration that \( P(x[j]) \) was previously evaluated for some \( j < i \), then rather than evaluating \( P \), we simply...
use the (cached) value $P(x[i])$. The pseudocode for the optimized loop
is given in Fig. 10. (Here, $P$ is assumed to be side-effect-free.)

Robustness of $Q$ (and $P$) is a critical requirement for this ap-
proximation to be “sound.” Suppose $Q$ is $K$-robust with respect
to the input $x$. In this case, the scheme in Fig. 10 is semanti-
cally 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) returns $Q(x')$. However, by the robustness of $Q$,
the outputs $Q(x)$ and $Q'(x')$ of $Q$ on $x$ and $x'$ differ at most by $K\epsilon$.
Thus, assuming $K$ has a “reasonable” value (or $\epsilon$ is suitably small),
the optimization approximately preserves the semantics of $Q$. On
the other hand, if $Q$ is non-robust, the approximated and original
versions of $Q$ may behave very differently on the same input.

It is possible to optimize the scheme further if the dataset $x$
equisemantically perturbed with high probability.
(We note that this property holds in most multimedia datasets—
e.g., in most images, neighboring pixels, for the most part, have
similar colors.) In this case, applying the previous principle, we
can, in most cases, use the value $P(x[i])$ as a proxy for $P(x[i+1])$;
in other words, every alternate loop iteration can simply skip the
call to $P$. Observe that we have now arrived at an approximation
scheme that is very similar to loop perforation! Of course, to be
sure that we can in fact apply this heuristic, we need a proof of
robustness of $Q$. Enter our analysis.

But what about those few values of $i$ where $x[i]$ and $x[i+1]$ have
different significant values? Suppose there are $M$ such values of $i$
($M \ll \text{size}(x)$); also, for all such $i$, let us have $d(x[i], x[i+1]) < \epsilon$.
For the remaining $i$, let $d(x[i], x[i+1]) < \delta \approx 0$. Let us pick the
$L_1$-norm as the notion of distance between arrays—in many cases,
our static analysis can be adapted to this metric in a straightforward
way. If $x'$ is the perturbed version of $x$ implicitly created by the
approximation, we have $d(x, x') \approx M\epsilon + (N - M)\delta/2$. If $Q$
is $K$-robust with respect to this metric on arrays, then we have
$d(Q(x), Q(x')) < K M\epsilon + K(N - M)\delta/2$. This upper bound
may often be small enough to not make a difference in practice.

In our experiments, we looked at the loops in the Parsedc bench-
marks claimed to be amenable to perforation in [19]. All of them
were carrying out provably robust computations, and on each of
them, the “locality-optimized” variant of our scheme gave similar
results as loop perforation. As a specific example, consider the code
fragments in Fig. 11, obtained from the computer vision application
called Bodytrack in the PARSEC benchmarks [20]. The goal of
this application is to track the major body components (torso, arms,
etc.) of a moving subject; the code in Fig. 11 performs a sampling
computation inside a cylinder (projected body part). In this code,
the loops at lines 2, 3, 7, and 12 can be perforated with good re-
sults [19]. However, we observe that the reason behind this is that
the sampling process performed between lines 4 to 17 is robust.
In fact, it is possible to see that an intuitive interpretation to the
perforation heuristic in this case: it amounts to sampling at a lower
frequency (which is acceptable, but only if the process is robust).
It is worth pointing out one significant way in which our ap-
proach improves upon loop perforation. The latter, as acknowled-
edged in [19], is not applicable when skipping the iterations is not
acceptable according to the logic of the loop body (for example,
when the loop body performs discrete computations, pointer up-
dates, etc). Perforating these loops may lead to unacceptably in-
correct results at best and system crashes at worst. Our approach,
on the other hand, does not skip any iterations, but only the rob-
ust, expensive computations inside the iterations. A simple exam-
ple would be a loop that performs an expensive computation in each
iteration and assigns the result of the computation to a new, unin-
itialized element of an array. Applying loop perforation to this loop
will result in some elements of the array remaining uninitialized.
On the other hand, our scheme provides a systematic way to change
the body of the loop, such that instead of skipping an iteration alto-
gether, only a robust part of the iteration is skipped.

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. 12) which re-
turns the number of individuals that are over six feet tall in a popu-
lation. Let us represent the set of rows in the database as two arrays
of the same size: rows an array of integers whose element values
will range over $\{0,1\}$, and heights an array of reals. The first ar-
ray, rows marks whether a certain row is present or not in the set,
and the second, heights contains the height of the individual.

To compute the amount of noise to be used for $\epsilon$-differential
privacy, one first needs to determine the robustness parameter of
the query with respect to a suitable metric. In differential pri-
vacy, we are only interested in determining the robustness of
over six feet with respect to the first parameter. Note that for
this case we will need to use the $L_1$-norm on the type intarray.
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 $L_1$
value 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 $L_{\infty}$ norm).

According to [10] random noise with variance $1/\epsilon$ will be
needed so that the query over six feet is $\epsilon$-differentially pri-
vate. Thus, our analysis can guide the amount of noise that needs
to be added to ensure the differential privacy guarantee.

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.
OVER_SIX_FEET(rows : intarray, heights : realarray)
1 result := 0;
2 for i := 0 to n
3 do if (rows[i] = 1 ∧ heights[i] > 6)
4 then result := result + 1;

Figure 12. A 1-robust query

Sorting Algorithms. Our tool was able to verify the robustness of several classic sorting algorithms that take an array \( A_{in} \) as input and return the sorted array \( A_{out} \) as output. In particular, we considered InsertionSort, BubbleSort, SelectionSort and MergeSort. In [5] those four algorithms where proved continuous with respect to input \( A_{in} \) and output \( A_{out} \) using proof system CONT. This time we proved that the computation of \( A_{out} \) is arithmetic free. Due to continuity, this meant they are 1-robust as well.

