Quantitative Input Usage Static Analysis

Denis Mazzucato[0000−0002−3613−2035], Marco Campion[0000−0002−1099−3494], and Caterina Urban[0000−0002−8127−9642]

INRIA & ENS | PSL,
{denis.mazzucato,marco.campion,caterina.urban}@inria.fr

Abstract. Programming errors in software applications may produce plausible yet erroneous results, without providing a clear indication of failure. This happens, for instance, when certain inputs have a disproportionate impact on the program result. To address this issue, we propose a novel quantitative static analysis for determining the impact of inputs on the program computations, parametrized in the definition of impact. This static analysis employs an underlying abstract backward analyzer and computes a sound over-approximation of the impact of program inputs, providing valuable insights into how the analyzed program handles them. We implement a proof-of-concept static analyzer to demonstrate potential applications.

1 Introduction

Disastrous outcomes may result from programming errors in safety-critical settings, especially when they do not result in software failures but instead produce a plausible yet erroneous outcome. Such bugs are hard to spot since they provide no indication that something went wrong. A potential source of such errors is when an input variable has disproportionate impact on the program computations compared to the developers' expectations. A notable example is the Reinhart and Rogoff article "Growth in a Time of Debt" [19], which was heavily cited to justify austerity measures around the world in the following years, and was later discovered to be flawed [12]. Notably, one of the several programming and methodological errors discovered in the article is the incorrect usage of the input value relative to Norway's economic growth in 1964, compromising the authors' conclusion. Hence, it is important to employ techniques that enhance the confidence in the usage of input variables.

In this direction, Barowy et al. [2] proposed a stochastic approach specific for spreadsheet applications. Such approach is able to estimate the impact of input cells. However, the lack of mathematical guarantees precludes the employment of such technique in safety-critical contexts. On the other hand, existing formal methods-based approached only target qualitative properties about input data usage, e.g., only addressing whether an input variable is used or not [22, 23].

In this work, we present a novel quantitative input usage framework to discriminate between input variables with different impact on the outcome of a program. Such knowledge could either certify intended behavior or reveal potential flaws, by matching the developers' intuition on the expected impact of their input with the actual result of the quantitative study. We characterize the impact of an input variable with a notion of dependency between variables and outputs. Compared to other quantitative notions of dependency, e.g., quantitative information flow [10, 11], there are some key differences as the information we measure or the granularity of input contributions. Our framework is parametric in the choice of impact definition to better fit several factors, such as the program structure, the environment, the expertise of the developer, and the intuition of the researcher.

We propose a sound static analysis leveraging a backward analyzer to compute an over-approximation of the program semantics. In particular, this last component takes as input sets of program outputs, called output buckets, and computes an over-approximation of the input states leading to these buckets. Then, the end-user chooses the impact definition that best fits their needs, and our analysis applies such definition on the result of the previous phase. This approach, parametrized on the impact definition, ensures a more targeted and customizable analysis. We demonstrate the potential applications of our approach, by evaluating an automatic proof-of-concept tool of our static analysis against a set of use cases.

Contributions We make the following contributions:

1. In Section 3, we develop a theoretical framework by abstract interpretation [9] to quantify the impact of input variables by considering two instances of impact: OUTCOMES and RANGE. Section 6 discusses the origins of our impact definitions in comparison with related metrics found in the literature.
2. In Section 4, we present our static analysis and a possible abstract implementation of the impact instances.
3. Finally, Section 5 evaluates our proof-of-concept against four use cases: a simplified program from the Reinhart and Rogoff article, a program extracted from the recent OpenAI keynote, one from termination analysis, and the example presented in the overview. More use cases can be found in our online supplementary material [16, Appendix B].

2 Overview

In this section, we present an overview of our quantitative analysis using the simple Program 1, referred to as L, which is a prototype of an aircraft landing alarm system. The goal of program L is to inform the pilot about the level of risk associated with the landing approach. It takes two input variables, denoted as `angle` and `speed`, for the aircraft-airstrip alignment angle and the aircraft speed, respectively. A value of 1 represents a good alignment while -4 a non-

Input preconditions:

$$\texttt{angle} \in \{-4, 1\}$$
$$\texttt{speed} \in \{1, 2, 3\}$$

```
landing_coeff = abs(angle) + speed
if landing_coeff < 2 then
  risk = 0
else if landing_coeff > 5 then
  risk = 3
else
  risk = floor(landing_coeff) - 2
```

Program 1: Aircraft landing alarm system.

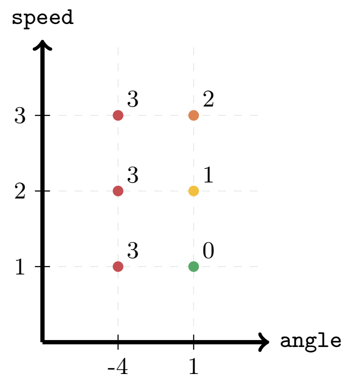

Fig. 1: Input space.

aligned angle, whereas 1, 2, 3 denote low, medium, and high speed[1]. A safer approach is indicated by lower speed. The landing risk coefficient combines the absolute landing angle and speed. The output variable `risk` is the danger level with possible values $\{0, 1, 2, 3\}$, where 0 represents low danger and 3 high danger. Figure 1 shows the input space composition of this system, where the label near each input represents the degree of risk assigned to the corresponding input configuration. It is easy to note that a nonaligned angle of approach corresponds to a considerably higher level of risk, whereas the risk with a correct angle depends mostly on the aircraft speed. Our goal is to develop a static analysis capable of quantifying the contribution of each input variable to the computation of the output variable `risk`.

Impact Analysis. We propose two impact definitions which, from value variations of the input variable under consideration, respectively focus on *the number of* resulting reachable outputs, and *the distance of* extreme reachable outputs.

The column $\textsc{Input}_{\text{L}}$ in Table 1 shows all the possible input configurations $\langle angle, speed\rangle$ for the program L. For each input configuration, column Relevant Traces groups together the program traces resulting from value variation of the input variable of interest (in column variable), and column Outputs collects the set of all reachable outputs.

First Impact Definition (Outcomes). The first impact definition that we consider is $\textsc{Outcomes}_i(\text{P})$, where i is the input variable of interest and P the program under analysis. Intuitively (the formal definition is given in Section 3), $\textsc{Outcomes}_i$ returns the maximum number of outputs that are reachable from value variations of the input variable i. For the program L, the result is shown in column Outcomes(L) of Table 1: we obtain $\textsc{Outcomes}_{\texttt{angle}}(\text{L}) = 2$ and $\textsc{Outcomes}_{\texttt{speed}}(\text{L}) = 3$. The conclusion is that `speed` has a greater influence

[1] We initially focus on discrete values to simplify the example and convey the concept. We expand to continuous inputs in Section 5.

Table 1: Impact of for OUTCOMES(L) and RANGE(L) definitions for both `angle` and `speed` variables. Computational features are highlighted in blue.

