Quantitative Static Timing Analysis

Anonymous

Anonymous

Abstract. Programming errors in software applications can often be difficult to detect, as they may appear without clear indications of failure. One such example is when certain input variables have an unexpected impact on the program's behavior. As an indicator of the program's runtime behavior, this work studies the impact of input variables on the number of loop iterations in a program. Such information is valuable for debugging, optimizing performance, and analyzing security vulnerabilities, such as in side-channel attacks where execution times can be exploited. To address this issue, we propose a sound static analysis based on abstract interpretation to quantify the impact of each input variable on the global number of iterations. Our approach combines a dependency analysis with a global loop bound analysis to derive an over-approximation of the impact quantity. We demonstrate our prototype tool in the S2N-Bignum library for cryptographic systems to certify the absence of timing side-channels.

1 Introduction

Detecting programming errors that yield a plausible yet incorrect or unsafe behavior is a challenging task. Particularly when these errors do not lead to obvious failures, like program crashes, but may degrade performance or introduce security vulnerabilities. One source of such errors may arise from the unexpected impact of input variables on the program's runtime. For instance, in the context of security, an unexpected impact of input variables on the program's runtime could reveal sensitive information [46], leading to potential security threats. Even in cryptography, where programs are mathematically robust, vulnerabilities to timing attacks persist, depending on the implementation choices and design. Kocher [22] demonstrated that widely used public key cryptographic algorithms, like RSA, are vulnerable to timing attacks, and may leak information about the secret key. The value of such attacks lies in their simplicity; attackers do not need to possess detailed knowledge of the program implementation or engage in computationally expensive operations. Merely the information of which primitives are used in the program is enough. For example, knowledge that the exponentiation operation, commonly found on cryptographic programs, is performed using the square-and-multiply algorithm can enable an attacker to exploit the program's runtime to infer the secret key [46]. Indeed, Dhem et al. [14] mounted such attack against CASCADE smart cards, observing timing differences during the square operations of the square-and-multiply algorithm. Furthermore, knowing the timing behavior of a program could certify intended behavior or reveal latent flaws by matching developers' expectations with the actual program

behavior. For performance optimization, identifying input variables that most significantly affect loop iterations can help developers focus on critical code segments [31]. As a consequence, achieving a comprehensive understanding of the impact of input variables on the program runtime is paramount. In this study we focus on quantifying the impact of input variables on the number of loop iterations in a program, as an indicator of the program's runtime behavior.

Related issues such as worst-case execution time [45], timing side-channel attacks [24, 33, 37], and termination analyses [41, 35] have been thoroughly addressed in the literature. Unlike worst-case execution time analysis, which generates an invariant on loop iterations, our approach quantifies the impact of each input variable across this invariant. Our work complements timing side-channel attack analyses by providing a quantitative measure. While quantitative information flow analyses [25, 13, 18] theoretically could infer similar information, they are designed to measure a quantity considering all the sensitive input variables together, rather than individually.

In this paper, we propose a static analysis based on abstract interpretation to quantify the impact of input variables on global number of iterations. We leverage an underlying global loop bound analysis to derive an over-approximation of the global loop bound and encode the quantification of each input variable's impact as a linear programming problem. Our approach blends syntactic and semantic information: to improve accuracy, the global loop bound analysis generates invariants as a set of linear constraints; to improve scalability, we combine the global loop bound analysis with a syntactic dependency analysis [43], reducing the number of variables to analyze.

Our approach is implemented in TimeSec: a sound, automatic, and open-source static analyzer. We demonstrate the effectiveness of our tool in the real-world library S2N-Bignum[1] for cryptographic applications. Notably, we certify the S2N-Bignum library's immunity to timing side-channel attacks on certain numerical input variables by showing that they have no impact on loop iterations. Additionally, we evaluate TimeSec against programs drawn from SV-Comp in Appendix A.

Contributions. We claim the following contributions:

1. In Section 4, we define quantitative input data usage for global loop bounds, our property of interest.
2. In Section 5, we propose a static analysis based on abstract interpretation, employing a linear constraint abstract domain, global loop bound analysis, and linear programming encoding to quantify input variable impact on loop iterations. We discuss implementation features of TimeSec for scalability and efficiency in Section 6.
3. Finally, in Section 7, we evaluate TimeSec against the S2N-Bignum library.

[1] `https://github.com/awslabs/s2n-bignum`

```
def Add(p, z, m, x, n, y):
  r = min(p, m)
  s = min(p, n)
  if (r < s):
    t = p - s
    q = s - r
    # i = 0
    # a = 0
    for (; r > 0; r--):
      # s = x[i]
      # w = y[i]
      # z[i] = s + w + a
      # i = i + 1
      # a = (w < a) ||
      #     (s + w < s) ||
      #     (s + w + a < s)
    do:
      # r = y[i]
      # b = (r < a) ||
      #     (r + a < r)
      # z[i] = r + a
      # i = i + 1
      q--
      # a = b
    while (q > 0)
  ...
...
else:
  t = p - r
  q = r - s
  # i = 0
  # b = 0
  for (; s > 0; s--):
    # r = x[i]
    # w = y[i]
    # z[i] = r + w + b
    # i = i + 1
    # b = (w < b) ||
    #     (r + w < r) ||
    #     (r + w + b < r)
  for (; q > 0; q--):
    # r = x[i]
    # z[i] = r + b
    # i = i + 1
    # b = (r < b) ||
    #     (r + b < r)
if (t > 0):
  # z[i] = b
  while (t > 0):
    # i = i + 1
    t--
    # if (t > 0):
      # z[i] = 0
```

Fig. 1: Program `Add`, computing the sum of two numbers `x` and `y` into `z`.

2 Overview

In this section, we present an overview of our quantitative analysis using the program depicted in Figure 1, referred to as `Add`, which resembles the addition function decompiled from the S2N-BIGNUM library [3]. The statements that are not relevant to the number of iterations of loops are commented out (cf. #). The goal of program `Add` is to compute the sum of two given numbers `x` and `y`, storing the result into `z`. The input variables `x`, `y`, and the output `z` are represented in the form of arrays, respectively of length `m`, `n`, and `p`, of 64-bit unsigned integers. `Add` computes the column addition of the two input arrays. For instance, assuming $\texttt{x} = [3\ 8]$, $\texttt{y} = [4]$ and size of `z` is 3, then $\texttt{Add}(3, \texttt{z}, 2, [3\ 8], 1, [4])$ computes:

$$\begin{array}{r} [3\ \ 8] + \\ \quad [4] = \\ \hline [0\ \ 4\ \ 2] \end{array}$$

where the result is stored back into `z`, available in the calling context of the function.

Table 1: A few executions to show how the many times the program `Add` iterates. The symbol " $*$ " represents any value.

`Add( p, z, m, x, n, y )`	$\rightsquigarrow$	global number of iterations
`Add(` 0, $*$, 0, $*$, 0, $*$ `)`	$\rightsquigarrow$	0
`Add(` 1, $*$, 0, $*$, 0, $*$ `)`	$\rightsquigarrow$	1
`Add(` 2, $*$, 0, $*$, 0, $*$ `)`	$\rightsquigarrow$	2
`Add(` 0, $*$, 1, $*$, 0, $*$ `)`	$\rightsquigarrow$	0
`Add(` 1, $*$, 1, $*$, 0, $*$ `)`	$\rightsquigarrow$	1
`Add(` 2, $*$, 1, $*$, 0, $*$ `)`	$\rightsquigarrow$	2
`Add(` 0, $*$, 0, $*$, 1, $*$ `)`	$\rightsquigarrow$	0
`Add(` 1, $*$, 0, $*$, 1, $*$ `)`	$\rightsquigarrow$	1
`Add(` 2, $*$, 0, $*$, 1, $*$ `)`	$\rightsquigarrow$	2

Our goal is to quantify the impact of each input variable on the number of loop iterations. The bigger the impact, the more influence the input variable may have on the runtime.

Our Approach. We abstract the runtime of the program by the sum of the number of iterations of all the loops. For clarity, Figure 1 highlights only the statements of the program `Add` that are relevant to the loop iterations. Specifically, `Add` starts by computing the minimum length of the input numbers `x` and `y` compared to `z`, and stores the results into `r` and `s`, respectively Line 2 and 3. Then, the first conditional statement (Line 4) checks whether `r` is smaller than `s`. If true, it computes two loops, the first one (Line 9) iterates `r` times to sum `x` and `y` until completion of `x`, and the second one (Line 17) iterates $\texttt{s} - \texttt{r}$ times over the remaining of `y`. On the other hand, the else branch (Line 27) computes a loop of `s` iterations to sum `x` and `y` until completion of `y`, and another of $\texttt{r} - \texttt{s}$ iterations over the remaining of `x`, respectively Line 32 and 40. At the end, a final loop (Line 48) iterates `t` times to apply padding in the array `z`.

It is trivial to conclude that the number of iterations in `Add` does not depend on the values of the input arrays `x`, `y`, and `z`, as no relevant statement in Figure 1 uses them. On the other hand, determining that even the length variables `m` and `n` do not influence the number of iterations of the loops in `Add`, is quite harder as it requires a precise numerical invariant. From Table 1, we can observe that the number of iterations of the program `Add` is influenced solely by the input variable `p` as only variations in its value lead to different numbers of iterations. In fact, the total number of iterations performed by the program, referred to as the *global number of iterations*, is exactly the initial value of `p`, which is the length of the output array `z`. This is the information our analysis aims to capture.

Assuming a deterministic system, an input variable `x` has an influence on the global number of iterations when, by fixing the values of all the other variables, there exist (at least) two different input values for `x` that lead to a different global number of iterations. In our case, consider the first and second row of Table 1, these two executions differ in the value of the input variable `p` (cf. respectively

of value 0 and 1), and they lead to a different global number of iterations (cf. respectively 0 and 1). Whereas, the variations of the input variable `m` do not affect the global number of iterations, e.g., both the first and the fourth row of Table 1 (cf. respectively of value 0 and 1 for `m`) lead to 0 iterations. Generally, $\texttt{Add}(i, \dots)$, for $i \geq 0$, leads to i iterations. In our framework, we quantify such impact by considering the range of the global number of iterations from variations of the input variable under analysis. In the example of Figure 1, if we assume $\texttt{p}, \texttt{m}, \texttt{n} \in [0, u]$ where $u \in \mathbb{N}$, then the length of the range of possible values for the global number of iterations from variations in the value of `p` is u; that is, our impact measure. Note that, the impact quantity does not always coincide with the upper bound on the global number of iterations. For instance, the impact of `m` and `n` is 0 as the global number of iterations is not influenced by their values. Our goal is to develop a static analysis capable of quantifying the impact of each input variable to the global number of iterations of a program. Specifically, our impact quantity measures the variations on the number of loop iterations for terminating executions, as non-terminating executions yield an infinite number of iterations.

Static Analysis. Global loop bound analyses [8, 38] consider the aggregate behavior of all the loops of a program to generate an invariant with respect to the global number of iterations. In our work, we aim to quantify the contribution of each input variable with respect to the global loop bounds. To this end, we propose a static analysis that employs an underlying global loop bound analysis and a linear programming encoding to automatically infer a sound upper bound on the impact of each input variable.

In order to make the analysis feasible, we first identify which statements in the program under analysis are relevant to the computation of the global number of iterations. We do so by employing a syntactic dependency analysis [43] to collect, for each program statement, an over-approximation of the set of program variables that influence the loop iterations. Then, a global loop bound analysis infers a sound upper bound on the global number of iterations using the program simplified by the syntactic dependency analysis. Figure 1 is already the result of the syntactic dependency analysis, where irrelevant statements are commented out. The global loop bound analysis discovers the following invariant:

$$\texttt{nit} = p \ \wedge \ 0 \leq p \leq u \tag{1}$$

where p refers to the initial value of `p`, and `nit` is the global number of iterations.

