Termination Resilience Static Analysis

Naïm Moussaoui Remil and Caterina Urban

Inria & ENS | PSL, Paris, France
{naim.moussaoui-remil,caterina.urban}@inria.fr

Abstract. We present a novel abstract interpretation-based static analysis framework for proving Termination Resilience, the absence of Robust Non-Termination vulnerabilities in software systems. Robust Non-Termination characterizes programs where an untrusted (e.g., externally-controlled) input can force infinite execution, independently of other trusted (e.g., controlled) variables. Our framework is a semantic generalization of Cousot and Cousot's abstract interpretation-based ranking function derivation, and our sound static analysis extends Urban and Miné's decision tree abstract domain in a non-trivial way to manage the distinction between untrusted and trusted program variables. Our approach is implemented in an open-source tool and evaluated on benchmarks sourced from SV-COMP and modeled after real-world software, demonstrating practical effectiveness in verifying Termination Resilience and detecting potential Robust Non-Termination vulnerabilities.

1 Introduction

Termination is one of the oldest and most fundamental properties of programs. Yet, in practice, requiring termination in every possible case can be unnecessarily restrictive. Indeed, not all program variables play the same role in program termination. Some variables are *untrusted*: their values may be externally controlled, for instance by user input, an operating system scheduler, or a third-party library. If controlling such variables alone is sufficient to enforce divergence, we say that the program is vulnerable to *Robust Non-Termination*, which represents a serious concern in practice. By contrast, other variables are *trusted*: their values stem from non-deterministic choices, internal computations, or library calls under the control of the programmer. Non-termination that may be avoided through these trusted choices may be acceptable, as it cannot be reliably exploited by an adversary; it reflects internal system behavior rather than external influence. From a security perspective, this distinction is crucial: divergence caused by untrusted inputs constitutes a potential denial-of-service vulnerability, whereas divergence caused by trusted variables does not. From a software engineering standpoint, the distinction is equally important as it hints at a principled way to triage non-termination alarms, helping developers prioritize the more critical cases.

This distinction motivates the property of *Termination Resilience*; for every possible (sequence of) untrusted input(s), there exists at least one terminating execution. In semantic terms, we treat untrusted variables demonically, since termination must be ensured under all value possibilities, while trusted variables

```
1 x := input;
2 z := 10;
if 3(...) {
   while 4(z >= 0) {
     5z := z - x;
   }
} else {
  6c := [-2, 1];
   while 7(z >= x) {
     8z := z + c;
   }
}9
```

Fig. 1: Program P.

are treated angelically, since it suffices that some of their values enable termination. Besides, we support both scenarios where untrusted inputs have full and partial knowledge on trusted variables. Consider program P (inspired from our benchmarks and written in the small programming language that we use for illustration in the paper) in Figure 1: the variable `x` is an untrusted input (cf. program label 1), while the value of `c` is non-deterministically chosen between -2 and 1 (cf. label 6). Both `while` loops in the program are non-terminating, the first when $x \leq 0$, and the second when ($z \geq x$ and) $c \geq 0$. However, only for the first loop (non-)termination depends on untrusted variables, while the second loop satisfies Termination Resilience. Termination Resilience thus provides a nuanced alternative to Termination. Note that in this program the untrusted input is the first to choose a value (cf. label 1), hence it has no knowledge about the trusted variables (`c` and `z`) values.

We design *an abstract interpretation-based static analysis framework to prove Termination Resilience*. Our framework is a generalization of Cousot and Cousot's idea of computing a ranking function by abstract interpretation [12]. Specifically, we extend their semantic definitions to accommodate the asymmetric treatment of untrusted and trusted program variables. We derive a sound static analysis for Termination Resilience by building upon Urban and Miné's decision tree abstract domain [34], enhancing it with novel and non-trivial transformer operators to effectively manage this mixed settings. When unable to prove that a program always satisfies Termination Resilience, *our static analysis automatically infers sufficient preconditions that ensure Termination Resilience*. For example, for program P in Figure 1, our analysis proves Termination Resilience for the second loop and automatically infers the precondition $x > 0$ for the first loop.