VARIABLE	INPUTL	RELEVANT TRACES	OUTPUTS	OUTCOMES	RANGE
`angle`	$\langle -4, 1\rangle$	$\langle -4, 1\rangle \to \langle 3\rangle, \langle 1, 1\rangle \to \langle 0\rangle$	$\{3, 0\}$	2	3
	$\langle -4, 2\rangle$	$\langle -4, 2\rangle \to \langle 3\rangle, \langle 1, 2\rangle \to \langle 1\rangle$	$\{3, 1\}$		
	$\langle -4, 3\rangle$	$\langle -4, 3\rangle \to \langle 3\rangle, \langle 1, 3\rangle \to \langle 2\rangle$	$\{3, 2\}$		
	$\langle 1, 1\rangle$	$\langle 1, 1\rangle \to \langle 0\rangle, \langle -4, 1\rangle \to \langle 3\rangle$	$\{0, 3\}$		
	$\langle 1, 2\rangle$	$\langle 1, 2\rangle \to \langle 1\rangle, \langle -4, 2\rangle \to \langle 3\rangle$	$\{1, 3\}$		
	$\langle 1, 3\rangle$	$\langle 1, 3\rangle \to \langle 2\rangle, \langle -4, 3\rangle \to \langle 3\rangle$	$\{2, 3\}$		
`speed`	$\langle -4, 1\rangle$	$\langle -4, 1\rangle \to \langle 3\rangle, \langle -4, 2\rangle \to \langle 3\rangle, \langle -4, 3\rangle \to \langle 3\rangle$	$\{3\}$	3	2
	$\langle -4, 2\rangle$	$\langle -4, 1\rangle \to \langle 3\rangle, \langle -4, 2\rangle \to \langle 3\rangle, \langle -4, 3\rangle \to \langle 3\rangle$	$\{3\}$		
	$\langle -4, 3\rangle$	$\langle -4, 1\rangle \to \langle 3\rangle, \langle -4, 2\rangle \to \langle 3\rangle, \langle -4, 3\rangle \to \langle 3\rangle$	$\{3\}$		
	$\langle 1, 1\rangle$	$\langle 1, 1\rangle \to \langle 0\rangle, \langle 1, 2\rangle \to \langle 1\rangle, \langle 1, 3\rangle \to \langle 2\rangle$	$\{0, 1, 2\}$		
	$\langle 1, 2\rangle$	$\langle 1, 1\rangle \to \langle 0\rangle, \langle 1, 2\rangle \to \langle 1\rangle, \langle 1, 3\rangle \to \langle 2\rangle$	$\{0, 1, 2\}$		
	$\langle 1, 3\rangle$	$\langle 1, 1\rangle \to \langle 0\rangle, \langle 1, 2\rangle \to \langle 1\rangle, \langle 1, 3\rangle \to \langle 2\rangle$	$\{0, 1, 2\}$		

than `angle` on the output of the program.

Second Impact Definition (RANGE). The second impact definition is RANGE_i, which yields the maximum difference between the maximum and the minimum outputs that are reachable from value variations of the input variable i. The result for program L is shown in column RANGE(L) of Table 1: the range of reachable outputs from variations of `angle` is, at most, the interval $[0, 3]$, with a length of 3. Instead, the range of reachable outputs from variations of `speed` is, at most, the interval $[0, 2]$, with a length of 2. Therefore, we obtain $\text{RANGE}_{\texttt{angle}}(\text{L}) = 3$ and $\text{RANGE}_{\texttt{speed}}(\text{L}) = 2$. In other words, varying the angle of approach might drastically alter the landing risk, whereas the speed has less influence. This is in contrast to the conclusion of OUTCOMES where `speed` has a greater impact than `angle`. Although it may seem counterintuitive at first, the difference between the two impact instances is due to the different program traits they explore. RANGE quantifies over the variance in the extreme values of the set of output values, while OUTCOMES quantifies over the variance in the number of unique output values. Consequently, changes in `angle` yield a bigger variation in the degree of risk compared to `speed`, while changes in `speed` reach far more risk levels compared to `angle`. Note that, the impact definitions presented above are not computationally practical as they rely on a complete enumeration of all possible input configurations. Specifically, when dealing with more complex input space compositions, this approach is highly inefficient or even infeasible (as in the case of continuous input spaces). As a consequence, our approach is based on an abstraction of input-output relations, which allows us to automatically infer a sound upper bound on the program's impact.

Abstract Analysis. The analysis starts with a set of output abstractions called *output buckets*. A bucket is an abstract element representing a set of output states. While this abstraction may limit the ability to precisely reason about the impact of output values within the same bucket, it permits automatic reasoning across different buckets. Afterwards, an abstract interpretation-based static analyzer propagates each output bucket backward through the program under consideration. The analyzer returns an abstract element for each output bucket, representing an over-approximation of the set of input configurations that lead to the output values inside the starting bucket. This result contains also spurious input configurations that may not lead to a value inside the output bucket. Based on the chosen impact definition IMPACT (e.g., RANGE or OUTCOMES), we perform computations and comparisons on the abstract elements returned by the analysis to obtain an upper bound k'. This upper bound is sound by construction of the theoretical framework, meaning that if k is the real (concrete) impact quantity obtained by IMPACT, then $k \leq k'$. The precision of our analysis is mostly affected by the choice of output buckets and the approximation induced by the backward analysis (as outlined by the use cases shown in Section 5 and in the supplementary material [16, Appendix B]).

3 Quantitative Input Data Usage

In this section we present some preliminaries on program computations, then we introduce our quantitative framework with the formal definitions of RANGE and OUTCOMES.

Program Semantics. The *semantics* of a program is a mathematical characterization of its behavior for all possible input data. We model the operational semantics of a program as a *transition system* $\langle \Sigma, \tau \rangle$ where Σ is a (potentially infinite) set of program states and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes the feasible transitions between states [9, 8]. The set $\Omega \overset{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma.\ \langle s, s' \rangle \notin \tau\}$ represents the *final states* of the program.

Let $\Sigma^n \overset{\text{def}}{=} \{s_0 \dots s_{n-1} \mid \forall i < n.\ s_i \in \Sigma\}$ be the set of all sequences of exactly n program states. We write ϵ to denote the empty sequence, i.e., $\Sigma^0 \overset{\text{def}}{=} \{\epsilon\}$. We define $\Sigma^\star \overset{\text{def}}{=} \bigcup_{n \in \mathbb{N}} \Sigma^n$ as the set of all finite sequences, $\Sigma^+ \overset{\text{def}}{=} \Sigma^\star \setminus \Sigma^0$ as the set of all non-empty finite sequences, $\Sigma^\infty \overset{\text{def}}{=} \{s_0 \dots \mid \forall i \in \mathbb{N}.\ s_i \in \Sigma\}$ as the set of all infinite sequences, and $\Sigma^{+\infty} \overset{\text{def}}{=} \Sigma^+ \cup \Sigma^\infty$ as the set of all non-empty finite or infinite sequences. Additionally, let $\Sigma^\perp \overset{\text{def}}{=} \Sigma \cup \{\perp\}$. Given a sequence $\sigma \in \Sigma^{+\infty}$, we write $\sigma_0 \in \Sigma$ to denote the initial state of σ and $\sigma_\omega \in \Sigma^\perp$ to denote the final state of σ when $\sigma \in \Sigma^+$, otherwise $\sigma_\omega = \perp$ when $\sigma \in \Sigma^\infty$. To concatenate two sequences of states $\sigma, \sigma' \in \Sigma^{+\infty}$, we write $\sigma \cdot \sigma'$. It holds that $\sigma \cdot \epsilon = \epsilon \cdot \sigma = \sigma$

and $\sigma \cdot \sigma' = \sigma$ whenever $\sigma \in \Sigma^{\infty}$. To merge two sets of sequences $T \subseteq \Sigma^{+}$ and $T' \subseteq \Sigma^{+\infty}$, we write $T \ ; \ T' \overset{\text{def}}{=} \{\sigma \cdot s \cdot \sigma' \mid s \in \Sigma \wedge \sigma \cdot s \in T \wedge s \cdot \sigma' \in T'\}$ when a finite sequence in T terminates with the initial state of a sequence in T'.

In the rest of the paper, $\mathbb{I} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$ represents a set of numerical values. We write $\mathbb{I}^{\pm\infty}$ to denote $\mathbb{I}$ extended with the symbols $+\infty$ and $-\infty$. The set $\mathbb{I}_{\geq 0} \overset{\text{def}}{=} \{n \in \mathbb{I} \mid n \geq 0\}$ denotes non-negative numbers. Similarly, we can use other predicates, for instance, $\mathbb{I}_{\leq m} \overset{\text{def}}{=} \{n \in \mathbb{I} \mid n \leq m\}$ denotes the set of numbers below or equal $m \in \mathbb{I}$.