Impact Quantification. We employ a linear programming encoding based on the invariant generated by the global loop bound analysis, cf. Equation (1). Given an input variable of interest, we isolate its contribution by removing it from the invariant. For example, removing p from "$\texttt{nit} = p \ \wedge \ 0 \leq p \leq u$" results in:

$$0 \leq \texttt{nit} \leq u \tag{2}$$

this operation is called the projection operation. We quantify the impact by the range of all the possible values of `nit` resulting from perturbations of an input

variable. Computationally, this impact quantity is the distance k between the maximum and minimum values of $\texttt{nit}$, denoted by $\overline{\texttt{nit}}$ and $\underline{\texttt{nit}}$ respectively. Our linear programming problem aims to maximize the distance k, subject to the constraints defined by two projected invariants: Equation (2) where $\overline{\texttt{nit}}$ and $\underline{\texttt{nit}}$ respectively substitute $\texttt{nit}$, allowing them to vary independently. Formally, to compute the impact of $\texttt{p}$, we solve the following linear programming problem:

$$\begin{aligned} \text{maximize} \quad & k \\ \text{subject to} \quad & 0 \leq \overline{\texttt{nit}} \leq u \\ \wedge \quad & 0 \leq \underline{\texttt{nit}} \leq u \\ \wedge \quad & 0 \leq k \leq \overline{\texttt{nit}} - \underline{\texttt{nit}} \end{aligned}$$

which is achieved at $k = u$. On the other hand, we consider the impact of the other input variables $\texttt{m}$ and $\texttt{n}$. To this end, we remove $\texttt{m}$ (or equivalently $\texttt{n}$) from the invariant Equation (1). Since $\texttt{m}$ and $\texttt{n}$ do not appear in the invariant, the projection operation does not modify it. Hence, after substituting the maximum and minimum values of $\texttt{nit}$, we solve the following linear programming problem:

$$\begin{aligned} \text{maximize} \quad & k \\ \text{subject to} \quad & \overline{\texttt{nit}} = p \ \wedge \ 0 \leq p \leq u \\ \wedge \quad & \underline{\texttt{nit}} = p \ \wedge \ 0 \leq p \leq u \\ \wedge \quad & 0 \leq k \leq \overline{\texttt{nit}} - \underline{\texttt{nit}} \end{aligned}$$

Clearly, k maximizes at 0 as it holds that $\overline{\texttt{nit}} = \underline{\texttt{nit}} = p$. As a result, with the combination of the syntactic dependency analysis, the global loop bound analysis, and the linear programming encoding, we are able to over-approximate the impact of each input variable on the global number of iterations of a program. Section 5.3 includes a more complex example to demonstrate the capabilities of the linear programming encoding, especially in scenarios where the invariant computed by the global loop bound analysis does not already reveal the impact of an input variable.

Timing Side Channel Attacks. When the program `Add` is employed in cryptographic applications, their safety relies also on the safety of `Add`. A potential attacker could exploit the timing behavior of the program `Add` to infer secret information of the underlying cryptographic system. For instance, consider the fictitious cryptographic program `Check` of Figure 2. It utilizes `Add` to check whether a user provided input number, represented by the array `key` of length `p`, is equal to a secret. Such secret has been previously negated and stored in the array `secret`, of length `p`. Therefore, to check whether the user provided input is equal to the secret, the program `Check` computes the sum of the user input and the secret, then checks if the result is all zeros. In the context of timing side-channel attacks, an attacker could exploit the timing behavior of the program `Check` to infer the secret information stored in the array `secret`. Specifically, the number of iterations of the program `Check` is given by the initial value of the variable `p`

```
53 def Check(key, secret, p):
54   z = malloc(p)
55   Add(p, z, p, secret, p, key)
56   out = 1
57   for i in range(p):
58     if z[i] != 0:
59       out = 0
60   return out
```

Fig. 2: Program `Check` employs the program `Add` to check whether a user provided input `key` is equal to a hidden secret `secret`.

(for loop at Line 57), plus the number of iterations of the program `Add` (Line 55). If the number of iterations of the program `Add` were influenced by the values of the input array `x` (instantiated with `secret` in Line 55), then the attacker could exploit the runtime of the program `Add` to infer the secret information. As the S2N-BIGNUM library [3] is designed for cryptographic applications, certifying that the runtime of its primitives depends only on the nominal length variables (cf. `p`, `m`, `n`) rather than their values (cf. `z`, `x`, `y`) is crucial to ensure the security of cryptographic applications that employ it.

With our work, an impact strictly greater than 0 on the global number of iterations means that there is a potential timing side-channel over the input variable under analysis.

3 Related Work

Loop bound analyses are widely studied in the literature, as they are essential for various program analyses, including termination analysis, WCET analysis, and side-channel analysis. In this section, we discuss the most relevant works in these areas and compare them with our approach.

Worst-Case Execution Time. Worst-Case Execution Time (WCET) Analysis aims to derive sound upper bounds on the execution time of programs (see Wilhelm et al. [45] for a survey). On a broader perspective, the problem of deriving an upper bound on the number of loop iterations is a particular case of the problem of deriving loop bounds [5, 15], which aims to infer invariants on the number of iterations of a single specific loop. Global loop bound analyses, instead, consider the aggregate behavior of all the loops of a program [8, 38]. Our work is orthogonal to the existing literature on WCET and loop bound analysis as we do not generate invariants on the global number of iterations itself. Instead, we focus on quantifying the impact on the global number of loop iterations of each input variable individually. However, we leverage an underlying global loop bound analysis to understand the input-output relationship involving the global loop counter.

Side-Channels Analysis. Side-channel attacks exploit vulnerabilities in cryptographic system implementations rather than directly targeting computational complexity [22]. These attacks, including those based on power consumption [23] and speculative executions [21, 27], can leak sensitive information without physical tampering [46]. Various methods have been proposed to quantify the information leaked through side-channels. The quantitative analysis of side-channel attacks, initially introduced by Köpf and Basin [24], utilizes model counting and a greedy algorithm to mitigate the exponential growth of potential paths that must be examined. Phan et al. [33, 34] and Saha et al. [37] apply symbolic execution to enumerate the equivalence classes of program's output values, and utilizes entropy measures to quantify the information leakage. Other static analyses, e.g., Assaf et al. [2] and Clark et al. [9], are instead based on abstract interpretation. However, our work differs by quantifying the impact of each input variable separately, rather than providing a comprehensive quantification for all input variables. This fine-grained analysis is essential for identifying specific variables that significantly influence program behavior. While quantitative analyses theoretically could infer similar information, their entropy measures are not designed for this purpose [29], and they usually require k-times self-composition [4, 40], which is computationally expensive [1]. Instead, our approach only requires a single abstract analysis and leverages variable projections to separate the contributions of each input variable [44, 28].

Input Data Usage. The syntactic dependency analysis proposed by Urban and Müller [43] can be seen as the qualitative counterpart to our quantitative definition. It holds that whenever the measured impact is zero, the input variable is definitely unused, w.r.t. the number of iterations of the loop. Indeed, by considering only the syntactic dependencies, we would have obtained an abstraction of the input data usage property. The qualitative definition of Urban and Müller [43] additionally considers non-deterministic systems, we discuss a possible extension to non-deterministic systems in Section 8.

4 Quantitative Input Data Usage for Global Loop Bounds

In this section, we present some preliminaries on program computations, then we introduce our property of interest – quantitative input data usage for global loop bounds.

4.1 Program Semantics

We introduce a simple sequential programming language which we use for illustration throughout the rest of the paper. The variables are statically allocated, represented by the set $\mathbb{V}$, and the only data type is the numerical values in $\mathbb{I} \in \{\mathbb{N}, \mathbb{Z}, \mathbb{R}\}$. The finite set $\Delta \subseteq \mathbb{V}$ contains the input variables. The syntax of

$$
\begin{aligned}
&\Lambda^{\rightarrow}[\![\texttt{skip}]\!]S \stackrel{\text{def}}{=} S \\
&\Lambda^{\rightarrow}[\![\texttt{x} := e]\!]S \stackrel{\text{def}}{=} \{s[\texttt{x} \leftarrow v] \mid v \in \mathcal{A}[\![e]\!]S\} \\
&\Lambda^{\rightarrow}[\![\texttt{assert } b]\!]S \stackrel{\text{def}}{=} \mathcal{B}[\![b]\!]S \\
&\Lambda^{\rightarrow}[\![\texttt{if } b \texttt{ then } stmt \texttt{ else } stmt']\!]S \stackrel{\text{def}}{=} \\
&\quad \Lambda^{\rightarrow}[\![stmt]\!](\mathcal{B}[\![b]\!]S) \cup \Lambda^{\rightarrow}[\![stmt']\!](\mathcal{B}[\![\neg b]\!]S) \\
&\Lambda^{\rightarrow}[\![\texttt{while } b \texttt{ do } stmt \texttt{ done}]\!]S \stackrel{\text{def}}{=} \mathcal{B}[\![\neg b]\!](\text{lfp F}) \\
&\quad \text{F}(X) \stackrel{\text{def}}{=} S \cup \textsc{Incr}_{\texttt{nit}}(\Lambda^{\rightarrow}[\![stmt]\!](\mathcal{B}[\![b]\!]X)) \qquad (3) \\
&\Lambda^{\rightarrow}[\![stmt;\ stmt']\!]S \stackrel{\text{def}}{=} \Lambda^{\rightarrow}[\![stmt]\!](\Lambda^{\rightarrow}[\![stmt']\!]S)
\end{aligned}
$$

$$
\Lambda^{\rightarrow}[\![\texttt{entry } stmt]\!]S \stackrel{\text{def}}{=} \Lambda^{\rightarrow}[\![stmt]\!](S[\texttt{nit} \leftarrow 0]) \qquad (4)
$$

Fig. 3: Concrete forward reachability semantics $\Lambda^{\rightarrow}[\![\text{P}]\!]$.

the language is given by the following grammar:

$$
\begin{array}{rcll}
e & ::= & v \mid \texttt{x} \mid e + e \mid e - e & \text{(Arithmetic expressions)} \\
b & ::= & e \leq v \mid e = v \mid b \wedge b \mid \neg b & \text{(Boolean expressions)} \\
stmt & ::= & \texttt{skip} \mid \texttt{x} := e \mid \texttt{assert } b & \text{(Statements)} \\
& \mid & \texttt{if } b \texttt{ then } stmt \texttt{ else } stmt & \\
& \mid & \texttt{while } b \texttt{ do } stmt \texttt{ done} & \\
& \mid & stmt;\ stmt & \\
\text{P} & ::= & \texttt{entry } stmt & \text{(Programs)}
\end{array}
$$

where $v \in \mathbb{I}$ is a constant value, and $\texttt{x} \in \mathbb{V}$ is a variable. Statements can be defined recursively via the conditional, looping, or composition statement. The entry point of a program P is a statement. We introduce a variable $\texttt{nit} \notin \mathbb{V}$, called the *global loop counter*, to count the global number of iterations through the program execution. The set Σ is a (potentially infinite) set of program states, i.e., maps from variables in $\mathbb{V}^+ \stackrel{\text{def}}{=} \mathbb{V} \cup \{\texttt{nit}\}$ to values $\mathbb{I}$. In the following, $\mathbb{I}_{\geq 0} \stackrel{\text{def}}{=} \{n \in \mathbb{I} \mid n \geq 0\}$ denotes the set of non-negative values, and $\mathbb{I}^{\infty}$ the set of values extended with the symbol ∞.

The *semantics* of a program is a mathematical characterization of its behavior for all possible input data. In this work, we are interested in the impact of input variables on the global number of iterations at the end of the program computation. Thus, our concrete semantics for a program P is the *dependency semantics* $\Lambda^{\rightsquigarrow}_{\text{P}} \in \wp(\Sigma \times \Sigma)$ which captures the dependencies between states before and after the execution of a program. We write $\Lambda^{\rightsquigarrow}_{\text{P}}$ to denote the dependency semantics of a particular program P, the same applies for other semantics defined in this work. The semantics $\Lambda^{\rightsquigarrow}_{\text{P}}$ is defined as:

$$
\Lambda^{\rightsquigarrow}_{\text{P}} \stackrel{\text{def}}{=} \{\langle s, s' \rangle \mid s \in \Sigma \wedge s' \in \Lambda^{\rightarrow}[\![\text{P}]\!]\{s\}\} \qquad (5)
$$

Table 2: Dependency semantics of program `Add`. The global number of iterations is highlighted in blue on the right.

$\Lambda^{\rightarrow}_{\texttt{Add}}$

$\Lambda^{\rightarrow}[\![\texttt{Add}]\!](\langle$ p,	m,	n,	...,	nit $\rangle)$	=	$\langle$ p,	m,	n,	...,	nit $\rangle$
0	0	0		$*$		0	0	0		0
1	0	0		$*$		1	0	0		1
2	0	0		$*$		2	0	0		2
0	1	0		$*$		0	1	0		0
1	1	0		$*$		1	1	0		1
2	1	0		$*$		2	1	0		2
0	0	1		$*$		0	0	1		0
1	0	1		$*$		1	0	1		1
2	0	1		$*$		2	0	1		2
⋮	⋮	⋮		⋮		⋮	⋮	⋮		⋮
a	b	c		$*$		a	b	c		a

where $\Lambda^{\rightarrow}[\![\mathrm{P}]\!] \in \wp(\Sigma) \rightarrow \wp(\Sigma)$ is the standard *reachability semantics* instrumented to collect the global loop counter in the variable `nit`. Formally, $\Lambda^{\rightarrow}[\![\mathrm{P}]\!]$ is defined inductively on the syntax of our language in Figure 3. The semantics for arithmetic $\mathcal{A}[\![e]\!] \in \wp(\Sigma) \rightarrow \wp(\mathbb{I})$ and boolean expressions $\mathcal{B}[\![b]\!] \in \wp(\Sigma) \rightarrow \wp(\Sigma)$ are the standard ones and therefore omitted. To update the value of a variable $\mathtt{x} \in \mathbb{V}^+$ in a state $s \in \Sigma$ with the value $v \in \mathbb{I}$, we write $s[\mathtt{x} \leftarrow v]$. Hence, $s[\mathtt{x} \leftarrow v](\mathtt{x}) = v$ holds in the updated state. We lift the update operation to sets of states, i.e., $S[\mathtt{x} \leftarrow v] \stackrel{\text{def}}{=} \{s[\mathtt{x} \leftarrow v] \mid s \in S\}$ where $S \in \wp(\Sigma)$.