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 Dijkstra 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 Dijkstra 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 minimum spanning tree problem is \( N \)-robust, as explained in Example 2. The proofs for the particular spanning tree algorithms, Kruskal 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 \( v \) containing the weight and value respectively of various objects, and a knapsack capacity \( Budget \) and returns the set of items with maximum combined value \( tot \), such that their combined weight is less than the knapsack capacity. Clearly value of \( tot \) 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. 9). The algorithm produces two outputs in two variables, pressure 1 and pressure 2. By looking at the control flow one can easily see that it is 1-robust on pressure 1 and 1000-robust on pressure 2. The prover determines 1-robustness on pressure1, and non robust on pressure2. To arrive to that conclusion the prover uses the proof system ROBMAT in conjunction to the SMT solver to discard proof obligations for each discontinuity. These proof obligations arise due to the proof rule ITE-DIS-CON presented in figure 8. The current version submits proofs for some relevant values of \( K \). Notice in table 1 that given the number of discontinuities in this algorithm, the prover submits 60 different proofs to the SMT solver. In the future we envision to use a nonlinear optimization package to obtain a better initial estimate for \( K \); and thus reducing the number of tentative proofs.

Approximate memoization examples. Our proof rules are able to establish robustness proofs for code segments from applications of Parsec Benchmark Suite [20] where there are loops that are claimed to be amenable to perforation by [19]. For example, the presented code snippet from Bodytrack, shown in Fig. 11, can be proved robust by our analysis.

We have looked at several other examples of Parsec Benchmark suite with loops that are amenable to perforate. Our general observation was that almost all the interesting examples with such loops are performing some kind of a sampling over a continuous dataset, for example an image, and are similar to the code in Fig. 11. Discussing robustness proofs for all these examples will not be possible due to space limitations, so we only consider one more example. x264 is a media application in Parsec suite which performs H.264 encoding on a video stream. Two outer most loops in function pixel-satd-wxh are claimed to be amenable to perforation with good results by [19]. 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. Therefore, we can reuse the result of computation from the previous loop iteration.

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.

<table>
<thead>
<tr>
<th>Example</th>
<th>Time</th>
<th># SMT proofs</th>
<th>Proof method</th>
</tr>
</thead>
<tbody>
<tr>
<td>BubbleSort</td>
<td>0.250</td>
<td>continuity + arithmetic freedom</td>
<td></td>
</tr>
<tr>
<td>InsertionSort</td>
<td>0.098</td>
<td>continuity + arithmetic freedom</td>
<td></td>
</tr>
<tr>
<td>SelectionSort</td>
<td>0.293</td>
<td>continuity + arithmetic freedom</td>
<td></td>
</tr>
<tr>
<td>MergeSort</td>
<td>0.560</td>
<td>continuity + arithmetic freedom + array partitioning</td>
<td></td>
</tr>
<tr>
<td>Dijkstra</td>
<td>0.354</td>
<td>continuity + robustness matrix + linear loops</td>
<td></td>
</tr>
<tr>
<td>Bellman-Ford</td>
<td>0.316</td>
<td>continuity + robustness matrix + linear loops</td>
<td></td>
</tr>
<tr>
<td>Kruskal</td>
<td>1.198</td>
<td>continuity + robustness matrix + linear loops</td>
<td></td>
</tr>
<tr>
<td>Prim</td>
<td>0.547</td>
<td>continuity + robustness matrix + linear loops</td>
<td></td>
</tr>
<tr>
<td>Knapsack</td>
<td>1.480</td>
<td>continuity + robustness matrix + array partitioning</td>
<td></td>
</tr>
<tr>
<td>Controller</td>
<td>0.790</td>
<td>discrete robustness</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Benchmark Examples
6. Related work

Robustness is a standard correctness property in control theory [21, 22], and there is an entire subfield of control theory—robust control [26]—specializing in 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. So far as we know, the only effort to generally define and study the robustness of embedded control software is Majumdar et al's work [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 \( \epsilon \), where \( \epsilon \) 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 behavior 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, as explained in Sec. 3, 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 [24] 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 [23]. 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. We emphasize that AD does not attempt verification—no attempt is made to certify a program as differentiable or Lipschitz.

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. We have presented a static analysis that solves the latter.

As for future work, we are currently exploring the applications of our analysis in proving the robustness of real-world embedded control code. Aside from sound analysis of robustness, a pressing problem here will be the problem of generating tests that trigger robustness bugs [17]. Also, many settings involving uncertainty demand a generalization of the robustness property. A controller that is otherwise robust might change mode—nonrobustly—now and then; however, such switching may be safe with respect to a higher-level specification. But what is the best language for such a specification that still allows verification? Third, a more ambitious problem than verification of robustness is synthesis for robustness. While synthesis of robust controllers has been studied in control theory as well as the hybrid systems community, how do we synthesize robust programs?

Finally, we believe that analytic reasoning about programs has a bright future in approximate computation [19, 2]. In the present paper, we only showed how robustness can explain an existing program approximation scheme. However, in ongoing work, we are developing new, systematic methods, based on robustness analysis, for exploring resource-accuracy tradeoffs in programs.

References