Given a transition system $\langle \Sigma, \tau \rangle$, a *trace* is a non-empty sequence of program states that respects the transition relation τ, i.e., for every pair of consecutive states $s, s' \in \Sigma$ in the trace, it holds that $\langle s, s' \rangle \in \tau$. The *trace semantics* $\Lambda \in \wp(\Sigma^{+\infty})$ generated by a transition system $\langle \Sigma, \tau \rangle$ is the union between all finite traces that are terminating in a final state in Ω, and all non-terminating infinite traces [8]:

$$\begin{aligned}\Lambda \overset{\text{def}}{=} & \bigcup_{n \in \mathbb{N}_{\geq 0}} \{s_0 \dots s_{n-1} \in \Sigma^n \mid \forall i < n-1.\ \langle s_i, s_{i+1} \rangle \in \tau \wedge s_{n-1} \in \Omega\} \\ & \cup \{s_0 \dots \in \Sigma^{\infty} \mid \forall i \in \mathbb{N}.\ \langle s_i, s_{i+1} \rangle \in \tau\}\end{aligned}$$

We write $\Lambda[\![\mathrm{P}]\!]$ to denote the trace semantics of a particular program P. The same applies for other semantics defined in the rest of paper.

The trace semantics fully describes the behavior of a program. However, reasoning about a particular property of a program is facilitated by the design of a semantics that abstracts away from irrelevant details about program executions. In our work, we focus on *extensional* properties, namely, properties based on the observation of input-output relations of $\Lambda[\![\mathrm{P}]\!]$. Therefore, we employ the dependency semantics $\Lambda^{\rightsquigarrow} \in \wp(\Sigma \times \Sigma^{\perp})$ [22] as an abstraction of the trace semantics removing intermediate steps, i.e., $\Lambda^{\rightsquigarrow} \overset{\text{def}}{=} \{\langle \sigma_0, \sigma_\omega \rangle \mid \sigma \in \Lambda[\![\mathrm{P}]\!]\}$. Starting from the dependency semantics, we define our property of interest – quantitative input data usage – and use abstract interpretation to systematically derive a semantics tailored to reason about this property.

Property. A *property* is specified by its extension, that is, the set of elements that manifest such a property [9]. We consider properties of programs, with dependency semantics in $\wp(\Sigma \times \Sigma^{\perp})$, which are sets of sets of dependencies in $\wp(\wp(\Sigma \times \Sigma^{\perp}))$. The strongest property of the dependency semantics $\Lambda^{\rightsquigarrow}$ is the standard *collecting semantics* $\Lambda^{c} \in \wp(\wp(\Sigma \times \Sigma^{\perp}))$, defined as $\Lambda^{c} \overset{\text{def}}{=} \{\ \Lambda^{\rightsquigarrow}\ \}$, which is satisfied only and exactly by $\Lambda^{\rightsquigarrow}$. Therefore, a program P satisfies a given property $F \in \wp(\wp(\Sigma \times \Sigma^{\perp}))$, written $\mathrm{P} \models F$, if and only if P belongs to F, or equivalently, its collecting semantics Λ^{c} is a subset of F, formally

$$\mathrm{P} \models F \ \Leftrightarrow\ \Lambda^{c}[\![\mathrm{P}]\!] \subseteq F$$

Our goal is to quantify the impact of a specific input variable on the computation of the program. To this end, we introduce the notion of impact, denoted by the function $\textsc{Impact}_{i} \in \wp(\Sigma \times \Sigma^{\perp}) \to \mathbb{I}^{+\infty}_{\geq 0}$, which maps program semantics to a non-negative domain of quantities, where i represents the input variable of interest in the program under analysis. We implicitly assume the use of an *output descriptor* $\phi \in \Sigma^{\perp} \to \mathbb{I}^{\pm\infty}$ to determine the desired output of a program by observations on program states[2]. The output descriptor ϕ is generic enough to cover plenty of use cases, providing the end-user the flexibility to choose the interpretation and meaning of program outputs.

Example 1. Consider the Program 1 for the landing alarm system with program states $\Sigma = \{\langle a, b, c, d\rangle \mid a \in \{-4, 1\} \wedge b \in \{1, 2, 3\} \wedge c \in \mathbb{N} \wedge d \in \{0, 1, 2, 3\}\}$, where a is the value of `angle`, b of `speed`, c of `landing_coeff`, and d of `risk`. Here, we abuse the notation and use Σ as set of tuples instead of a map between variables and values, the two views are equivalent. The output descriptor is instantiated with

$$\phi(x) \overset{\text{def}}{=} \begin{cases} d & \text{if } x = \langle a, b, c, d\rangle \\ +\infty & \text{otherwise} \end{cases}$$

In other words, we are interested in the value of `risk` for terminating traces.

Given an impact definition of interest, we define the *k-bounded impact property* $\mathscr{B}_{i}^{\leq k} \in \wp(\wp(\Sigma \times \Sigma^{\perp}))$ as the set of dependency semantics with impact with respect to the input variable i below the threshold $k \in \mathbb{I}_{\geq 0}^{+\infty}$. Formally,

$$\mathscr{B}_{i}^{\leq k} \overset{\text{def}}{=} \{\Lambda^{\rightsquigarrow} \in \wp(\Sigma \times \Sigma^{\perp}) \mid \textsc{Impact}_{i}(\Lambda^{\rightsquigarrow}) \leq k\}$$

We require $\textsc{Impact}_i$ to be monotonic, i.e., for any $S, S' \in \wp(\Sigma \times \Sigma^{\perp})$, it holds that:

$$S \subseteq S' \Rightarrow \textsc{Impact}_{i}(S) \leq \textsc{Impact}_{i}(S')$$

Intuitively, this ensures that an impact applied to an over-approximation of the program semantics can only produce a higher quantity, allowing for a sound k-bounded impact verification.

Next, we formalize the already introduced impact metrics Outcomes and Range. Given a program P and its variables $\mathbb{V}$, we assume program states are maps from variables to a numerical domain, i.e., $\Sigma = \mathbb{V} \to \mathbb{I}$. The set $\Delta \subseteq \mathbb{V}$ is the set of input variables. We write $\Sigma|_K = K \to \mathbb{I}$ for the program states reduced to the subset of variables $K \subseteq \mathbb{V}$. For instance $\Sigma|_{\Delta}$ is the set of states restricted to the input variables. The predicate $s =_K s'$ indicates that the two states $s, s' \in \Sigma^{\perp}|_K$, agree on the values of the variables in $K \subseteq \mathbb{V}$, or they are both $\perp$, formally

$$s =_K s' \Leftrightarrow (s \neq \perp \wedge s' \neq \perp \wedge \forall v \in K.\ s(v) = s'(v)) \vee (s = \perp \wedge s' = \perp)$$

Outcomes. Formally $\textsc{Outcomes}_{i} \in \wp(\Sigma \times \Sigma^{\perp}) \to \mathbb{N}^{+\infty}$ counts the number of different output values reachable by varying the input variable $i \in \Delta$. Intuitively, for any possible input configuration $s \in \Sigma|_{\Delta}$, we gather the set $S \in \wp(\Sigma \times \Sigma^{\perp})$

[2] The option of returning $\pm\infty$ from the output descriptor is to deal with infinite traces, which do not have a final state ($\sigma_{\omega} = \perp$ for any $\sigma \in \Sigma^{\infty}$).

of all input-output state dependencies with an input configuration that is a variation of s on the input variable i, i.e., $\{\langle s_0, s_\omega \rangle \in S \mid s_0 =_{\Delta \setminus \{i\}} s\}$. Then, $\textsc{Outcomes}_i$ is the maximal cardinality of the output values $\{\phi(s_\omega) \mid \langle s_0, s_\omega \rangle \in S \wedge s_0 =_{\Delta \setminus \{i\}} s\}$. Formally,

$$\textsc{Outcomes}_i(S) \overset{\text{def}}{=} \sup_{s \in \Sigma|_\Delta} \mid \{\phi(s_\omega) \mid \langle s_0, s_\omega \rangle \in S \wedge s_0 =_{\Delta \setminus \{i\}} s\} \mid \tag{1}$$

where $\mid \cdot \mid$ is the cardinality operator, and $\sup(X)$ is the supremum operator, i.e., the smallest q such that $q \geq x$ for all $x \in X$. From the definition above, it is easy to note that $\textsc{Outcomes}_i(S)$ is monotone in the amount of dependencies S. That is, the more dependencies in input, the higher the impact as only more dependencies can satisfy the condition of Eq. (1), cf. $s_0 =_{\Delta \setminus \{i\}} s$, and hence increase the number of outcomes.