The reachability semantics $\Lambda^{\rightarrow}[\![\mathrm{P}]\!]$ collects the global number of iterations in the counter variable `nit`, semantically incremented after each loop iteration (cf. Equation (3)) via the transformer $\textsc{Incr}_{\texttt{nit}}(S) \stackrel{\text{def}}{=} \{s[\texttt{nit} \leftarrow s(\texttt{nit}) + 1] \mid s \in S\}$. At the beginning (cf. Equation (4)), we initialize the global loop counter to 0. The rest of the semantics is standard. Note that, our semantics consider only terminating traces as non-terminating traces yield an infinite number of loop iterations. We discuss the handling of non-terminating traces in Section 8.

Example 1. The dependency semantics of program `Add` in Figure 1, $\Lambda^{\rightarrow}_{\texttt{Add}}$, is defined in Table 2. For brevity, we omit the values of states that are not relevant. As noted in Section 2, the global loop counter `nit` at the end of the computation (on the right of Table 2) is equal to the initial value of the input variable `p` (on the left of Table 2). Instead, variations in the value of the input variables `m` and `n`, or any other variable, do not affect the value of the global loop counter.

4.2 Quantitative Input Data Usage

Building on the dependency semantics, we define our property of interest – quantitative input data usage for global loop bounds. We define a property by its extension, that is, the set of elements that manifest such property [10]. We consider

properties of programs, with dependency semantics in $\wp(\Sigma \times \Sigma)$, which are sets of sets of dependencies in $\wp(\wp(\Sigma \times \Sigma))$. The strongest property of the dependency semantics $\Lambda_{\mathrm{P}}^{\rightsquigarrow}$ is the standard *collecting semantics* $\Lambda_{\mathrm{P}}^{\mathbf{C}} \in \wp(\wp(\Sigma \times \Sigma))$, defined as $\Lambda_{\mathrm{P}}^{\mathbf{C}} \stackrel{\text{def}}{=} \{\ \Lambda_{\mathrm{P}}^{\rightsquigarrow}\ \}$, which is satisfied only and exactly by $\Lambda_{\mathrm{P}}^{\rightsquigarrow}$. Therefore, a program P satisfies a given property $F \in \wp(\wp(\Sigma \times \Sigma))$, written $\mathrm{P} \models F$, if and only if the dependency semantics $\Lambda_{\mathrm{P}}^{\rightsquigarrow}$ belongs to F, or equivalently, its collecting semantics $\Lambda_{\mathrm{P}}^{\mathbf{C}}$ is a subset of F, formally:

$$\mathrm{P} \models F \ \Leftrightarrow\ \Lambda_{\mathrm{P}}^{\mathbf{C}} \subseteq F$$

Our goal is to quantify the impact of a specific input variable on the global number of loop iterations of a program. To this end, we employ the framework proposed by Mazzucato et al. [29] for the verification of quantitative input data usage. Such framework employs an impact function $\textsc{Impact}_{\mathtt{i}} \in \wp(\Sigma \times \Sigma) \to \mathbb{I}_{\geq 0}^{\infty}$, which maps program semantics to a non-negative domain of quantities, where $\mathtt{i} \in \Delta$ denotes the current input variable of interest of the program under analysis. The *k-bounded impact property* $\mathscr{B}_{\mathtt{i}}^{\leq k} \in \wp(\wp(\Sigma \times \Sigma))$ is defined as the set of dependency semantics $\Lambda_{\mathrm{P}}^{\rightsquigarrow}$ whose impact of the input variable $\mathtt{i}$ is bounded by $k \in \mathbb{I}_{\geq 0}^{\infty}$. Formally,

$$\mathscr{B}_{\mathtt{i}}^{\leq k} \ \stackrel{\text{def}}{=}\ \{\Lambda_{\mathrm{P}}^{\rightsquigarrow} \in \wp(\Sigma \times \Sigma) \mid \textsc{Impact}_{\mathtt{i}}(\Lambda_{\mathrm{P}}^{\rightsquigarrow}) \leq k\} \tag{6}$$

The following theorem shows that our concrete semantics $\Lambda_{\mathrm{P}}^{\rightsquigarrow}$ is sound and complete for the verification of this k-bounded impact property.

Theorem 1 (Soundness and Completeness of $\Lambda_{\mathrm{P}}^{\mathbf{C}}$). *For any program* P*, the collecting semantics* $\Lambda_{\mathrm{P}}^{\mathbf{C}} \stackrel{\text{def}}{=} \{\Lambda_{\mathrm{P}}^{\rightsquigarrow}\}$ *is* sound and complete *for the verification of the k-bounded impact property* $\mathscr{B}_{\mathtt{i}}^{\leq k}$*. Formally:*

$$\mathrm{P} \models \mathscr{B}_{\mathtt{i}}^{\leq k} \ \Leftrightarrow\ \Lambda_{\mathrm{P}}^{\mathbf{C}} \subseteq \mathscr{B}_{\mathtt{i}}^{\leq k} \tag{7}$$

We require the impact function $\textsc{Impact}_{\mathtt{i}}$ to be monotonic in the amount of dependencies, so that larger semantics generate higher impact quantities. Formally, for all $S, S' \in \wp(\Sigma \times \Sigma)$, it holds that

$$S \subseteq S' \ \Rightarrow\ \textsc{Impact}_{\mathtt{i}}(S) \leq \textsc{Impact}_{\mathtt{i}}(S') \tag{8}$$

By the monotonicity result, we note that the *k-bounded impact property* $\mathscr{B}_{\mathtt{i}}^{\leq k}$ is a subset-closed property Mazzucato et al. [29], formally:

$$S \subseteq S' \ \wedge\ S' \in \mathscr{B}_{\mathtt{i}}^{\leq k} \ \Rightarrow\ S \in \mathscr{B}_{\mathtt{i}}^{\leq k} \tag{9}$$

Equation (9) is significant because it allows the use of abstract interpretation-based state reachability analyses to verify the k-bounded impact property. In combination with Theorem 1, it holds that the program P satisfies the k-bounded impact property $\mathscr{B}_{\mathtt{i}}^{\leq k}$ whenever the property holds for an over-approximation of the dependency semantics $\Lambda_{\mathrm{P}}^{\rightsquigarrow}$.

From the impact definitions already introduced by Mazzucato et al. [29], we select $\textsc{Range}_{\mathtt{i}} \in \wp(\Sigma \times \Sigma) \to \mathbb{N}$ for its definition matching our intuition on what

we need to measure, and the fact that it can be efficiently computed by a linear programming encoding (see later in Section 5.1). The impact $\textsc{Range}_{\mathtt{i}}$ determines the length of the range of outcomes, in our instance, the global number of loop iterations, from all the possible variations in the input variable `i`. In the following, for any set of states $S \in \wp(\Sigma)$, we write $S|_K \in K \to \mathbb{I}$ for the states of S 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 $s, s' \in \Sigma|_K$ agree on the values of the variables in K, i.e., $s =_K s' \overset{\text{def}}{\Leftrightarrow} \forall \mathtt{x} \in K.\ s(\mathtt{x}) = s'(\mathtt{x})$. Formally, $\textsc{Range}$ is defined as follows:

$$\textsc{Range}_{\mathtt{i}}(S) \overset{\text{def}}{=} \sup_{s_0 \in \Sigma|_\Delta} \textsc{Length}(\{s'(\mathtt{nit}) \mid \langle s, s' \rangle \in S \wedge s =_{\Delta \setminus \{\mathtt{i}\}} s_0\}) \quad (10)$$

where $\textsc{Length}(X) \overset{\text{def}}{=} \sup X - \inf X$ if $X \neq \emptyset$, otherwise $\textsc{Length}(\emptyset) \overset{\text{def}}{=} 0$. The sup and inf are the supremum and infimum operators, respectively. From the definition above it is easy to note that $\textsc{Range}_{\mathtt{i}}(S)$ is monotonic in the amount of dependencies S.

Example 2. Consider the dependency semantics $\Lambda^{\rightsquigarrow}_{\mathtt{Add}}$ defined in Table 2. For brevity, we consider states as $\Sigma = \{\langle p, m, n \rangle \mid p, m, n \in [0, u]\}$, where p is the value of `p`, m of `m`, n of `n`, all ranging in the interval $[0, u]$. Assuming we are interested in the impact of the input variable `p`, $\textsc{Range}_{\mathtt{p}}(\Lambda^{\rightsquigarrow}_{\mathtt{Add}})$ considers all possible input values $s_0 \in \Sigma|_\Delta$, on the left of Table 2. We collect all the input-output dependencies $\langle s, s' \rangle \in \Lambda^{\rightsquigarrow}_{\mathtt{Add}}$ that agree with the input value s_0 on the variables `m` and `n`, i.e., $s =_{\{\mathtt{m},\mathtt{n}\}} s_0$. For instance, $s_0 = \langle 0, 0, 0 \rangle$ collects the dependencies that start from states in $\{\langle p, 0, 0 \rangle \mid 0 \leq p \leq u\}$. From this set of dependencies, we consider only the output values $s'(\mathtt{nit})$, on the right of Table 2. For instance, regarding the input $s_0 = \langle 0, 0, 0 \rangle$, we collect the output values $[0, u]$. Then, we apply the operator $\textsc{Length}([0, u]) = \sup [0, u] - \inf [0, u] = u$. For all the input values, the maximum value is taken; we obtain $\textsc{Range}_{\mathtt{p}}(\Lambda^{\rightsquigarrow}_{\mathtt{Add}}) = u$.

Let us analyze the impact of other input variables, e.g. the input variable `m`. By considering the input value $s_0 = \langle 0, 0, 0 \rangle$, we collect the dependencies that start from states in $\{\langle 0, m, 0 \rangle \mid m \in [0, u]\}$. As we notice from Table 2, the number of iterations starting from any of these states is always 0. Hence, $\textsc{Length}(\{0\}) = \sup \{0\} - \inf \{0\} = 0$. For all the input values, we obtain: $\textsc{Range}_{\mathtt{m}}(\Lambda^{\rightsquigarrow}_{\mathtt{Add}}) = \textsc{Range}_{\mathtt{n}}(\Lambda^{\rightsquigarrow}_{\mathtt{Add}}) = 0$. We conclude that:

$$\mathtt{Add} \models \mathscr{B}^{\leq u}_{\mathtt{p}}, \qquad \mathtt{Add} \models \mathscr{B}^{\leq 0}_{\mathtt{n}}, \text{ and} \qquad \mathtt{Add} \models \mathscr{B}^{\leq 0}_{\mathtt{m}}$$

As a consequence, we can infer that there exist two executions starting from a different value for the input variable `p` that differ in the global number of iterations by at most u. On the contrary, any variation in the input variables `m` and `n` does not affect the global number of iterations.

5 A Static Analysis for Global Loop Bounds

In this section, we present a sound computable static analysis to quantify an upper bound on the impact of input variables on the global loop counter. The

soundness of the approach leverages: (1) an abstract domain of conjunctions of linear constraints, (2) a sound global loop bound analysis to collect the dependencies of the loop counter `nit` from the input variables, and (3) a linear programming encoding as a sound implementation of the impact RANGE.