Our analysis is implemented in an open-source tool, as an extension of the FuncTion static analyzer [32], and evaluated on benchmarks drawn from the International Competition on Software Verification (SV-COMP) and modeled after real-world software. The experimental results demonstrate the practical usefulness of the analysis as a means to triage non-termination alarms based on

whether they depend on untrusted or trusted variables, provings developers with a systematic way to prioritize the most critical cases.

2 Termination Resilience

In this section, we give a mathematical characterization of the behavior of a program, and we formally define our property of interest, *Termination Resilience*.

Program Semantics. We model the operational behavior of a program as a *transition system* $\langle \Sigma, \tau \rangle$ where Σ is a (potentially infinite) set of program states, and the transition relation $\tau \subseteq \Sigma \times \Sigma$ describes possible transition between states [10,9]. Specifically, for our purposes, $\tau \stackrel{\text{def}}{=} \tau^i \uplus \tau^r$ is the disjoint union of *input transitions* in τ^i, which are transitions that consume an untrusted input, and *regular transitions* in τ^r. For simplicity, without loss of generalization, we assume that all transitions $\langle s, s' \rangle \in \tau$ from a state $s \in \Sigma$ are of the same kind, i.e., either all input transition or all regular transitions. The set of final program states is $\Omega_\tau \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma \colon \langle s, s' \rangle \notin \tau\}$.

Given a transition system $\langle \Sigma, \tau \rangle$, the function $\mathrm{pre}_{\tau^r} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a given set of states $X \subseteq \Sigma$ to the set of their predecessors with respect to τ^r: $\mathrm{pre}_{\tau^r}(X) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \exists s' \in X : \langle s, s' \rangle \in \tau^r\}$, and the function $\widetilde{\mathrm{pre}}_{\tau^i} \colon \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)$ maps a set of states $X \subseteq \Sigma$ to the set of states whose successors with respect to τ^i are all in X: $\widetilde{\mathrm{pre}}_{\tau^i}(X) \stackrel{\text{def}}{=} \{s \in \Sigma \mid \forall s' \in \Sigma : \langle s, s' \rangle \in \tau^i \Rightarrow s' \in X\}$.

In the following, given a set S, let ε be an empty sequence of elements in S, S^+ be the set of all non-empty finite sequences of elements in S, S^ω be the set of all infinite sequences, $S^{+\infty} \stackrel{\text{def}}{=} S^+ \cup S^\omega$ be the set of all non-empty finite or infinite sequences of elements in S, and $S^{*\infty} \stackrel{\text{def}}{=} S^{+\infty} \cup \{\varepsilon\}$. We write $T^+ \stackrel{\text{def}}{=} T \cap S^+$, $T^\omega \stackrel{\text{def}}{=} T \cap S^\omega$ for the selection of the non-empty finite sequences and the infinite sequences of $T \in \mathcal{P}(S^{+\infty})$, and $T; T' = \{tst' \in S^{+\infty} \mid s \in S \wedge ts \in T \wedge st' \in T'\}$ for the merging of two sets of sequences $T \in \mathcal{P}(S^+)$ and $T' \in \mathcal{P}(S^{+\infty})$, when a finite sequence in T terminates with the initial element of a sequence in T'.