$\textsc{Range}$. The quantity $\textsc{Range}_i \in \wp(\Sigma \times \Sigma^\perp) \to \mathbb{R}_{\geq 0}^{+\infty}$ determines the length of the range of output values from all the possible variations in the input variable $i \in \Delta$. This definition employs the auxiliary function $\textsc{Length} \in \wp(\mathbb{I}^{\pm\infty}) \to \mathbb{I}_{\geq 0}^{+\infty}$, defined as follows: $\textsc{Length}(X) \overset{\text{def}}{=} \sup \ X - \inf \ X$ if $X \neq \emptyset$, where sup and inf are the supremum and infimum operators, while $\textsc{Length}(X) \overset{\text{def}}{=} 0$ otherwise. Formally,

$$\textsc{Range}_i(S) \overset{\text{def}}{=} \sup_{s \in \Sigma|_\Delta} \textsc{Length}(\{\phi(s_\omega) \mid \langle s_0, s_\omega \rangle \in S \wedge s_0 =_{\Delta \setminus \{i\}} s\})$$

Similarly to $\textsc{Outcomes}$, $\textsc{Range}$ is monotone in the amount of dependencies S.

4 A Static Analysis for Quantitative Input Data Usage

In this section, we introduce a sound computable static analysis to determine an upper bound on the impact of an input variable i. The soundness of the approach leverages two elements: (1) an underlying abstract semantics $\Lambda^{\leftarrow}$ to compute an over-approximation of the dependency semantics $\Lambda^{\rightsquigarrow}$; and (2) a sound computable implementation of $\textsc{Impact}_i$, written $\texttt{Impact}_i^\natural$, used in the property $\mathscr{B}_i^{\leq k}$. All proofs can be found in the supplementary material [16, Appendix A].

To quantify the usage of an input variable, we need to determine the input configurations leading to specific output values. As our impact definitions $\textsc{Outcomes}_i$ and $\textsc{Range}_i$ measure over the different output values (i.e., $\phi(s_\omega)$) our underlying abstract semantics will be a *backward* (co-)reachability semantics starting from *disjoint* abstract post-conditions, over-approximating the (concrete) output values of the dependency semantics. Specifically, we abstract the concrete output values with an indexed set $B^\natural \in \mathbb{D}^{\natural^n}$ of n disjoint *output buckets*, where $\langle \mathbb{D}^\natural, \sqsubseteq, \perp^\natural, \top^\natural, \sqcup, \sqcap \rangle$ is an abstract state domain with concretization function $\gamma^\natural \in \mathbb{D}^\natural \to \wp(\Sigma^\perp)$. The choice of these output buckets is essential for obtaining a precise and meaningful analysis result.

For each output bucket $B^\natural_j \in \mathbb{D}^\natural$, our analysis computes an over-approximation of the dependency semantics restricted to the input configurations leading to $\gamma^\natural(B^\natural_j)$. More formally, let $\Lambda^{\rightsquigarrow}|_X \overset{\text{def}}{=} \{\langle s_0, s_\omega\rangle \in \Lambda^{\rightsquigarrow} \mid s_\omega \in X\}$ be the reduction of the dependency semantics $\Lambda^{\rightsquigarrow}$ to the dependencies with final states in X. Our static analysis is parametrized by an underlying backward abstract family[3] of semantics $\Lambda^{\leftarrow}[\![\mathrm{P}]\!] \in \mathbb{D}^\natural \rightarrow \mathbb{D}^\natural$ which computes the backward semantics $\Lambda^{\leftarrow}[\![\mathrm{P}]\!]B^\natural_j$ from a given output bucket $B^\natural_j \in \mathbb{D}^\natural$. The concretization function $\gamma^{\leftarrow} \in (\mathbb{D}^\natural \rightarrow \mathbb{D}^\natural) \rightarrow \mathbb{D}^\natural \rightarrow \wp(\Sigma \times \Sigma^\perp)$ employs $\gamma^\natural$ to restore all possible input-output dependencies, i.e., $\gamma^{\leftarrow}(\Lambda^{\leftarrow}[\![\mathrm{P}]\!])B^\natural_j \overset{\text{def}}{=} \{\langle s_0, s_\omega\rangle \mid s_0 \in \gamma^\natural(\Lambda^{\leftarrow}[\![\mathrm{P}]\!]B^\natural_j) \land s_\omega \in \gamma^\natural(B^\natural_j)\}$. We can thus define the soundness condition for the backward semantics with respect to the reduction of the dependency semantics.

Definition 1 (Sound Over-Approximation for $\Lambda^{\leftarrow}$). *For all programs* P*, and output bucket $B^\natural_j \in \mathbb{D}^\natural$, the family of semantics $\Lambda^{\leftarrow}$ is a* sound over-approximation *of the dependency semantics $\Lambda^{\rightsquigarrow}$ reduced with $\gamma^\natural(B^\natural_j)$, when it holds that:*

$$\Lambda^{\rightsquigarrow}[\![\mathrm{P}]\!]|_{\gamma^\natural(B^\natural_j)} \subseteq \gamma^{\leftarrow}(\Lambda^{\leftarrow}[\![\mathrm{P}]\!])B^\natural_j$$

We define $\Lambda^{\times} \in \mathbb{D}^{\natural n} \rightarrow \mathbb{D}^{\natural n}$ as the backward semantics repeated on a set of output buckets $B^\natural \in \mathbb{D}^{\natural n}$, that is, $\Lambda^{\times}[\![\mathrm{P}]\!]B^\natural \overset{\text{def}}{=} (\Lambda^{\leftarrow}[\![\mathrm{P}]\!]B^\natural_j)_{j\leq n}$. Again, the concretization function $\gamma^{\times} \in (\mathbb{D}^{\natural n} \rightarrow \mathbb{D}^{\natural n}) \rightarrow \mathbb{D}^{\natural n} \rightarrow \wp(\Sigma \times \Sigma^\perp)$ employs the abstract concretization $\gamma^\natural$ to restore all possible input-output dependencies over all the output buckets, i.e., $\gamma^{\times}(\Lambda^{\times}[\![\mathrm{P}]\!])B^\natural \overset{\text{def}}{=} \bigcup_{j\leq n}\{\langle s_0, s_\omega\rangle \mid s_0 \in \gamma^\natural((\Lambda^{\times}[\![\mathrm{P}]\!]B^\natural)_j) \land s_\omega \in \gamma^\natural(B^\natural_j)\}$.

Lemma 1 (Sound Over-Approximation for $\Lambda^{\times}$). *For all programs* P*, output buckets $B^\natural \in \mathbb{D}^{\natural n}$, and a family of semantics $\Lambda^{\leftarrow}$, the semantics $\Lambda^{\times}$ is a* sound over-approximation *of the dependency semantics $\Lambda^{\rightsquigarrow}$ when reduced to $\bigcup_{j\leq n}\gamma^\natural(B^\natural_j)$:*

$$\Lambda^{\rightsquigarrow}[\![\mathrm{P}]\!]|_{\bigcup_{j\leq n}\gamma^\natural(B^\natural_j)} \subseteq \gamma^{\times}(\Lambda^{\times}[\![\mathrm{P}]\!])B^\natural$$

Whenever the output buckets *cover* the whole output space, $\Lambda^{\times}$ is a sound over-approximation of $\Lambda^{\rightsquigarrow}$. The concept of covering for output buckets ensures that no final states of the dependency semantics, i.e. $\Omega^{\rightsquigarrow} \overset{\text{def}}{=} \{s_\omega \mid \langle s_0, s_\omega\rangle \in \Lambda^{\rightsquigarrow}\}$, are missed from the analysis.

Definition 2 (Covering). *We say that the output buckets $B^\natural \in \mathbb{D}^{\natural n}$ cover the whole output space whenever $\Omega^{\rightsquigarrow} \subseteq \bigcup_{j\leq n}\gamma^\natural(B^\natural_j)$.*

Next, we expect a sound implementation $\texttt{Impact}^\natural_i \in \mathbb{D}^{\natural n} \times \mathbb{D}^{\natural n} \rightarrow \mathbb{I}^{\pm\infty}$ to return a bound on the impact which is always higher than the concrete counterpart $\textsc{Impact}_i$.