5.1 Conjunctions of Linear Constraints

We define the numerical abstract state domain used in the global loop bound analysis. In principle, our abstract domain could be any convex abstract domain subsumed by the polyhedra domain [11], such as the interval domain, octagon domain [30], or the polyhedra domain itself. The elements of the abstract domain are conjunctions of linear constraints of the form:

$$c_1 \cdot \mathtt{x}_1 \;+\; \ldots \;+\; c_n \cdot \mathtt{x}_n \;+\; c_{n+1} \;\geq\; 0$$

where $\mathtt{x}_j \in \mathbb{V}^+$ are variables and $c_j \in \mathbb{I}$ are constant values. For a better readability, we avoid writing the constant-variable multiplication term $c_i \cdot \mathtt{x}_i$ when $c_i = 0$; and we abuse the notation, e.g. for the constraint $\mathtt{x}_1 = \mathtt{x}_2$, to denote the conjunction of the two linear constraints $\mathtt{x}_1 - \mathtt{x}_2 \geq 0$ and $\mathtt{x}_2 - \mathtt{x}_1 \geq 0$. The abstract domain is a lattice $\langle \mathcal{D}^\sharp, \sqsubseteq^{\mathcal{D}^\sharp}, \sqcup^\sharp, \sqcap^\sharp, \top^\sharp, \bot^\sharp \rangle$ equipped with a concretization function $\gamma^{\mathcal{D}^\sharp} \in \mathcal{D}^\sharp \to \wp(\Sigma)$, defined as:

$$\begin{aligned}\gamma^{\mathcal{D}^\sharp}(d) \;\stackrel{\text{def}}{=}\; \{s \in \Sigma \mid \forall\, (c_1 \cdot \mathtt{x}_1 + \ldots + c_n \cdot \mathtt{x}_n + c_{n+1} \geq 0) \in d.\\ c_1 \cdot s(\mathtt{x}_1) + \ldots + c_n \cdot s(\mathtt{x}_n) + c_{n+1} \geq 0\}\end{aligned}$$

Additionally, in order to be effectively used in the context of the global loop bound analysis, we assume:

(i) an operator $\textsc{Subs}^\sharp[\![\mathtt{x} \leftarrow e]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ to substitute the variable $\mathtt{x} \in \mathbb{V}^+$ with the expression e,
(ii) an operator $\textsc{Filter}^\sharp[\![b]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ to handle boolean expressions b,
(iii) a widening operator $\nabla^\sharp \in \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ to ensure termination of the analysis, and
(iv) a project operator $\textsc{Proj}^\sharp_{\mathtt{x}} \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ to remove the input variable $\mathtt{x}$ from the given abstract state.

5.2 Global Loop Bounds

The global loop bound semantics $\Lambda^{\mathrm{g}}[\![\mathrm{P}]\!] \in \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is a backward abstract semantics that generates an invariant over the global loop counter `nit`. We define the *abstract dependency semantics* $\Lambda^*_{\mathrm{P}} \in \mathcal{D}^\sharp$ as the global loop bound semantics $\Lambda^{\mathrm{g}}[\![\mathrm{P}]\!]$ starting from the post-condition $\mathtt{nit} = 0$. During the backward analysis, the value of `nit` increases from 0 to an over-approximation of the possible global number of iterations. As a consequence, the pre-condition invariant generated by the abstract dependency semantics Λ^*_{P} over-approximates relations between `nit` and the initial values of variables in $\mathbb{V}$. Formally,

$$\Lambda^*_{\mathrm{P}} \;\stackrel{\text{def}}{=}\; \Lambda^{\mathrm{g}}[\![\mathrm{P}]\!](\mathtt{nit} = 0)$$

The concretization function $\gamma^* \in \mathcal{D}^\sharp \to \wp(\Sigma \times \Sigma)$ maps an abstract state to a set of input-output dependencies. Its goal is to preserve the relations between input values and the global loop counter `nit`. Potentially, $\gamma^*(\Lambda^*_{\mathrm{P}})$ introduces dependencies that are not present in the concrete dependency semantics $\Lambda^{\rightsquigarrow}_{\mathrm{P}}$. Nevertheless, these additional dependencies are irrelevant for the quantification of the impact. Formally:

$$\gamma^*(\Lambda^*_{\mathrm{P}}) \stackrel{\text{def}}{=} \cup \{\bigcup_{v \in \mathbb{I}} s[\texttt{nit} \leftarrow v] \times \bigcup_{\texttt{x} \in \mathbb{V}, v \in \mathbb{I}} s[\texttt{x} \leftarrow v] \mid s \in \gamma^{\mathcal{D}^\sharp}(\Lambda^*_{\mathrm{P}})\} \tag{11}$$

Equation (11) restores the relations between concrete input-output dependencies. The result of the abstract dependency semantics Λ^*_{P} is an abstract element relating variables' input values to the global loop counter `nit`. Specifically, the input states are $\bigcup_{v \in \mathbb{I}} s[\texttt{nit} \leftarrow v]$ where we reset `nit` to any possible value, as it is irrelevant for input states. The output states $\bigcup_{\texttt{x} \in \mathbb{V}, v \in \mathbb{I}} s[\texttt{x} \leftarrow v]$ preserve only the value of the global loop counter `nit` and ignore the other variables. Potentially, such operation may introduce dependencies that do not belong to the concrete dependency semantics $\Lambda^{\rightsquigarrow}_{\mathrm{P}}$. This is due to the fact that the abstract dependency semantics Λ^*_{P} looses relations among the variables of output states. However, the additional dependencies do not affect the value of the global loop counter `nit` and are irrelevant for the quantification of the impact.

Example 3. We consider the example in Figure 1, where states Σ are tuples $\langle p, m, n, nit \rangle$, respectively for the variables `p`, `m`, `n`, and the global loop counter `nit`. Let us assume that the computation of the abstract dependency semantics on the program `Add` results in:

$$\Lambda^*_{\texttt{Add}} \;=\; \Lambda^{\mathrm{g}}[\![\texttt{Add}]\!](\texttt{nit} = 0) \;=\; (\texttt{p} = \texttt{nit})$$

Then, the concretization of the abstract element $\texttt{p} = \texttt{nit}$ is:

$$\gamma^{\mathcal{D}^\sharp}(\texttt{p} = \texttt{nit}) \;=\; \{\langle p, m, n, p \rangle \mid p, m, n \in \mathbb{I}\}$$

where $\langle p, m, n, p \rangle$ is the concrete state in which the input variable `p` (first p in the tuple) is equal to the loop counter `nit` (last p in the tuple). The goal of the concretization of $\gamma^*(\Lambda^*_{\texttt{Add}})$ is to over-approximate the dependency semantics $\Lambda^{\rightsquigarrow}_{\texttt{Add}}$. Additional dependencies may be introduced, but they are irrelevant for the quantification of the impact. For instance, consider the state $\langle 2, 0, 1, 2 \rangle$ from $\gamma^{\mathcal{D}^\sharp}(\texttt{p} = \texttt{nit})$, representing input-output dependencies from $\texttt{p} = 2, \texttt{m} = 0$, and $\texttt{n} = 1$ to an output state with 2 iterations. We concretize the following dependencies: $\{\langle 2, 0, 1, v \rangle \mid v \in \mathbb{I}\} \times \{\langle v, v', v'', 2 \rangle \mid v, v', v'' \in \mathbb{I}\}$.

From any state $\langle p, m, n, p \rangle$ we obtain the left part of Table 2, cf. the input states, by resetting the value global loop counter `nit`:

$$\bigcup_{v \in \mathbb{I}} \langle p, m, n, p \rangle[\texttt{nit} \leftarrow v] \;=\; \{\langle p, m, n, v \rangle \mid v \in \mathbb{I}\}$$

Table 3: Concretization of the abstract dependency semantics of program `Add`. The global number of iterations is highlighted in blue on the right.

$\gamma^{\mathcal{D}^\natural}(\Lambda^*_{\texttt{Add}})$ $\quad$ $\gamma^{\mathcal{D}^\natural}(\Lambda^*_{\texttt{Add}})$

$\bigcup_{v\in\mathbb{I}} \langle p,m,n,p\rangle[\texttt{nit} \leftarrow v]$ $\quad$ $\bigcup_{\texttt{x}\in\mathbb{V},v\in\mathbb{I}} \langle p,m,n,p\rangle[\texttt{x} \leftarrow v]$

$\gamma^*(\Lambda^*_{\texttt{Add}})\{$

$\{\langle\ p,$	$m,$	$n,$	$v\ \rangle \mid v \in \mathbb{I}\}$	$\times$	$\{\langle\ v,$	$v',$	$v'',$	$\mid p\ \rangle \mid v, v'v'' \in \mathbb{I}\}$
0	0	0	$*$		$*$	$*$	$*$	0
1	0	0	$*$		$*$	$*$	$*$	1
2	0	0	$*$		$*$	$*$	$*$	2
0	1	0	$*$		$*$	$*$	$*$	0
1	1	0	$*$		$*$	$*$	$*$	1
2	1	0	$*$		$*$	$*$	$*$	2
0	0	1	$*$		$*$	$*$	$*$	0
1	0	1	$*$		$*$	$*$	$*$	1
2	0	1	$*$		$*$	$*$	$*$	2
$\vdots$	$\vdots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$	$\vdots$	$\vdots$

Regarding the right part of Table 2, cf. the output states, we are only interested in the value of the global loop counter `nit`. We reset the value of all the other variables, obtaining:

$$\bigcup_{\texttt{x}\in\mathbb{V},v\in\mathbb{I}} \langle p,m,n,p\rangle[\texttt{x} \leftarrow v] \;=\; \{\langle v,v',v'',p\rangle \mid v,v',v'' \in \mathbb{I}\}$$

The modifier $\texttt{x} \in \mathbb{V}$ collects all the variables but the global loop counter `nit`. Table 3 shows the concretization of the abstract dependency semantics, which over-approximates the dependency semantics $\Lambda^{\rightsquigarrow}_{\texttt{Add}}$ of Table 2.

For the abstract dependency semantics Λ^*_{P} to be sound, we require the concretization γ^* to over-approximate the dependency semantics $\Lambda^{\rightsquigarrow}_{\mathrm{P}}$, formally:

$$\Lambda^{\rightsquigarrow}_{\mathrm{P}} \subseteq \gamma^*(\Lambda^*_{\mathrm{P}}) \tag{12}$$

The soundness condition, cf. Equation (12), is of significant importance as it allows any sound global loop bound analysis $\Lambda^{\mathrm{g}}[\![\mathrm{P}]\!]$ to verify the k-bounded impact property $\mathscr{B}^{\leq k}_{\mathbb{I}}$. A possible candidate semantics for the global loop bound analysis $\Lambda^{\mathrm{g}}[\![\mathrm{P}]\!] \in \mathcal{D}^\natural \to \mathcal{D}^\natural$ is defined in Figure 4. The semantics $\Lambda^{\mathrm{g}}[\![\mathrm{P}]\!]$ is a *backward co-reachability semantics* instrumented to increment the loop counter `nit` after each loop iteration. The loop counter `nit` is handled semantically in the abstract domain without loss of precision. The rest of the semantics is classical. The following result states the soundness of the backward semantics defined in Figure 4.

$$
\begin{aligned}
&\Lambda^{\mathrm{g}}[\![\texttt{skip}]\!]d \stackrel{\text{def}}{=} d\\
&\Lambda^{\mathrm{g}}[\![\texttt{x} := e]\!]d \stackrel{\text{def}}{=} \textsc{Subs}^{\natural}[\![\texttt{x} \leftarrow e]\!]d\\
&\Lambda^{\mathrm{g}}[\![\texttt{assert } b]\!]d \stackrel{\text{def}}{=} \textsc{Filter}^{\natural}[\![b]\!]d\\
&\Lambda^{\mathrm{g}}[\![\texttt{if } b \texttt{ then } stmt \texttt{ else } stmt']\!]d \stackrel{\text{def}}{=}\\
&\quad \textsc{Filter}^{\natural}[\![b]\!](\Lambda^{\mathrm{g}}[\![stmt]\!]d) \sqcup^{\natural} \textsc{Filter}^{\natural}[\![\neg b]\!](\Lambda^{\mathrm{g}}[\![stmt']\!]d)\\
&\Lambda^{\mathrm{g}}[\![\texttt{while } b \texttt{ do } stmt \texttt{ done}]\!]d \stackrel{\text{def}}{=} \lim_{n} F^{\natural}_{n}\\
&\quad F^{\natural}_{0} \stackrel{\text{def}}{=} d\\
&\quad F^{\natural}_{n+1} \stackrel{\text{def}}{=} F^{\natural}_{n} \nabla^{\natural} F^{\natural}(F^{\natural}_{n})\\
&\quad F^{\natural}(a) \stackrel{\text{def}}{=} \textsc{Filter}^{\natural}[\![\neg b]\!]d \sqcup^{\natural} \textsc{Filter}^{\natural}[\![b]\!](\Lambda^{\mathrm{g}}[\![stmt]\!](\textsc{Subs}^{\natural}[\![\texttt{nit} \leftarrow \texttt{nit} - 1]\!]a))\\
&\Lambda^{\mathrm{g}}[\![stmt;\ stmt']\!]d \stackrel{\text{def}}{=} \Lambda^{\mathrm{g}}[\![stmt]\!](\Lambda^{\mathrm{g}}[\![stmt']\!]d)\\
\\
&\Lambda^{\mathrm{g}}[\![\texttt{entry } stmt]\!]d \stackrel{\text{def}}{=} \Lambda^{\mathrm{g}}[\![stmt]\!]d
\end{aligned}
$$

Fig. 4: Abstract backward semantics $\Lambda^{\mathrm{g}}[\![\mathrm{P}]\!]$ for global loop bound analysis.