A *trace* $\sigma \in \Sigma^{+\infty}$ is a non-empty sequence of program states where each pair of consecutive states $s, s' \in \Sigma$ is described by the transition relation τ, i.e., $\langle s, s' \rangle \in \tau$. We write σ_n to denote the n-th state of a trace $\sigma \in \Sigma^{+\infty}$ (with $n < |\sigma|$), σ^i (resp. σ^r) for the (possibly empty) sequence of input (resp. regular) transitions (i.e., pairs of consecutive states) in σ, and T^i for the set $\{\sigma^i \mid \sigma \in T\}$ of sequences of input transitions in traces in a set $T \in \mathcal{P}(\Sigma^{+\infty})$ (resp. T^r for the set $\{\sigma^r \mid \sigma \in T\}$ of sequences of regular transitions). The *trace semantics* $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ generated by a transition system $\langle \Sigma, \tau \rangle$ is the union of all non-empty finite traces terminating in Ω_τ, and all infinite traces. It can be expressed as a least fixpoint in the complete lattice $\langle \mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+ \rangle$[9]:

$$\begin{aligned} \Lambda &= \mathsf{lfp}^{\sqsubseteq} \Theta \\ \Theta(T) &\stackrel{\text{def}}{=} \Omega_\tau \cup (\tau; T) \end{aligned} \tag{1}$$

(a) The transition system $\langle \Sigma_1, \tau_1 \rangle$. (b) The transition system $\langle \Sigma_2, \tau_2 \rangle$.

Fig. 2: The transition systems of Examples 1 and 2. Input transitions are represented with dashed red lines, while solid black lines represent regular transitions.

where the computational order is $T_1 \sqsubseteq T_2 \overset{\text{def}}{\Leftrightarrow} T_1^+ \subseteq T_2^+ \wedge T_1^\omega \supseteq T_2^\omega$. In the following, we write $\Lambda[\![P]\!]$ to denote the trace semantics of a given program P. Given an initial set of states $I \subseteq \Sigma$, we define $\Lambda[\![P]\!]I \overset{\text{def}}{=} \{\sigma \in \Lambda[\![P]\!] \mid \sigma_0 \in I\}$ as the restriction of the trace semantics of P to traces starting in I.

Example 1. Let $\langle \Sigma_1, \tau_1 \rangle$ be a transition system with $\Sigma_1 = \{a, b, c\}$ and $\tau_1 = \tau_1^i \cup \tau_1^r$ where $\tau_1^i = \{\langle a, b\rangle, \langle a, c\rangle\}$ and $\tau_1^r = \{\langle c, c\rangle\}$ (cf. Figure 2a). The trace semantics generated by $\langle \Sigma_1, \tau_1 \rangle$ and starting in $I = \{a\}$ is $\Lambda_1 = \{ab, ac^\omega\}$. ◁

Example 2. Let $\langle \Sigma_1, \tau_1 \rangle$ be the transition system from Example 1 and let $\langle \Sigma_2, \tau_2 \rangle$ be another transition system with $\Sigma_2 = \Sigma_1 \cup \{d, e\}$ and $\tau_2 = \tau_1^i \cup \tau_2^r$ where $\tau_2^r = \tau_1^r \setminus \{\langle c, c\rangle\} \cup \{\langle b, d\rangle, \langle b, e\rangle, \langle e, e\rangle\}$ (cf. Figure 2b). The trace semantics generated by $\langle \Sigma_2, \tau_2 \rangle$ and starting in $I = \{a\}$ is $\Lambda_2 = \{abd, abe^\omega, ac\}$. ◁

Program Property. As customary in abstract interpretation [10], we represent properties of entities in a universe $\mathbb{U}$ by a subset of this universe. We formally define the Robust Non-Termination program property.

Definition 1 (Robust Non-Termination). *The* Robust Non-Termination *program property is the set of sets of traces for which a (possibly empty) sequence of input transitions only yields diverging traces:*

$$\mathcal{RNT} \overset{def}{=} \left\{T \in \mathcal{P}(\Sigma^{+\infty}) \mid \exists j \in T^i \, \forall \sigma \in T \colon \sigma^i = j \Rightarrow \sigma \in T^\omega\right\} \tag{2}$$

Note that any set $T \subseteq \mathcal{P}(\Sigma^\omega)$ of infinite traces always belongs to $\mathcal{RNT}$, even if the traces do not consume input values (in such case $j = \varepsilon$), i.e., an always non-terminating program is always vulnerable to non-termination exploits.