Definition 3 (Sound Implementation). *For all output buckets $B^\natural$ and family of semantics $\Lambda^{\leftarrow}$, $\texttt{Impact}^\natural_i$ is a* sound implementation *of* $\textsc{Impact}$*, whenever*

$$\textsc{Impact}_i(\gamma^{\times}(\Lambda^{\times}[\![\mathrm{P}]\!])B^\natural) \leq \texttt{Impact}^\natural_i(\Lambda^{\times}[\![\mathrm{P}]\!]B^\natural, B^\natural)$$

[3] A family of semantics is a set of program semantics parametrized by an initialization.

The next result shows that our static analysis is sound when employed to verify the property of interest $\mathscr{B}_i^{\leq k}$ for the program P. That is, if $\texttt{Impact}_i^\natural$ returns the bound k', and $k' \leq k$, then the program P satisfies the property $\mathscr{B}_i^{\leq k}$, cf. $\mathrm{P} \models \mathscr{B}_i^{\leq k}$.

Theorem 1 (Soundness). *Let $\mathscr{B}_i^{\leq k}$ be the property of interest we want to verify for the program* P *and the input variable $i \in \Delta$. Whenever,*

(i) $\Lambda^{\leftarrow}$ *is sound with respect to* $\Lambda^{\leadsto}$*, cf. Def.* (1)*, and*
(ii) $B^\natural$ *covers the whole output space, cf. Def.* (2)*, and*
(iii) $\texttt{Impact}_i^\natural$ *is a sound implementation of* $\textsc{Impact}_i$*, cf. Def.* (3)*,*

the following implication holds:

$$\texttt{Impact}_i^\natural(\Lambda^\times [\![\mathrm{P}]\!] B^\natural, B^\natural) = k' \;\wedge\; k' \leq k \;\Rightarrow\; \mathrm{P} \models \mathscr{B}_i^{\leq k}$$

Finally, we define $\texttt{Range}_i^\natural$ and $\texttt{Outcomes}_i^\natural$ as possible implementations for $\textsc{Range}_i$ and $\textsc{Outcomes}_i$, respectively. We assume the underlying abstract state domain $\mathbb{D}^\natural$ is equipped with an operator $\texttt{Project}_i^\natural \in \mathbb{D}^\natural \to \mathbb{D}^\natural$ to project away the input variable i. For example, in the context of the interval domain, where each input variable is related to a possibly unbounded lower and upper bound, $\texttt{Project}_i^\natural(\langle i \mapsto [1,3], j \mapsto [2,4]\rangle) = \langle i \mapsto [-\infty,\infty], j \mapsto [2,4]\rangle$ removes the constraints related to i.

The definition of $\texttt{Outcomes}_i^\natural$ first projects away the input variable i from all the given abstract values, then it collects all intersecting abstract values via the meet operator $\sqcap$. These intersections represent potential concrete input configurations where variations on the value of i lead to changes of program outcome, from a bucket to another. We return the maximum number of abstract values that intersects after projections:

$$\texttt{Outcomes}_i^\natural(X^\natural, B^\natural) \overset{\text{def}}{=} \max\ \{|J| \mid J \in \texttt{IntersectAll}((\texttt{Project}_i^\natural(X_j^\natural))_{j\leq n})\}$$

Note the use of max instead of sup as in the concrete counterpart (Eq. (1)) since the number of intersecting abstract values is bounded by n, i.e., the number of output buckets. The function $\texttt{IntersectAll}$ takes as input an indexed set of abstract values and returns the set of indices of abstract values that intersect together, defined as follows:

$$\texttt{IntersectAll}(X^\natural \in \mathbb{D}^{\natural n}) \overset{\text{def}}{=} \{J \mid J \subseteq \mathbb{N} \wedge \forall j \leq n, p \leq n.\ j \in J \wedge p \in J \wedge X_j^\natural \sqcap X_p^\natural\}$$

Finding all the indices of intersecting abstract values is equivalent to find cliques in a graph, where each node represents an abstract value and an edge exists between two nodes if and only if the corresponding abstract values intersect. Therefore, $\texttt{IntersectAll}$ can be efficiently implemented based on the graph algorithm by Bron and Kerbosch [3].

Similarly, we define $\texttt{Range}_i^\natural$ as the maximum length of the range of the extreme values of the buckets represented by intersecting abstract values after projections. In such case, we assume $\mathbb{D}^\natural$ is equipped with an additional abstract operator $\texttt{Length}^\natural \in \mathbb{D}^\natural \to \mathbb{I}_{\geq 0}^{+\infty}$, which returns the length of the given abstract element, otherwise $+\infty$ if the abstract element is unbounded or represents multiple variables.

$$\begin{aligned}\texttt{Range}^\natural_i(X^\natural, B^\natural) &\overset{\text{def}}{=} \max\ \{\texttt{Length}^\natural(K) \mid K \in I\}\\ \text{where } I &= \{\sqcup\{B^\natural_j \mid j \in J\} \mid J \in \texttt{IntersectAll}((\texttt{Project}^\natural_i(X^\natural_j))_{j \leq n})\}\end{aligned}$$

In the supplementary material [16, Appendix A], we prove that the abstract counterparts $\texttt{Range}^\natural_i$ and $\texttt{Outcomes}^\natural_i$ are sound over-approximations of the concrete counterparts $\textsc{Range}_i$ and $\textsc{Outcomes}_i$.

5 Experimental Results

The goal of this section is to highlight the potential of our static analysis for quantitative input data usage. We implemented a proof-of-concept tool[4] in Python 3 that employs the `Interproc`[5] abstract interpreter to perform the backward analysis. Then, we exploited this tool to automatically derive a sound input data usage of three different scenarios. More use cases are shown in the supplementary material [16, Appendix B]. As each impact result must be interpreted with respect to what the program computes, we analyze each scenario separately.

Growth in a Time of Debt. Reinhart and Rogoff article "Growth in a Time of Debt" [19] proposed a correlation between high levels of public debt and low economic growth, and was heavily cited to justify austerity measures around the world. One of the several errors discovered in the article is the incorrect usage of the input value relative to Norway's economic growth in 1964. The data used in the article is publicly available but not the spreadsheet file. We reconstructed this simplified example based on the technical critique by Herndon et al. [12], and an online discussion[6]. The Program 2 computes the cross-country mean growth for the public debt-to-GDP $60 - 90\%$ category, key point to the article's conclusions. The input data is the average growth rate for each country within this public dept-to-GDP category. The problem with this computation is that Norway has only one observation in such category, which alone could disrupt the mean computation among all the countries. Indeed, the year that Norway appears in the $60 - 90\%$ category achieved a growth rate of 10.2%, while the average growth rate for the other countries is 2.7%. With such high rate, the mean growth rate raised to 3.4%, altering the article's conclusions. We assume growth rate values between -20% and 20% for all countries, consequentially, the output ranges are between these bounds as well. We instrumented the output buckets to cover the full output space in buckets of size 1, i.e., $\{t \leq \texttt{avg} < t + 1 \mid -20 \leq t \leq 20\}$. Results for both $\textsc{Outcomes}$ and $\textsc{Range}$ are shown in Table 2. The analysis discovers that the Norway's only observation for this

[4] `https://github.com/denismazzucato/impatto`
[5] `https://github.com/jogiet/interproc`
[6] `https://economics.stackexchange.com/q/18553`

Table 2: Quantitative input usage for Program 2 from the Reinhart and Rogoff's article.