Theorem 2 (Soundness of $\Lambda^{\mathrm{g}}[\![\mathbf{P}]\!]$). *The semantics $\Lambda^{\mathrm{g}}[\![\mathrm{P}]\!] \in \mathcal{D}^{\natural} \to \mathcal{D}^{\natural}$ defined in Figure 4 is a* sound over-approximation *of the dependency semantics $\Lambda^{\rightsquigarrow}_{\mathrm{P}}$:*

$$\Lambda^{\rightsquigarrow}_{\mathrm{P}} \ \subseteq\ \gamma^{*}(\Lambda^{*}_{\mathrm{P}}) \ =\ \gamma^{*}(\Lambda^{\mathrm{g}}[\![\mathrm{P}]\!](\texttt{nit} = 0))$$

Proof (Sketch). The dependency semantics $\Lambda^{\rightsquigarrow}_{\mathrm{P}}$ of Equation (5) *is defined as the set of input-output dependencies employing the concrete forward reachability semantics $\Lambda^{\rightarrow}[\![\mathrm{P}]\!]$. Equivalently, on Figure 5 we define a concrete backward reachability semantics $\Lambda^{\leftarrow}[\![\mathrm{P}]\!]$ to compute the dependency semantics $\Lambda^{\leftsquigarrow}_{\mathrm{P}}$. The backward semantics $\Lambda^{\leftarrow}[\![\mathrm{P}]\!]$ is the concrete reachability semantics starting from* $\texttt{nit} = 0$ *and decrementing backwardly the loop counter* nit *after each loop iteration. The backward decrement is defined as* $\textsc{Decr}^{\leftarrow}_{\texttt{nit}}(S) \stackrel{\text{def}}{=} \{s \in \Sigma \mid s[\texttt{nit} \leftarrow s(\texttt{nit}) - 1] \in S\}$. *Employing the backward semantics, we define the dependency semantics $\Lambda^{\leftsquigarrow}_{\mathrm{P}}$ as $\Lambda^{\leftsquigarrow}_{\mathrm{P}} \stackrel{\text{def}}{=} \{\langle s, s'\rangle \mid s' \in \Sigma \wedge s \in \Lambda^{\leftarrow}[\![\mathrm{P}]\!]\{s\}\}$. It is easy to prove by induction on the syntax of the program that the global loop bound semantics $\Lambda^{\mathrm{g}}[\![\mathrm{P}]\!]$ of Figure 4 is an over-approximation of the concrete backward reachability semantics $\Lambda^{\leftarrow}[\![\mathrm{P}]\!]$. As a consequence, it holds that the abstract dependency semantics is an over-approximation of the backward concrete dependency semantics, i.e., $\Lambda^{\leftsquigarrow}_{\mathrm{P}} \subseteq \gamma^{*}(\Lambda^{*}_{\mathrm{P}})$. As both forward and backward reachability semantics are concrete semantics equipped with the global loop counter* nit*, they equivalently represent the same set of input-output dependencies, i.e., $\Lambda^{\rightsquigarrow}_{\mathrm{P}} = \Lambda^{\leftsquigarrow}_{\mathrm{P}}$. By transitivity, we conclude that: $\Lambda^{\rightsquigarrow}_{\mathrm{P}} = \Lambda^{\leftsquigarrow}_{\mathrm{P}} \subseteq \gamma^{*}(\Lambda^{\mathrm{g}}[\![\mathrm{P}]\!](\texttt{nit} = 0))$.* □

$$
\begin{aligned}
&\Lambda^{\leftarrow}[\![\texttt{skip}]\!]S \stackrel{\text{def}}{=} S\\
&\Lambda^{\leftarrow}[\![\texttt{x} := e]\!]S \stackrel{\text{def}}{=} \{s \in \Sigma \mid v \in \mathcal{A}[\![e]\!] \wedge s[\texttt{x} \leftarrow v] \in S\}\\
&\Lambda^{\leftarrow}[\![\texttt{assert } b]\!]S \stackrel{\text{def}}{=} \mathcal{B}[\![b]\!]S\\
&\Lambda^{\leftarrow}[\![\texttt{if } b \texttt{ then } stmt \texttt{ else } stmt']\!]S \stackrel{\text{def}}{=}\\
&\quad \mathcal{B}[\![b]\!](\Lambda^{\leftarrow}[\![stmt]\!]S) \cup \mathcal{B}[\![\neg b]\!](\Lambda^{\leftarrow}[\![stmt']\!]S)\\
&\Lambda^{\leftarrow}[\![\texttt{while } b \texttt{ do } stmt \texttt{ done}]\!]S \stackrel{\text{def}}{=} \text{lfp F}\\
&\quad \mathrm{F}(X) \stackrel{\text{def}}{=} \mathcal{B}[\![\neg b]\!]S \cup \mathcal{B}[\![b]\!](\Lambda^{\leftarrow}[\![stmt]\!]\textsc{Decr}^{\leftarrow}_{\texttt{nit}}(X))\\
&\Lambda^{\leftarrow}[\![stmt;\ stmt']\!]S \stackrel{\text{def}}{=} \Lambda^{\leftarrow}[\![stmt]\!](\Lambda^{\leftarrow}[\![stmt']\!]S)\\
&\Lambda^{\leftarrow}[\![\texttt{entry } stmt]\!]S \stackrel{\text{def}}{=} \Lambda^{\leftarrow}[\![stmt]\!](S[\texttt{nit} \leftarrow 0])
\end{aligned}
$$

Fig. 5: Concrete backward reachability semantics $\Lambda^{\leftarrow}[\![\mathrm{P}]\!]$

5.3 Linear Programming Encoding

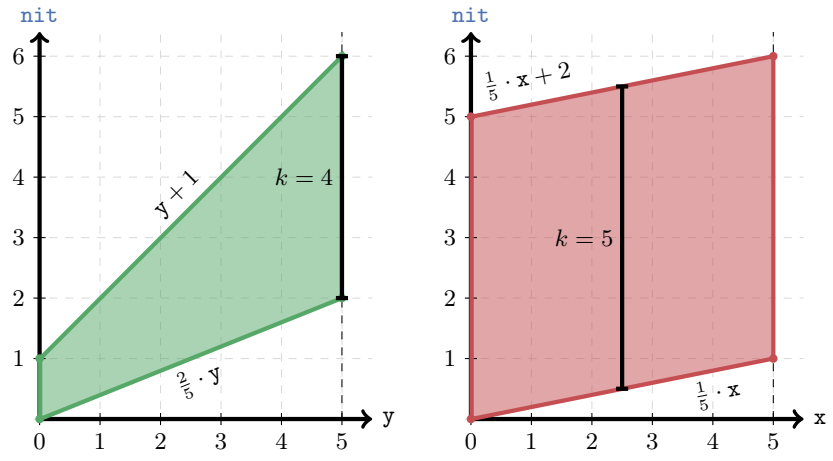
Finally, we present a linear programming encoding $\texttt{Range}^{\natural}_{\texttt{i}} \in \mathcal{D}^{\natural} \to \mathbb{I}^{\infty}_{\geq 0}$ to compute the abstract implementation of the impact $\textsc{Range}_{\texttt{i}}$. The problem is defined as:

$$
\begin{aligned}
\texttt{Range}^{\natural}_{\texttt{i}}(d) \;=\; &\text{maximize } k && (13)\\
&\text{subject to } \textsc{Proj}^{\natural}_{\texttt{i}}(\textsc{Subs}^{\natural}[\![\texttt{nit} \leftarrow \overline{\texttt{nit}}]\!](d)) && (14)\\
&\wedge\ \textsc{Proj}^{\natural}_{\texttt{i}}(\textsc{Subs}^{\natural}[\![\texttt{nit} \leftarrow \underline{\texttt{nit}}]\!](d)) && (15)\\
&\wedge\ 0 \leq k \leq \overline{\texttt{nit}} - \underline{\texttt{nit}} && (16)
\end{aligned}
$$

where $\overline{\texttt{nit}}, \underline{\texttt{nit}}$ are fresh variables. Since k should be an integer variable, we specifically solve a mixed-integer linear programming problem. As seen in the overview, Equation (14) substitutes the variable nit with $\overline{\texttt{nit}}$ to account for the maximal value of the global loop counter nit. Then, it projects away the input variable i to encompass any possible variation of that variable. Equation (15) substitutes the variable nit with $\underline{\texttt{nit}}$ for the minimal value of nit, and again projects away the input variable i. Hence, the set of constraints from Equation (14) and Equation (15) only differ in the variable of the loop counter, respectively $\overline{\texttt{nit}}$ and $\underline{\texttt{nit}}$. Finally, the objective function, Equation (13), maximizes the value of the bound k, which ranges between 0 and $\overline{\texttt{nit}} - \underline{\texttt{nit}}$, cf. Equation (16). The maximum value of the bound k is the length of the range of the feasible values for the loop counter nit.

Example 4. We consider again the example of the program Add in Figure 1. Let us assume that $\texttt{p} \in [0, u]$ and the computation of the abstract dependency semantics on the program Add results in:

$$\Lambda^{*}_{\texttt{Add}} \;=\; (\texttt{p} = \texttt{nit} \;\wedge\; 0 \leq \texttt{p} \leq u)$$



(a) $\frac{2}{5} \cdot \mathtt{y} \leq \mathtt{nit} \leq \mathtt{y} + 1$ (b) $\frac{1}{5} \cdot \mathtt{x} \leq \mathtt{nit} \leq \frac{1}{5} \cdot \mathtt{x} + 2$

Fig. 6: Graphical representation of the feasible regions for the two linear programming problems of Example 5.

To compute the abstract range $\mathtt{Range}^\natural_\mathtt{p}$ for the input variable $\mathtt{p}$, we solve the linear programming problem Equation (13)–Equation (16). Where Equation (14) and Equation (15) are respectively:

$$\begin{aligned}
&\textsc{Proj}^\natural_\mathtt{p}(\textsc{Subs}^\natural[\![\mathtt{nit} \leftarrow \overline{\mathtt{nit}}]\!](\mathtt{p} = \mathtt{nit} \ \wedge \ 0 \leq \mathtt{p} \leq u)) \\
&\quad = \ \textsc{Proj}^\natural_\mathtt{p}(\mathtt{p} = \overline{\mathtt{nit}} \ \wedge \ 0 \leq \mathtt{p} \leq u) \ = \ 0 \leq \overline{\mathtt{nit}} \leq u \\
&\textsc{Proj}^\natural_\mathtt{p}(\textsc{Subs}^\natural[\![\mathtt{nit} \leftarrow \underline{\mathtt{nit}}]\!](\mathtt{p} = \mathtt{nit} \ \wedge \ 0 \leq \mathtt{p} \leq u)) \\
&\quad = \ \textsc{Proj}^\natural_\mathtt{p}(\mathtt{p} = \underline{\mathtt{nit}} \ \wedge \ 0 \leq \mathtt{p} \leq u) \ = \ 0 \leq \underline{\mathtt{nit}} \leq u
\end{aligned}$$

Therefore, the linear programming encoding for $\mathtt{Range}^\natural_\mathtt{p}(\mathtt{p} = \mathtt{nit} \ \wedge \ 0 \leq \mathtt{p} \leq u)$ is defined as:

$$\begin{aligned}
\text{maximize} \quad & k \\
\text{subject to} \quad & 0 \leq \overline{\mathtt{nit}} \leq u \\
\wedge \quad & 0 \leq \underline{\mathtt{nit}} \leq u \\
\wedge \quad & 0 \leq k \leq \overline{\mathtt{nit}} - \underline{\mathtt{nit}}
\end{aligned}$$

which maximizes at u. On the other hand, projecting away the other input variables leaves the invariant $\mathtt{p} = \mathtt{nit} \wedge 0 \leq \mathtt{p} \leq u$ unchanged. Thus, k maximizes at 0 as the variable $\mathtt{p}$ is equal to both $\overline{\mathtt{nit}}$ and $\underline{\mathtt{nit}}$. Interestingly, we did not lose any precision regarding the k-bounded impact property as $\textsc{Range}_\mathtt{i}(\Lambda^{\rightsquigarrow}_{\mathtt{Add}}) = \mathtt{Range}^\natural_\mathtt{i}(\Lambda^*_{\mathtt{Add}})$ for any input variable $\mathtt{i}$ in the program $\mathtt{Add}$.