Impact	portugal1	portugal2	portugal3	norway1	uk1	uk2	uk3	uk4	us1	us2	us3
Outcomes	5	5	5	10	2	2	2	2	3	3	3
Range	5	5	5	10	2	2	2	2	3	3	3

```python
def mean_growth_rate_60_90(
    portugal1, portugal2, portugal3,
    norway1,
    uk1, uk2, uk3, uk4,
    us1, us2, us3):
  portugal_avg = (portugal1 + portugal2 + portugal3) / 3
  norway_avg = norway1
  uk_avg = (uk1 + uk2 + uk3 + uk4) / 4
  us_avg = (us1 + us2 + us3) / 3
  avg = (portugal_avg + norway_avg + uk_avg + us_avg) / 4
```

Program 2: Program computing the mean growth rate in the $60 - 90\%$ category.

category `norway1` has the biggest impact on the output, as perturbations on its value are capable of reaching 10 different outcomes (cf. column `norway1`), while the other countries only have 5, 2, and 3, respectively for Portugal, UK, and US. The same applies to Range as the output buckets have size 1 and all the input perturbations are only capable of reaching contiguous buckets. Hence, we obtain the same exact results.

Our analysis is able to discover the disproportionate impact of Norway's only observation in the mean computation, which would have prevented one of the several programming errors found in the article. From a review of Program 2, it is clear that Norway's only observation has a greater contribution to the computation, as it does not need to be averaged with other observations first. However, such methodological error is less evident when dealing with a higher number of input observations (1175 observations in the original work) and the computation is hidden behind a spreadsheet.

GPT-4 Turbo. The second use case we present is drawn from Sam Altman's recent OpenAI keynote in September 2023[7], where he presented the GPT-4 Turbo. This new version of the GPT-4 language model brings the ability to write and interpret code directly without the need of human interaction. Hence, as showcased in the keynote, the user could prompt multiple information to the model, such as related to the organization of a holiday trip with friends in Paris, and

[7] `https://www.youtube.com/live/U9mJuUkhUzk?si=HOzuH3-gr_kTdhCt&t=2330`

Table 3: Results of the quantitative analysis for Program 3 and Program 4.

Impact	airbnb_total_cost_eur	flight_cost_usd	number_of_friends
Outcomes	10	17	9
Range	1099	1709	999

(a) Program 3 computing the share division among friends.

	x	y
	50	10
	499	99

~~(b)~~ Program 4.

```python
def share_division(
    airbnb_total_cost_eur,
    flight_cost_usd,
    number_of_friends):
  share_airbnb = airbnb_total_cost_eur / number_of_friends
  usd_to_eur = 0.92
  flight_cost_eur = flight_cost_usd * usd_to_eur
  total_cost_eur = share_airbnb + flight_cost_eur
```

Program 3: Program computing share division for holiday planning among friends.

the model automatically generates the code to compute the share of the total cost of the trip and run it in background. In this environment, users are unable to directly view the code unless they access the backend console. This limitation makes it challenging for them to evaluate whether the function has been implemented correctly or not, assuming users have the capability to do so. From the keynote, we extracted the Program 3 which computes the user's share of the total cost of a holiday trip to Paris, given the total cost of the Airbnb, the flight cost, and the number of friends going on the trip. Regarding the input bounds, users are willing to spend between 500 and 2000 for the Airbnb, between 50 and 1000 for the flight, and travel with between 2 and 10 friends. As a result, they expect their share, variable `total_cost_eur`, to be between 90 and 1900. To compute the impact of the input variables we choose the output buckets to cover the expected output space in buckets of size 100, i.e., $\{100t + 90 \leq \texttt{total\_cost\_eur} < \min\{100(t+1) + 90, 1900\} \mid 0 \leq t \leq 19\}$. The findings are similar for both the Outcomes and Range analysis, see Table 3a. The input variable `flight_cost_usd` has the biggest impact on the output, as perturbations on its value are capable of reaching 17 different output buckets (resp. a range of 1709 output values), while the other two, `airbnb_total_cost_eur` and `number_of_friends`, only reach 10 and 9 output buckets (resp. have ranges of size 1099 and 999), respectively.

```python
def example(x, y):
  counter = 0
  while x >= 0:
    if y <= 50:
      x += 1
    else
      x -= 1
    y += 1
    counter += 1
```

Program 4: Timing analysis.

These results confirm the user expectations about the proposed program from ChatGPT: the flight cost yields the biggest impact as it cannot be shared among friends.

Termination Analysis. Program 4 is adapted from the termination category of the software verification competition SV-COMP[8]. Assuming both input positives, $\mathtt{x}, \mathtt{y} \geq 0$, this program terminates in $\mathtt{x} + 1$ iterations if $\mathtt{y} > 50$, otherwise it terminates in $\mathtt{x} - 2\mathtt{y} + 103$ iterations. We define `counter` as the output variable, with output buckets defined as $\{10k \leq \mathtt{counter} < 10(k+1) \mid 0 \leq k < 50\}$ and $\{\mathtt{counter} \geq 500\}$. These output buckets represent cumulative ranges of iterations required for termination. The analysis results are illustrated in Table 3b, they show that the input variable `x` has the biggest impact. Modifying the value of `x` can result in the program terminating within any of the other 50 iteration ranges. On the other hand, perturbations on `y` can only result in the program terminating within 10 different iteration ranges. Such difference is motivated by the fact that `y` is only used to determine the number of iterations in the case where `y` is greater than 50, otherwise it is not used at all. Therefore, two values of `y`, e.g., y_0 and y_1, only result in two different ranges of iterations required to make the program terminate if either both of them are below 50 or $y_0 < 50 \land y_1 \geq 50$ or $y_0 \geq 50 \land y_1 < 50$, not in all the cases.

The given results can be interpreted as follows: the speed of termination of this loop is highly dependent on the value of `x`, while `y` has a much smaller impact.

Landing Risk System We apply our quantitative analysis to Program 1 for the landing alarm system extended with the continuous input space for the aircraft angle of approach, where $(-4 \leq \mathtt{angle} \leq 4) \land (1 \leq \mathtt{speed} \leq 3)$, see Figure 2. In this instance, the precision of the abstraction drastically drops as convex abstract domains are not able to capture the symmetric features of the input space around 0. Indeed, the analysis result, first row of Figure 3, is unable to reveal any difference in the input usage of input variables as all the abstract preconditions result of the backward analysis intersect together. As a consequence,

[8] `https://sv-comp.sosy-lab.org/`

Table 4: Quantitative input usage for Program 1.

Input Bounds		Outcomes		Range	
		`angle`	`speed`	`angle`	`speed`
$-4 \leq$ `angle` ≤ 4	$\wedge\, 1 \leq$ `speed` ≤ 3	3	3	3	3
$-4 \leq$ `angle` ≤ 0	$\wedge\, 1 \leq$ `speed` ≤ 3	3	2	3	2
$0 \leq$ `angle` ≤ 4	$\wedge\, 1 \leq$ `speed` ≤ 3	3	2	3	2

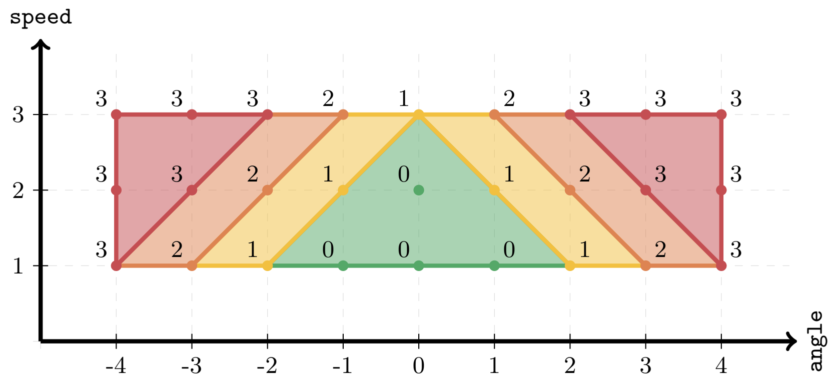

Fig. 2: Input space composition with continuous input values.

Outcomes and Range are unable to provide any meaningful information, first row of Table 4.