Example 5. We consider a more complex example to demonstrate the capabilities of the linear programming encoding, in a scenario where the invariant computed by the global loop bound analysis does not already reveal the impact of

input variables. Let us assume that the computation of the abstract dependency semantics for a program P results in:

$$\Lambda^*_{\mathrm{P}} \;=\; \left(\frac{2}{5}\cdot \mathtt{y} + \frac{1}{5}\cdot \mathtt{x} \;\leq\; \mathtt{nit} \;\leq\; \mathtt{y} + \frac{1}{5}\cdot \mathtt{x} \;\wedge\; \mathtt{x},\mathtt{y} \in [0,5]\right)$$

To compute the abstract range $\mathtt{Range}^\natural_{\mathtt{x}}$ for the input variable $\mathtt{x}$, we first project away the variable $\mathtt{x}$ from the abstract state, obtaining:

$$\frac{2}{5}\cdot \mathtt{y} \;\leq\; \mathtt{nit} \;\leq\; \mathtt{y} + 1 \;\wedge\; \mathtt{y} \in [0,5]$$

Then, we solve the linear programming problem, where we substitute the variable $\mathtt{nit}$ with $\overline{\mathtt{nit}}$ and $\underline{\mathtt{nit}}$ to respectively maximize and minimize the global loop counter $\mathtt{nit}$:

$$\begin{aligned}
&\text{maximize} && k \\
&\text{subject to} && \frac{2}{5}\cdot \mathtt{y} \;\leq\; \overline{\mathtt{nit}} \;\leq\; \mathtt{y} + 1 \;\wedge\; \mathtt{y} \in [0,5] \\
&\wedge && \frac{2}{5}\cdot \mathtt{y} \;\leq\; \underline{\mathtt{nit}} \;\leq\; \mathtt{y} + 1 \;\wedge\; \mathtt{y} \in [0,5] \\
&\wedge && 0 \;\leq\; k \;\leq\; \overline{\mathtt{nit}} - \underline{\mathtt{nit}}
\end{aligned}$$

The linear programming encoding maximizes at $k = 4$ where $\mathtt{y} = 5$, Figure 6a shows graphically the feasible region and maximization point. On the other hand, to compute the abstract range $\mathtt{Range}^\natural_{\mathtt{y}}$ for the input variable $\mathtt{y}$, we solve linear programming encoding where we project away the variable $\mathtt{y}$ from the abstract state, obtaining:

$$\begin{aligned}
&\text{maximize} && k \\
&\text{subject to} && \frac{1}{5}\cdot \mathtt{x} \;\leq\; \overline{\mathtt{nit}} \;\leq\; \frac{1}{5}\cdot \mathtt{x} + 2 \;\wedge\; \mathtt{x} \in [0,5] \\
&\wedge && \frac{1}{5}\cdot \mathtt{x} \;\leq\; \underline{\mathtt{nit}} \;\leq\; \frac{1}{5}\cdot \mathtt{x} + 2 \;\wedge\; \mathtt{x} \in [0,5] \\
&\wedge && 0 \;\leq\; k \;\leq\; \overline{\mathtt{nit}} - \underline{\mathtt{nit}}
\end{aligned}$$

This second linear programming problem maximizes at $k = 5$ for any value of $\mathtt{x}$, Figure 6b shows graphically the feasible region and maximization point for this second programming problem. As a result, we obtain $\mathtt{Range}^\natural_{\mathtt{x}}(\Lambda^*_{\mathrm{P}}) = 4$ and $\mathtt{Range}^\natural_{\mathtt{y}}(\Lambda^*_{\mathrm{P}}) = 5$, showing that the variable $\mathtt{y}$ has a higher impact on the global loop counter $\mathtt{nit}$ than the variable $\mathtt{x}$.

The next result shows that $\mathtt{Range}^\natural_{\mathtt{i}}$ is a sound implementation of the impact $\textsc{Range}_{\mathtt{i}}$, by means of Definition 3 in Mazzucato et al. [29]. That is, the abstract quantity is always higher than the concrete counterpart.

Lemma 1 (Sound Implementation of Range). *For any program* P *the following holds:*

$$\textsc{Range}_{\mathtt{i}}(\Lambda^{\rightsquigarrow}_{\mathrm{P}}) \;\leq\; \mathtt{Range}^\natural_{\mathtt{i}}(\Lambda^*_{\mathrm{P}})$$

Proof. We show that $\textsc{Range}_{\mathtt{i}}(\gamma^*(\Lambda^*_{\mathrm{P}})) \leq \mathtt{Range}^{\natural}_{\mathtt{i}}(\Lambda^*_{\mathrm{P}})$ *by contradiction. Let us assume that* $\textsc{Range}_{\mathtt{i}}(\gamma^*(\Lambda^*_{\mathrm{P}})) > \mathtt{Range}^{\natural}_{\mathtt{i}}(\Lambda^*_{\mathrm{P}})$ *and* $\mathtt{Range}^{\natural}_{\mathtt{i}}(\Lambda^*_{\mathrm{P}}) = k$*. Then, by definition of* $\textsc{Range}_{\mathtt{i}}$*, cf. Equation* (10)*, there exists two concrete states, differing only in the value of* i*, such that the difference in the value of the global loop counter* nit *is greater than* k*. As a consequence, in the abstract implementation* $\mathtt{Range}^{\natural}_{\mathtt{i}}$*, the distance between* $\overline{\mathtt{nit}}$ *and* nit *should maximize at a value greater than* k*. This contradicts the assumption that* $k = \mathtt{Range}^{\natural}_{\mathtt{i}}(\Lambda^*_{\mathrm{P}})$*. From Theorem 2 and the fact that* $\textsc{Range}_{\mathtt{i}}$ *is monotonic, cf. Equation* (8)*, we conclude that:* $\textsc{Range}_{\mathtt{i}}(\Lambda^{\rightsquigarrow}_{\mathrm{P}}) \leq \textsc{Range}_{\mathtt{i}}(\gamma^*(\Lambda^*_{\mathrm{P}})) \leq \mathtt{Range}^{\natural}_{\mathtt{i}}(\Lambda^*_{\mathrm{P}})$. □

Thus, by the application of the quantitative framework, it holds that our static analysis is sound when employed to verify the property of interest $\mathscr{B}^{\leq k}_{\mathtt{i}}$ for the program P.

Theorem 3 (Soundness of $\mathtt{Range}^{\natural}_{\mathtt{i}}$**).** *Given a program* P *and an input variable* $\mathtt{i} \in \Delta$*, the following holds:*

$$\mathtt{Range}^{\natural}_{\mathtt{i}}(\Lambda^*_{\mathrm{P}}) \leq k \;\Rightarrow\; \mathrm{P} \models \mathscr{B}^{\leq k}_{\mathtt{i}}$$

Proof. From the fact that the abstract semantics Λ^*_{P} *is sound with respect to the dependency semantics* $\Lambda^{\rightsquigarrow}_{\mathrm{P}}$*, cf. Theorem 2, and that* $\mathtt{Range}^{\natural}_{\mathtt{i}}$ *is a sound implementation of the impact* $\textsc{Range}_{\mathtt{i}}$*, cf. Lemma 1, we obtain that* $\textsc{Range}_{\mathtt{i}}(\Lambda^{\rightsquigarrow}_{\mathrm{P}}) \leq k$*. Hence, it means that the collecting semantics* $\Lambda^{\mathbf{C}}_{\mathrm{P}}$ *is a subset of the property* $\mathscr{B}^{\leq k}_{\mathtt{i}}$*. By Equation* (7) *we conclude that* $\mathrm{P} \models \mathscr{B}^{\leq k}_{\mathtt{i}}$*.* □

6 Implementation

To support our claims, we implemented a tool, called TimeSec, in about 3000 lines of Python code[2]. The global loop bound analysis is based on the numerical library Apron [20], we instrumented the analysis with widening and narrowing operators after 2 fixpoint iterations. For the linear programming encoding, we used the Python library SciPy[3]. An artifact of TimeSec, including the source code, the benchmarks, and the evaluation results, is available in Zenodo[4]. In this section, we discuss some implementation features that make the analysis scalable, precise, and able to handle real-world programs.

6.1 Syntactic Dependency Analysis

As introduced in the overview, the global loop bound analysis can exploit the syntactic dependency analysis proposed by Urban and Müller [43, Section 10]. The syntactic dependency analysis is used to determine an over-approximation of the set of relevant variables for the global loop counter, for each program location. We employ such information to do program slicing. As a consequence, we reduce

[2] The tool will be available as open-source software in GitHub upon acceptance.
[3] `https://scipy.org`
[4] `https://doi.org/10.5281/zenodo.11127583`

the number of variables in the underlying abstract domain, and avoid analyzing irrelevant statements. The program under analysis can be evaluated even in presence of statements and expressions that are hard to handle, e.g. bitwise operations, array manipulation, and function calls to name a few, as long as they are not relevant. Indeed, excluding irrelevant statements from the analysis does not affect the global loop bounds.

It is worth noting that, this syntactic dependency analysis is already an input data usage analysis, as it already determines which variables have zero impact on the global loop counter, and which have a non-zero impact. Indeed, the syntactic dependency analysis is a qualitative counterpart to our quantitative definition.

Example 6. Regarding the example of Figure 1, the dependency analysis is able to discover that most of the variables are irrelevant for the global loop bound. Therefore, we are able to exclude most of the statements, including the bitwise operations regarding the remainder (e.g. Line 14), the array manipulation (e.g. Line 10), the array indices (e.g. Line 10), the conditional for the padding at the end (cf. Line 51). Overall, we excluded 33 from the original 52 lines of code (about the 60%), and the analysis was able to handle the program with ease without any specific handling for the excluded statements. Regarding the amount of variables, we reduced the number of variables from 13 to 7 (without counting the global loop counter `nit`).

Additionally, even though our formal syntax does not include the use of arrays, real-world programs often do. To remain sound, the syntactic dependency analysis can be extended to handle arrays, by considering a conservative points-to analysis to determine the shared memory locations. For our work, we consider a classical flow-insensitive points-to analysis [39] to determine an over-approximation of the memory locations shared by the variables at any program location. Whenever the analysis discovers that a variable is potentially used, all the variables that share the same memory location are considered relevant as well.

6.2 Optimizations

Many works in the literature propose to combine forward and backward phases to provide tighter invariants [17, 36, 44]. Therefore, in our tool we combine an initial forward reachability analysis to enhance both the syntactic dependency analysis and the global loop bound analysis. Furthermore, we employ a narrowing operator to refine the upper bound of the least fixpoint computed by the widening operator [10].

Example 7. Consider the first for-loop at Line 9 of the program `Add` in Figure 1. Here, we report it in the form of a while loop for simplicity:

```
1 assert r >= 0
2 while (r > 0):
3   r = r - 1
```

Without a forward pre-analysis, the backward analysis would infer that the global number of iterations is always greater or equal to the initial value of `r`. The missing information is that the value of `r` is always non-negative at the beginning of the loop. However, a forward pre-analysis could easily propagate such information to the backward analysis. As a consequence, the backward analysis would infer that the global number of iterations is always equal to the initial value of `r`. The main difference of the two approaches can be observed when the result of the backward analysis is used to verify the global loop bound property. With the invariant discovered without the forward analysis, cf. $\mathtt{r} \geq \mathtt{nit}$, the impact quantity maximizes the linear programming problem $\mathtt{Range}_{\mathtt{i}}^{\natural}(\mathtt{r} \geq \mathtt{nit})$ to u (cf. $0 \leq \mathtt{r} \leq u$), for any input variable `i`, even when $\mathtt{i} \neq \mathtt{r}$. The linear programming problem $\mathtt{Range}_{\mathtt{i}}^{\natural}(\mathtt{r} \geq \mathtt{nit})$ is:

$$\begin{aligned} \text{maximize} \quad & k \\ \text{subject to} \quad & \mathtt{r} \geq \overline{\mathtt{nit}} \\ \wedge \quad & \mathtt{r} \geq \underline{\mathtt{nit}} \\ \wedge \quad & 0 \leq \mathtt{r} \leq u \\ \wedge \quad & 0 \leq k \leq \overline{\mathtt{nit}} - \underline{\mathtt{nit}} \end{aligned}$$

In this case, the variable $\overline{\mathtt{nit}}$ and $\underline{\mathtt{nit}}$ are not constrained to be equal, thus they can be minimized and maximized independently as long as they satisfy the other constraints, resulting in $k = u$. On the other hand, the invariant discovered with the help of the forward pre-analysis is $\mathtt{r} = \mathtt{nit}$. The impact quantity maximizes the linear programming problem $\mathtt{Range}_{\mathtt{i}}^{\natural}(\mathtt{r} = \mathtt{nit})$ to 0 whenever the input variable $\mathtt{i} \neq \mathtt{r}$. In such case, the linear programming problem $\mathtt{Range}_{\mathtt{i}}^{\natural}(\mathtt{r} = \mathtt{nit})$ is:

$$\begin{aligned} \text{maximize} \quad & k \\ \text{subject to} \quad & \mathtt{r} = \overline{\mathtt{nit}} \\ \wedge \quad & \mathtt{r} = \underline{\mathtt{nit}} \\ \wedge \quad & 0 \leq \mathtt{r} \leq u \\ \wedge \quad & 0 \leq k \leq \overline{\mathtt{nit}} - \underline{\mathtt{nit}} \end{aligned}$$

Note that, both $\overline{\mathtt{nit}}$ and $\underline{\mathtt{nit}}$ are constrained to be equal. Therefore, their maximum distance is 0.

Example 8. To show that also the syntactic dependency analysis can benefit from a forward pre-analysis (and in general, from a numerical analysis [32]), consider the following program:

```
1 x = y
2 x = x - y
```

The program above assigns first the value of `y` to `x` and then subtracts `y` from `x`. The result is that `x` is zero at the end of the program execution, while `y` maintains its input value. Let us assume we are interested in the variables that are relevant

to compute the value of `x`. Without a forward pass, the syntactic dependency analysis (a backward analysis) would infer that `y` is relevant for the value of `x` after handling the second assignment at Line 2. Then, the first assignment at Line 1 would add no dependency as `x` is overwritten. On the other hand, a forward analysis could be able to infer that at the end of the program, the value of `x` is zero. As a consequence, the information that `x` is a constant value supersedes the information that `x` is used (at the end of the program). Therefore, the syntactic dependency analysis would infer that `y` is, in fact, irrelevant.

As all these optimizations and analysis stages may increase the analysis time (see Table 5), we allow the final user to choose whether to enable them.

7 Evaluation

In this section, we showcase the potential of TimeSec on the S2N-Bignum library[5]. In Appendix A, we show an evaluation on the SV-Comp benchmarks, focusing on the effect of changes in the input space, the analysis time, and the categorization of input variables.

The S2N-Bignum library [3] is a collection of arithmetic routines designed for cryptographic applications. All the routines are written in pure machine code, designed to be callable from C and other high-level languages. Each function is written in a constant-time style, to avoid leaking information through timing side-channels. Constant-time means that the execution time of an S2N-Bignum operation is independent of the actual numbers involved, depending only on their nominal sizes. If a result does not fit in the provided size, it is systematically truncated modulo that size. Allocation of memory is always the caller's responsibility, the S2N-Bignum interface only uses pointers to pre-existing arrays. The developers avoid the use of certain machine instructions known to be problematic for constant-time execution, such as the division instruction. Furthermore, on ARM platforms, the library sets the DIT (Data Independent Timing) bit to have hardware guaranteed constant-time execution.

The library is fully verified for functional correctness in HOL Light [19], but the verification of the constant-time property is still ongoing. At present, the constant-time property is enforced by the strict compliance to the constant-time design discipline and the use of empirical testing. Their empirical result[6] shows that the variation in runtime with respect to the data being manipulated is within a few percent in all the cases. Unfortunately, the empirical study is not sufficient to guarantee the constant-time property, as it is not exhaustive and does not cover all the possible inputs. On the other hand, the quantitative analysis of TimeSec provides a formal verification of the constant-time property. In particular, whenever an input variable has no impact on the global number of

[5] `https://github.com/awslabs/s2n-bignum`

[6] (Last accessed: 14th May 2024) `https://github.com/awslabs/s2n-bignum?tab=readme-ov-file#benchmarking-and-constant-time`

loop iterations, it is formally guaranteed that the number of iterations is independent of the values of that input variable. Formally, a program P is free of timing side-channels with respect to an input variable $\mathtt{i} \in \Delta$, if and only if $\mathrm{P} \models \mathscr{B}_{\mathtt{i}}^{\leq 0}$. By Theorem 3, we know that this is implied from $\mathtt{Range}_{\mathtt{i}}^{\natural}(\Lambda_{\mathrm{P}}^{*}) \leq 0$. Therefore, the verification of *timing side-channel freedom* is sound with respect to our quantitative analysis of input variables. We partition the input variables of the S2N-Bignum library into two subsets. The nominal size variables and additional parameters that may safely influence the runtime into $\Delta|_{\mathrm{S}}$. The variables that represent the actual numerical values and additional parameters that, instead, should not influence the execution time into $\Delta|_{\mathrm{N}}$. The S2N-Bignum library is free of timing side-channels, whenever for any program P in S2N-Bignum and any numerical input variable $\mathtt{i} \in \Delta|_{\mathrm{N}}$, it holds that $\mathtt{Range}_{\mathtt{i}}^{\natural}(\Lambda_{\mathrm{P}}^{*}) = 0$.

For our setup, we consider the disassembled operations[7] of the S2N-Bignum library as input programs with a few rewriting steps to fit the set of supported operations of our tool. Mostly, the rewriting steps soundly resolve the few jumps that arise from the disassembling process. Our benchmark contains a total of 72 disassembled arithmetic routines, excluding only a single operation (program `bignum_modexp`) that has function calls, which our tool does not yet support. On average, each program has about 83 lines of code, for a total of 5984 lines of code.

The library contains a total of 1172 variables, 272 of which are input variables. Table 4 reports the analysis findings for the input variables of the S2N-Bignum library: column Maybe Dangerous reports variables which could be prone to timing side-channel attacks (namely $\mathtt{Range}_{\mathtt{i}}^{\natural}(\Lambda_{\mathrm{P}}^{*}) > 0$), column Zero Impact reports the variables with an impact quantity of zero (namely $\mathtt{Range}_{\mathtt{i}}^{\natural}(\Lambda_{\mathrm{P}}^{*}) = 0$). The property $\mathscr{B}_{\mathtt{i}}^{\leq 0}$ holds for input variables $\mathtt{i}$ that have an impact quantity of zero (column Zero Impact). Overall, we soundly verified that 187 (69%) of the input variables do not influence the global number of iterations, while 85 (31%) are maybe dangerous and maybe susceptible to timing side-channel attacks. Column Safe $\Delta|_{\mathrm{S}}$ reports the nominal size variables (called s_i), column Numerical $\Delta|_{\mathrm{N}}$ reports the numerical variables (called n_i, where i is the index of the variable as they appear in the function signature). Table 4 shows that no numerical variable is identified as potentially dangerous, indeed Maybe Dangerous $\cap\, \Delta|_{\mathrm{N}} = \emptyset$ in all rows. We conclude that the S2N-Bignum library is *free of timing side-channels*.

Additionally, we perform an ablation study to evaluate the impact of the dependency analysis and the other optimizations on our tool. The first row No Optimizations of Table 5 reports the analysis findings without the various analysis stages of Section 6, while the second row TimeSec shows the finding of the full TimeSec analysis. Without the dependency analysis we do not apply program slicing anymore, we handle bitwise operations and array accesses with a conservative over-approximation that may lead to false positives. Generally, we notice that the invariant inferred from the global loop bound analysis alone is

[7] We used Ghidra (`https://ghidra-sre.org/`) to disassemble the library and extract the arithmetic routines.

Table 4: Input composition of the S2N-BIGNUM library. The variables SAFE $\Delta|_S$ are highlighted in green, while the variables NUMERICAL $\Delta|_N$ in red. No numerical variable should be MAYBE DANGEROUS.

PROGRAM	INPUT VARIABLES Δ SAFE $\Delta\|_S$	INPUT VARIABLES Δ NUMERICAL $\Delta\|_N$	MAYBE DANGEROUS	ZERO IMPACT
Add	s_1, s_3, s_5	n_2, n_4, n_6	s_1	s_3, s_5, n_2, n_4, n_6
Amontifier	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Amontmul	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Amontredc	s_1, s_3, s_6	n_2, n_4, n_5	s_1, s_3, s_6	n_2, n_4, n_5
Amontsqr	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Bitfield	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Bitsize	s_1	n_2	s_1	n_2
Cdiv	s_1, s_3	n_2, n_4, n_5	s_1, s_3	n_2, n_4, n_5
Cdiv_exact	s_1, s_3	n_2, n_4, n_5	s_1	n_2, s_3, n_4, n_5
Cld	s_1	n_2	s_1	n_2
Clz	s_1	n_2	s_1	n_2
Cmadd	s_1, s_4	n_2, n_3, n_5	s_1, s_4	n_2, n_3, n_5
Cmnegadd	s_1, s_4	n_2, n_3, n_5	s_1, s_4	n_2, n_3, n_5
Cmod	s_1	n_2, n_3	s_1	n_2, n_3
Cmul	s_1, s_4	n_2, n_3, n_5	s_1, s_4	n_2, n_3, n_5
Coprime	s_1, s_3	n_2, n_4, n_5	s_1, s_3	n_2, n_4, n_5
Copy	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Copy_row_from_table	s_3, s_4	n_1, n_2, n_5	s_3, s_4	n_1, n_2, n_5
Copy_row_from_table_16_neon	s_3	n_1, n_2, n_4	s_3	n_1, n_2, n_4
Copy_row_from_table_32_neon	s_3	n_1, n_2, n_4	s_3	n_1, n_2, n_4
Copy_row_from_table_8n_neon	s_3, s_4	n_1, n_2, n_5	s_3, s_4	n_1, n_2, n_5
Ctd	s_1	n_2	s_1	n_2
Ctz	s_1	n_2	s_1	n_2
Demont	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Digit	s_1	n_2, n_3	s_1	n_2, n_3
Digitsize	s_1	n_2	s_1	n_2
Divmod10	s_1	n_2	s_1	n_2
Emontredc	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Eq	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Even	s_1	n_2		s_1, n_2
Ge	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Gt	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Iszero	s_1	n_2	s_1	n_2
Le	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Lt	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Madd	s_1, s_3, s_5	n_2, n_4, n_6	s_1, s_3, s_5	n_2, n_4, n_6
Modadd	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Moddouble	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Modifier	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Modinv	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Modoptneg	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Modsub	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Montifier	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Montmul	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Montredc	s_1, s_3, s_6	n_2, n_4, n_5	s_1, s_3, s_6	n_2, n_4, n_5
Montsqr	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Mul	s_1, s_3, s_5	n_2, n_4, n_6	s_1, s_3, s_5	n_2, n_4, n_6
Muladd10	s_1	n_2, n_3	s_1	n_2, n_3
Mux	s_2	n_1, n_3, n_4, n_5	s_2	n_1, n_3, n_4, n_5
Mux16	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Negmodinv	s_1	n_2, n_3	s_1	n_2, n_3
Nonzero	s_1	n_2	s_1	n_2
Normalize	s_1	n_2	s_1	n_2
Odd	s_1	n_2		s_1, n_2
Of_word	s_1	n_2, n_3	s_1	n_2, n_3
Optadd	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Optneg	s_1	n_2, n_3, n_4	s_1	n_2, n_3, n_4
Optsub	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Optsubadd	s_1	n_2, n_3, n_4, n_5	s_1	n_2, n_3, n_4, n_5
Pow2	s_1	n_2, n_3	s_1	n_2, n_3
Shl_small	s_1, s_3	n_2, n_4, n_5	s_1, s_3	n_2, n_4, n_5
Shr_small	s_1, s_3	n_2, n_4, n_5	s_1	s_3, n_2, n_4, n_5
Sqr	s_1, s_3	n_2, n_4	s_1, s_3	n_2, n_4
Sub	s_1, s_3, s_5	n_2, n_4, n_6	s_1	s_3, s_5, n_2, n_4, n_6
Word_bytereverse	n_1			n_1
Word_clz	n_1			n_1
Word_ctz	n_1			n_1
Word_divstep59	n_1, n_2, n_3, n_4			n_1, n_2, n_3, n_4
Word_max	n_1, n_2			n_1, n_2
Word_min	n_1, n_2			n_1, n_2
Word_negmodinv	n_1			n_1
Word_recip	n_1			n_1
TOTAL VARIABLES:	93	179	85	187

Table 5: Ablation study of TimeSec on the S2N-Bignum benchmark.

Component	Input Variables		Total Analysis Time (s)			
	Maybe Dangerous	Zero Impact	Deps	Inv	LP	Tot
No Optimizations	266	6	0.0	16.36	7.79	26.45 ± 0.81
TimeSec	85	187	51.81	55.83	0.27	110.48 ± 4.94

not tight enough to produce a precise quantification of the impact. Therefore, we are not able to infer useful insights from our analysis as 266 input variables are maybe dangerous. In particular, the 6 input variables with zero impact belong to acyclic programs. Regarding the analysis time, column Deps refers to the time of the dependency analysis, column Inv for the global loop bound analysis, and column LP for the quantification of impact. The time is reported in seconds for the evaluation of the 72 programs. The last column Tot reports the total analysis time, with the standard deviation after the symbol ±. Table 5 does not show the time for parsing, logging and other overheads of the tool. We notice that without the optimizations, the analysis time is about 4 times faster than the full analysis, with most time spent on the linear programming problem as more variables need to be quantified. In this case, the standard deviation of the total analysis time (after ± in the column Tot) is the lowest, meaning that the analysis time is more consistent among programs. With only the dependency analysis on, the analysis usually takes around 50 seconds and, without optimizations, the global loop bound analysis is quite fast. The full analysis is about 100 seconds in total, with an average of 1.22 seconds per program. Most of the analysis time is spent on the syntactic dependency and the global loop bound analysis. Notably, the linear programming problem to quantify the impact of input variables takes less than half a second in total for the whole library. However, the analysis time is not consistent for all the programs, in fact, the analysis time for each program ranges from 0.03 to 33.88 seconds (standard deviation of about 4 seconds). Nevertheless, the full analysis is also the most precise, as it is able to exclude the most number of maybe dangerous variables.