A possible approach to overcome the non-convexity of the input space is to split the input space into two subspaces (as a bounded set of disjunctive polyhedra), $-4 \leq$ `angle` ≤ 0 and $0 \leq$ `angle` ≤ 4, second and third row of Table 4. In the first subset $-4 \leq$ `angle` ≤ 0, we are able to perfectly captures the input regions that lead to each output bucket with our abstract analysis, second row of Figure 3. Therefore, we are able to recover the information that the input configurations from the bucket $\{$`risk` $= 3\}$ do not intersect with the ones from the bucket $\{$`risk` $= 0\}$ after projecting away the axis `speed`. As the end, our analysis notices that variations in the value of the input `angle` results in three possible output values, while variations in the `speed` input lead to two. Similarly, regarding the range of values, variations in the `angle` input cover the entire spectrum of output values, whereas to the `speed` input only span a range of 2 since it exists no input value such that modifications in the `speed` value could obtain a range of output values bigger than 2. The same reasoning applies to the other subspace with $0 \leq$ `angle` ≤ 4.

6 Related Work

Given the connection between *qualitative* input usage and information flow analyses [22], to design a quantitative input usage analysis that fits our purposes,

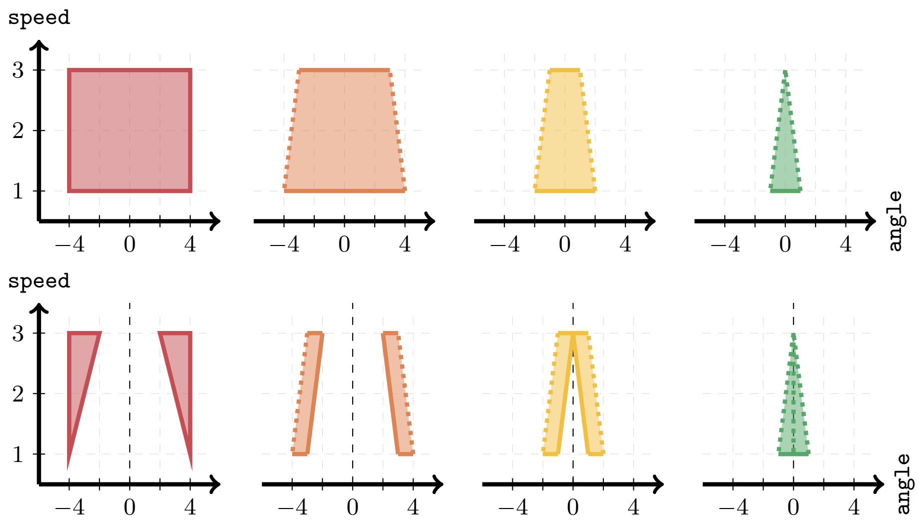

Fig. 3: Above, result of the analysis with convex polyhedra. Below, result after splitting the input space into two subspaces around `angle` $= 0$.

it may come natural to use *quantitative* information flow [10, 11]. Such analyses measure information leakage about a secret through the concept of entropy, based on observations of the program's output values. Remarkably, this similarity between entropy and our notion of impact is even more evident in the work proposed by Köpf and Rybalchenko [13] which quantifies an upper bound of the entropy of a program's input variables by computing an over-approximation of the set of input-output observations, sometimes called equivalence classes. They employ Shannon entropy, min-entropy, and other entropies through the enumeration of these equivalence classes and their respective sizes. The equivalence classes are partitions of the input space in which two input assignments belong to the same class whenever the program produces an equivalent output. For example, the equivalence classes of the program L are $\mathit{\Pi}(\mathrm{L}) = \{\{\langle x, y\rangle \mid \mathrm{L}(x, y) = z\} \mid z \in \{0, 1, 2, 3\}\} = \{\{\langle 1, 1\rangle\}, \{\langle 1, 2\rangle\}, \{\langle 1, 3\rangle\}, \{\langle -4, 1\rangle, \langle -4, 2\rangle, \langle -4, 3\rangle\}\}$. We developed our impact definitions by adapting entropy measures to our needs in three successive attempts.

Initially, we notice that Shannon-entropy computes the average uncertainty of input values based on observations of the program's outcomes, while min-entropy, defined as:

$$H_\infty(\mathrm{P}) \overset{\text{def}}{=} \log_2 \frac{|\textsc{Input}_\mathrm{P}|}{|\mathit{\Pi}(\mathrm{P})|}$$

computes the worst-case uncertainty, where $\textsc{Input}_\mathrm{P}$ is the set of all input values of a given program P. As a first attempt, we consider min-entropy as closer to our needs since our aim is to discover the worst-case impact, i.e., the case in which a variable contributes the most. By computing min-entropy on the program L, we obtain $H_\infty(\mathrm{L}) = 0.58$, indicating that the input is highly guessable. Indeed, when the risk level is 3, the potential values of input variables are `angle` $= -4$ and `speed` $\in 1, 2, 3$; for all other output values, the input values are completely determined. Unfortunately, min-entropy lacks granularity and measures the uncertainty of the input variables collectively. Instead, our aim is to quantify the individual contributions.

To address the previous issue, as a second attempt we exploit low and high labels for input variables, where the former are considered as public, available to the attacker, and the latter as secret. To assess the impact of each input variable, we prioritize one high variable at a time, considering all others as low variables. Subsequently, we compute the min-entropy of the labelled program to quantify the extent of the impact. We define $\mathrm{L}^{\texttt{angle}}(x) \overset{\text{def}}{=} \langle \mathrm{L}(x,1), \mathrm{L}(x,2), \mathrm{L}(x,3)\rangle$ which represents the sequence of programs where `angle` is high and `speed` is low. Similarly, $\mathrm{L}^{\texttt{speed}}(y) \overset{\text{def}}{=} \langle \mathrm{L}(-4,y), \mathrm{L}(1,y)\rangle$ where `speed` is high and `angle` is low. Computing $H_\infty(\mathrm{L}^{\texttt{angle}})$ and $H_\infty(\mathrm{L}^{\texttt{speed}})$ yields 0 on both because all equivalence classes consist of singletons, meaning the number of inputs equals the number of outputs. Thus, there's no uncertainty in the value of `angle` given outputs of $\mathrm{L}^{\texttt{angle}}$, or in the value of `speed` given outputs of $\mathrm{L}^{\texttt{speed}}$. Indeed, observing the output $\langle 3, 3, 3\rangle$ from the program $\mathrm{L}^{\texttt{angle}}$ implies that `angle` is -4, while observing $\langle 0, 1, 2\rangle$ implies that `angle` is 1. The same applies to $\mathrm{L}^{\texttt{speed}}$ where observing $\langle 3, 0\rangle$ implies `speed` $= 1$, $\langle 3, 1\rangle$ implies `speed` $= 2$, and $\langle 3, 2\rangle$ implies `speed` $= 3$. However, this approach does not isolate the contributions of high variables; these outcomes are combined into a tuple of values through the return statement and thus evaluated together. Consequently, min-entropy cannot distinguish the contribution of each input variable independently.

An immediate solution is to develop a similar approach to the one used for the high-low variables, but instead of using min-entropy for the derived programs ($\mathrm{L}^{\texttt{angle}}$ and $\mathrm{L}^{\texttt{speed}}$), we count the number of outcomes of the partially-applied programs, cf. programs $\mathrm{L}(x,1), \mathrm{L}(x,2)$, and $\mathrm{L}(x,3)$ for $\mathrm{L}^{\texttt{angle}}$; $\mathrm{L}(-4,y)$ and $\mathrm{L}(1,y)$ for $\mathrm{L}^{\texttt{speed}}$. These programs are referred to as $\mathrm{L}^{\texttt{angle}}_y$ and $\mathrm{L}^{\texttt{speed}}_x$ respectively. Therefore, the third attempt defines $H_{\mathrm{O}}(\mathrm{L}^{\texttt{angle}}) \overset{\text{def}}{=} \max\{|\mathit{\Pi}(\mathrm{L}^{\texttt{angle}}_y)| \mid y \in \{1,2,3\}\}$ and $H_{\mathrm{O}}(\mathrm{L}^{\texttt{speed}}) \overset{\text{def}}{=} \max\{|\mathit{\Pi}(\mathrm{L}^{\texttt{speed}}_x)| \mid x \in \{-4,1\}\}$ retaining the maximum to obtain the worst-case scenario. As a result, $H_{\mathrm{O}}(\mathrm{L}^{\texttt{angle}}) = 2$ and $H_{\mathrm{O}}(\mathrm{L}^{\texttt{speed}}) = 3$. This means that variations of the value of `angle` result in at most 2 different outputs, while variations of the value of `speed` result in at most 3. Effectively, this is the first notion of impact derived from min-entropy capable of discriminating the contribution of each input variable on the program computation, exploiting the number of reachable outcomes from variations of the value of the input variable under consideration. Indeed, $H_{\mathrm{O}}(\mathrm{P}^{i}) = \textsc{Outcomes}_{i}(\mathrm{P})$ holds for a generic program P.