In conclusion, the S2N-Bignum library is a good candidate for our analysis, as it is a real-world cryptographic library potentially vulnerable against timing side-channel attacks for numerical input variables. Up to the decompilation phase and the chosen abstraction of the runtime, cf. the global number of iterations, our analysis soundly verifies that no input variable containing numerical data is susceptible to timing side-channel attacks.

8 Conclusion

In this study, we proposed a static analysis based on abstract interpretation to quantify the impact of input variables on the global number of loop iterations. We certified the absence of timing side-channels in the S2N-Bignum library for cryptographic applications, demonstrating the practicality of our approach.

Looking ahead, we could extend our analysis by considering cost functions (as done in Carbonneaux et al. [8], Urban and Miné [42]), such as the number of executed machine instructions, to obtain a closer abstraction of the runtime behavior of a program. Moreover, the syntactic dependency analysis currently handles only control flow graphs generated without jumps. Considering jumps would allow the analysis to work with low level code, without the need of decompilers and rewriting phases.

Our quantitative approach contrasts the qualitative work proposed by Urban and Müller [43], which employs the trace semantics to address non-determinism and non-termination. A possible solution to integrate termination in the quantitative framework is to run a termination analysis [42, 16], alongside the global loop bound analysis, to add – as an output value – the potential non-termination state. For instance, Urban and Miné [41] partition the input space into equivalence classes that share the same output, considering termination as such. We are confident that a combination of their approach on termination with our quantitative analysis could discover termination-aware impact quantities. Indeed, by knowing that some executions do not terminate after a certain loop could improve the precision of the quantitative bound as we avoid considering all the successive iterations as potential executions. Furthermore, to address non-determinism we could consider the sequence of all possible non-deterministic choices as a parameter of the semantics, as done by Cousot and Monerau [12], Parolini and Miné [32].

Another promising direction is the exploration of new impact definitions for cyber-physical systems [26]. 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 [6, 7].

References

[1] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei. Decomposition instead of self-composition for proving the absence of timing channels. *PLDI*, 2017. https://doi.org/10.1145/3140587.3062378.

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

[3] AWS. s2n-bignum. `https://github.com/awslabs/s2n-bignum`.

[4] G. Barthe, P. R. D'Argento, and T. Rezk. Secure information flow by self-composition. *IEEE*, 2011. https://doi.org/10.1017/S0960129511000193.

[5] P. Cadek, C. Danninger, M. Sinn, and F. Zuleger. Using loop bound analysis for invariant generation. *FMCAD*, 2018. https://doi.org/10.23919/FMCAD.2018.8603005.

[6] 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.

[7] 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.

[8] Q. Carbonneaux, J. Hoffmann, and Z. Shao. Compositional certified resource bounds. *PLDI*, 2015. https://doi.org/10.1145/2737924.2737955.

[9] 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.

[10] 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.

[11] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. *POPL*, 1978. https://doi.org/10.1145/512760.512770.

[12] P. Cousot and M. Monerau. Probabilistic abstract interpretation. *ESOP*, 2012. https://doi.org/10.1007/978-3-642-28869-2_9.

[13] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1982. ISBN 0201101505.

[14] J.-F. Dhem, F. Koeune, P.-A. a. Leroux, P. Mestré, J.-J. Quisquater, and J.-L. Willems. A practical implementation of the timing attack. *CARDIS*, 2000. https://doi.org/10.1007/10721064_15.

[15] A. Ermedahl, C. Sandberg, J. Gustafsson, S. Bygde, and B. Lisper. Loop bound analysis based on a combination of program slicing, abstract interpretation, and invariant analysis. *WCET*, 2007. https://doi.org/10.4230/OASICS.WCET.2007.1194.

[16] L. Gonnord, D. Monniaux, and G. Radanne. Synthesis of ranking functions using extremal counterexamples. *PLDI*, 2015. https://doi.org/10.1145/2813885.2737976.

[17] P. Granger. Improving the results of static analyses of programs by local decreasing iterations. *FSTTCS*, 1992. https://doi.org/10.1007/3-540-56287-7_95.

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

[19] J. Harrison. HOL light: An overview. *TPHOLs*, 2009. https://doi.org/10.1007/978-3-642-03359-9_4.

[20] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. 2009. https://doi.org/10.1007/978-3-642-02658-4_52.

[21] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. 2019. https://doi.org/10.1109/SP.2019.00002.

[22] P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. *CRYPTO*, 1996. https://doi.org/10.1007/3-540-68697-5_9.

[23] P. C. Kocher, J. Jaffe, and B. Jun. Differential power analysis. *CRYPTO*, 1999. https://doi.org/10.1007/3-540-48405-1_25.

[24] B. Köpf and D. A. Basin. An information-theoretic model for adaptive side-channel attacks. *CCS*, 2007. https://doi.org/10.1145/1315245.1315282.
[25] B. Köpf and A. Rybalchenko. Automation of quantitative information-flow analysis. *SFM*, 2013. https://doi.org/10.1007/978-3-642-38874-3_1.
[26] M. Kwiatkowska. Advances and challenges of quantitative verification and synthesis for cyber-physical systems. *SOSCYPS*, 2016. https://doi.org/10.1109/SOSCYPS.2016.7579999.
[27] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. *USENIX Security Symposium*, 2018. https://doi.org/10.1145/3357033.
[28] 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.
[29] D. Mazzucato, M. Campion, and C. Urban. Quantitative Input Usage Static Analysis. *NFM*, 2024. URL `https://hal.science/hal-04339001`.
[30] A. Miné. The octagon abstract domain. *Working Conference on Reverse Engineering*, 2001. https://doi.org/10.1109/WCRE.2001.957836.
[31] H. Omar, M. Ahmad, and O. Khan. Graphtuner: An input dependence aware loop perforation scheme for efficient execution of approximated graph algorithms. *ICCD*, 2017. https://doi.org/10.1109/ICCD.2017.38.
[32] F. Parolini and A. Miné. Sound abstract nonexploitability analysis. *VMCAI*, 2024. https://doi.org/10.1007/978-3-031-50521-8_15.
[33] Q. Phan, L. Bang, C. S. Pasareanu, P. Malacaria, and T. Bultan. Synthesis of adaptive side-channel attacks. *IEEE*, 2017. https://doi.org/10.1109/CSF.2017.8.
[34] 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.
[35] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. *VMCAI*, 2004. https://doi.org/10.1007/978-3-540-24622-0_20.
[36] X. Rival. Understanding the origin of alarms in astrée. *SAS*, 2005. https://doi.org/10.1007/11547662_21.
[37] S. Saha, U. S. Barbara, U. S. Ghentiyala, and U. L. Shihua. Obtaining information leakage bounds via approximate model counting. *PLDI*, 2023. https://doi.org/10.1145/3591281.
[38] M. Sinn, F. Zuleger, and H. Veith. Complexity and resource bound analysis of imperative programs using difference constraints. *Journal of Automated Reasoning*, 2017. https://doi.org/10.1007/S10817-016-9402-4.
[39] B. Steensgaard. Points-to analysis in almost linear time. *POPL*, 1996. https://doi.org/10.1145/237721.237727.
[40] T. Terauchi and A. Aiken. Secure information flow as a safety problem. *SAS*, 2005. https://doi.org/10.1007/11547662_24.
[41] C. Urban and A. Miné. An abstract domain to infer ordinal-valued ranking functions. *ESOP*, 2014. https://doi.org/10.1007/978-3-642-54833-8_22.

[42] C. Urban and A. Miné. A decision tree abstract domain for proving conditional termination. *SAS*, 2014. https://doi.org/10.1007/978-3-319-10936-7_19.

[43] 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.

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

[45] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 2008. https://doi.org/10.1145/1347375.1347389.

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

A SV-Comp Benchmarks

The SV-Comp benchmarks[8] are a collection of programs used for verification competition. The benchmarks are divided into different categories, such as termination, memory safety, reachability. As of 2024, the SV-Comp repository hosts thousands of programs, which are written in C and annotated with assertions. In this evaluation, we conduct a comprehensive study focusing on: the effect of changes in the input space, the analysis time, and the categorization of input variables. We focus on the categories of Termination Crafted, and Termination Crafted Lit. These categories describe programs that are crafted to be challenging for termination analysis. In total, we selected 208 programs (68 from Termination Crafted, and 140 from termination Crafted Lit), with 5705 total lines of code. An average of 27 lines of code per program.

To evaluate TimeSec against the SV-Comp benchmarks, we consider the input variables as unbounded non-negative integers. We repeat the analysis 5 times, each time with a different bound on the input variables, ranging from $[0, 10]$ to $[-\infty, +\infty]$. Table 6 reports, for each bound range, the average quantity of impact (column Average), the standard deviation (column Std), and the analysis time for the dependency analysis (column Deps), the global loop bound analysis (column Inv), and the quantification of the impact (column LP). We exclude to take into account quantities that are infinite, as they would disrupt the average calculation. Note that, even in presence of a bounded input space, the impact of a variable could be infinite if the global loop bound analysis is not able to infer a bound on the possible number of iterations.

From Table 6, we observe that the average quantity of impact increases with the bound range (column Average). This is expected, as the larger the input space, the more the variance in the values of input variables, and the more the impact on the global number of iterations. However, as soon as the input space is unbounded, the measured quantities that are not infinite are very low. In this setting, a variable often has either an impact of 0 or $+\infty$. Regarding the analysis time, as expected we notice that the syntactic dependency analysis (column Deps) is not influenced by the bound range. The reason is that the syntactic dependency analysis is not a semantics analysis and does not depend on the values of the input variables. On the contrary, the global loop bound analysis (column Inv) and the quantification of the impact (column LP) are affected. From bounded to unbounded input space, we observe a reduction in the analysis time. In fact, in the context of a bigger input space, the analysis precision drops drastically and thus propagate less information faster. The global loop bound analysis is the most time-consuming part of the analysis. Overall, the analysis time is acceptable, with an average of 0.11 seconds per program, and a total of about 126 seconds for the whole benchmark suite, cf. 208 programs for 5 different bound ranges (1055 programs in total).

Table 7 shows the composition of variables of the two categories of SV-Comp benchmarks. In terms of the k-bounded impact property (cf. Equation (6)), the

[8] `https://sv-comp.sosy-lab.org/2024`

Table 6: Quantitative results for the SV-Comp benchmarks.

Benchmark	Bound Ranges	Quantities ($< \infty$) Average	Std	Analysis Time (s) Deps	Inv	LP	Tot
Termination Crafted (68 programs)	$0-10$	6.12	6.16	0.51	3.54	0.32	6.19
	$0-100$	50.13	48.44	0.51	3.5	0.32	6.18
	$0-1000$	500.13	483.18	0.5	3.53	0.32	6.15
	≥ 0	0.0	0.0	0.5	2.4	0.26	5.03
	$[-\infty,+\infty]$	0.0	0.0	0.46	2.01	0.05	4.38
Termination Crafted Lit (140 programs)	$0-10$	435.46	1892.32	1.56	16.6	1.02	23.08
	$0-100$	38248.52	194557.7	1.54	16.66	1.01	23.04
	$0-1000$	38577853.04	192885029.79	1.55	16.66	1.01	23.03
	≥ 0	0.0	0.0	1.49	9.66	0.77	15.8
	$[-\infty,+\infty]$	0.0	0.0	1.4	8.27	0.22	13.65

Table 7: Analysis findings for the SV-Comp benchmarks.

Benchmark	Bound Ranges	Variables May Impact	Zero Impact
Termination Crafted (68 programs)	$0-10$	99/135	13/284
	$0-100$	99/135	13/284
	$0-1000$	99/135	13/284
	≥ 0	99/135	13/284
	$[-\infty,+\infty]$	105/141	7/278
Termination Crafted Lit (140 programs)	$0-10$	275/336	36/707
	$0-100$	277/338	34/705
	$0-1000$	277/338	34/705
	≥ 0	277/338	34/705
	$[-\infty,+\infty]$	286/348	25/695

column May Impact corresponds to $\mathscr{B}_{\mathtt{i}}^{\leq k}$ for $k > 0$, while the columns Zero Impact corresponds to $\mathscr{B}_{\mathtt{i}}^{\leq 0}$. For each bound range, we report the number of input variables / total variables (cf., local and input variables together) that fall into each category. As expected, by enlarging the input space, the number of maybe dangerous variables increases, while the number of zero used variables decreases. Overall, our analysis is able to verify that most of the input variables in the SV-Comp benchmarks influence the global number of loop iterations. This is expected, as the benchmarks are crafted to be challenging for termination analysis, thus it is not surprising that the input variables have a significant impact on the global number of iterations. Unfortunately, such programs have invariants that are, on purpose, hard to infer. Our analysis can do little to achieve a tight quantification in such case. In conclusion, we notice that by enlarging the input space, the number of variables that may impact the runtime increases as more variety in the input values leads to more impact on the global number of iterations.