Overall, entropy measures and the approach proposed by Köpf and Rybalchenko [13] can be adapted to our needs. Nevertheless, their analysis grows exponentially with the number of low variables, which in our adaptation corresponds to the number of inputs, minus one. To address this limitation, we leverage an over-approximation of input-output observations of the program, focusing solely on the low variables. By doing so, we obtain the set of input configurations that lead to the same output value by variation on the value of high variables. As a result, our approach performs the backward analysis only one time per output bucket, independently of the number of low variables. A similar technique could also be used to mitigate such explosion in their work.

Other works include Chothia et al. [6], which proposed a statistical approach to quantify information leakage for Java programs. Phan et al. [18] and Saha et al. [20] employed symbolic execution techniques and model counting to obtain sound bounds on the entropy of programs. Other static analyses, e.g., Assaf et al. [1] and Clark et al. [7], are based on abstract interpretation. The key difference among our framework and their work is the information we measure. Our analysis quantifies the effect of each of the input variables on the program outcome, focusing on numerical properties, while quantitative information flow usually measures quantities in terms of bits of information transferred from the input variables to the program outcome, more specific to security properties. For example, Assaf et al. [1] counts how many bits of information of input variables are used to compute the result; Smith [21] computes the probability of guessing the value of a private variable from the value of input variables; and McCamant and Ernst [17] retrieves the channel capability, which is the worst-case possible for information leakages. Most of these approaches are based on the concept of entropy measures, which are orthogonal to our approach. Instead, we present a more fine-grained analysis, which discovers the impact of each input variable separately.

7 Conclusion

In this work, we presented an automated and sound analysis to statically quantify the usage of input variables based on a given impact definition. Our static analysis employs a backward analysis to compute an over-approximation of the precondition of the program, starting from the set of output buckets. While a forward approach based on input space partitioning seems plausible it is not practical due to the necessity of guessing the correct input partitions to obtain a meaningful result. On the other hand, the backward analysis directly starts from output values, thus avoiding the necessity of guessing the correct input partitions. Additionally, the input space partitions of the forward analysis need to not limit the axis of the input variable under consideration, otherwise the impact computed on two different partitions cannot be summed together, thus limiting the expressiveness of possible input partitions. This limitation is not present in the backward analysis as the expressiveness of the input invariant computed by the backward analysis is limited only by the chosen abstraction.

As a future work, we plan to develop a modular tool to support the analysis in a solid and extensible way. We also plan to introduce heuristics able to automatically (or iteratively) infer the output buckets, since the choice of the starting buckets is essential to our quantitative analysis. Future directions could extend fairness certification studies on neural network models [23, 15]. Our quantitative notion introduces a quantitative fairness measure. Another promising direction is the exploration of new impact definitions for cyber-physical systems [14]. It could also be interesting to exploit an impact definition to analyze the impact of abstract domains in static program analyzers, e.g., by using pre-metrics as defined in [4, 5]. Developing new relational abstract domains to discover specific non-linear variable relations could drastically improve the analysis precision, additionally taking into account input distributions. Further investigations of our analysis could also reveal new perspectives in the context of timing side-channel attacks [24], broadening the practical applications of our research.

Acknowledgements We thank Michele Pasqua for suggestions on related work, Jérôme Boillot for his thorough review, and the anonymous reviewer for valuable feedback. This work was partially supported by the SAIF project, funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference ANR-23-PEIA-0006.

References

[1] M. Assaf, D. A. Naumann, J. Signoles, Éric Totel, and F. Tronel. Hypercollecting semantics and its application to static analysis of information flow. 2017. `https://doi.org/10.1145/3009837.3009889`.

[2] D. W. Barowy, D. Gochev, and E. D. Berger. Checkcell: data debugging for spreadsheets. *OOPSLA 2014*. `https://doi.org/10.1145/2660193.2660207`.

[3] C. Bron and J. Kerbosch. Finding all cliques of an undirected graph (algorithm 457). *Commun. ACM 1973*.

[4] M. Campion, M. Dalla Preda, and R. Giacobazzi. Partial (in)completeness in abstract interpretation: limiting the imprecision in program analysis. *POPL 2022*, . `https://doi.org/10.1145/3498721`.

[5] M. Campion, C. Urban, M. Dalla Preda, and R. Giacobazzi. A formal framework to measure the incompleteness of abstract interpretations. *SAS 2023*, . `https://doi.org/10.1007/978-3-031-44245-2_7`.

[6] T. Chothia, Y. Kawamoto, and C. Novakovic. Leakwatch: Estimating information leakage from java programs. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2014. `https://doi.org/10.1007/978-3-319-11212-1_13/COVER`.

[7] D. Clark, S. Hunt, and P. Malacaria. A static analysis for quantifying information flow in a simple imperative language. *Journal of Computer Security*, 2007. `https://doi.org/10.3233/JCS-2007-15302`.

[8] P. Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci. 2002*. `https://doi.org/10.1016/S0304-3975(00)00313-3`.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL 1977*. `https://doi.org/10.1145/512950.512973`.

[10] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982.

[11] J. W. Gray. Toward a mathematical foundation for information flow security. *IEEE Computer Society 1991*. `https://doi.org/10.1109/RISP.1991.130769`.

[12] T. Herndon, M. Ash, and R. Pollin. Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff. *Cambridge Journal of Economics 2014*. `https://doi.org/10.1093/cje/bet075`.

[13] B. Köpf and A. Rybalchenko. Automation of quantitative information-flow analysis. *SFM 2013*. `https://doi.org/10.1007/978-3-642-38874-3_1`.

[14] M. Kwiatkowska. Advances and challenges of quantitative verification and synthesis for cyber-physical systems. *2016 Science of Security for Cyber-Physical Systems Workshop (SOSCYPS)*, 2016. `https://doi.org/10.1109/SOSCYPS.2016.7579999`.

[15] D. Mazzucato and C. Urban. Reduced products of abstract domains for fairness certification of neural networks. *SAS 2021*. `https://doi.org/10.1007/978-3-030-88806-0_15`.

[16] D. Mazzucato, M. Campion, and C. Urban. Quantitative Input Usage Static Analysis. *2023*. URL `https://hal.science/hal-04339001`. Supplementary material.

[17] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008. `https://doi.org/10.1145/1375581.1375606`.

[18] Q.-S. Phan, P. Malacaria, O. Tkachuk, and C. S. Păsăreanu. Symbolic quantitative information flow. *ACM SIGSOFT Software Engineering Notes*, 2012. `https://doi.org/10.1145/2382756.2382791`.

[19] C. M. Reinhart and K. S. Rogoff. Growth in a time of debt. *American Economic Review 2010*. `https://doi.org/10.1257/AER.100.2.573`.

[20] S. Saha, U. S. Barbara, U. S. Ghentiyala, and U. L. Shihua. Obtaining information leakage bounds via approximate model counting. 2023. `https://doi.org/10.1145/3591281`.

[21] G. Smith. On the foundations of quantitative information flow. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2009. `https://doi.org/10.1007/978-3-642-00596-1_21/COVER`.

[22] C. Urban and P. Müller. An abstract interpretation framework for input data usage. *ESOP 2018*. `https://doi.org/10.1007/978-3-319-89884-1_24`.

[23] C. Urban, M. Christakis, V. Wüstholz, and F. Zhang. Perfectly parallel fairness certification of neural networks. *OOPSLA 2020*. `https://doi.org/10.1145/3428253`.

[24] W. H. Wong. Timing attacks on RSA: revealing your secrets through the fourth dimension. *ACM Crossroads 2005*. `https://doi.org/10.1145/1144396.1144401`.