Example 3. Let $\langle \Sigma_1, \tau_1 \rangle$ be the transition system from Example 1. The input transition $\langle a, c\rangle$ only yields the diverging trace $ac^\omega \in \Lambda_1$. Thus $\Lambda_1 \in \mathcal{RNT}$. ◁

The negation of Robust Non-Termination is our property of interest. It expresses the ability of the program to (possibly) defend itself from non-termination exploits. Formally, we call this property *Termination Resilience*.

Definition 2 (Termination Resilience). *The* Termination Resilience *program property is the set of sets of program traces for which there exists a terminating trace for all sequence of input transitions:*

$$\mathcal{TR} \overset{def}{=} \neg\mathcal{RNT} = \{T \in \wp(\Sigma^{+\infty}) \mid \forall j \in T^i \,\exists \sigma \in T^+ : \sigma^i = j\} \tag{3}$$

Example 4. Let $\langle \Sigma_2, \tau_2 \rangle$ be the transition system from Example 2. For any sequence of input transitions in $(\Lambda_2)^i = \{\langle a, b\rangle, \langle a, c\rangle\}$, there is at least one terminating trace in Λ_2, i.e., abd for $\langle a, b\rangle$, and ac for $\langle a, c\rangle$. Thus, $\Lambda_2 \in \mathcal{TR}$. ◁

The trace semantics $\Lambda[\![P]\!]$ of a program P, fully describing the behavior of P, exactly characterizes Termination Resilience of P (and its negation, Robust Non-Termination), for an initial set of states $I \subseteq \Sigma$:

$$P \models_I \mathcal{TR} \Leftrightarrow \Lambda[\![P]\!]I \in \mathcal{TR} \Leftrightarrow \Lambda[\![P]\!]I \notin \mathcal{RNT} \Leftrightarrow P \not\models_I \mathcal{RNT}. \tag{4}$$

In the next two sections, we abstract the trace semantics Λ to a special-purpose sound and complete (but not computable) semantics Λ_{tr} that forgets details of the program behavior that are irrelevant for reasoning about Termination Resilience. Next, we further abstract this semantics into a sound but computable semantics $\Lambda^{\natural}_{\mathsf{tr}}$ that yields a static analysis for Termination Resilience.

3 Termination Resilience Semantics

We derive our concrete semantics Λ_{tr} by abstract interpretation of the trace semantics Λ. The abstraction eliminates program traces that are potentially branching to non-termination through different untrusted inputs, and proves Termination Resilience for the remaining traces by associating a well-founded quantity [31,13] to program states belonging to terminating traces. Specifically, we abstract the trace semantics $\Lambda \in \wp(\Sigma^{+\infty})$ into the *termination resilience semantics* $\Lambda_{\mathsf{tr}} : \Sigma \rightharpoonup \mathbb{O}$ (where $\mathbb{O}$ is the set of ordinals), which is the best (potential) ranking function [12,33] for the program states in the traces in Λ.

We define the termination resilience abstraction $\alpha_{\mathsf{tr}} : \wp(\Sigma^{+\infty}) \to (\Sigma \rightharpoonup \mathbb{O})$ to map sets of program traces to a (potential) ranking function as:

$$\alpha_{\mathsf{tr}}(T) \overset{\text{def}}{=} \alpha_{\mathsf{v}} \circ \overrightarrow{\alpha}(T) \tag{5}$$

where $\overrightarrow{\alpha} : \wp(\Sigma^{+\infty}) \to \wp(\Sigma \times \Sigma)$ extracts the smallest transition relation $\tau \subseteq \Sigma \times \Sigma$ that generates a given set of traces T: $\overrightarrow{\alpha}(T) \overset{\text{def}}{=} \{\langle s, s'\rangle \mid \exists \sigma, \sigma' : \sigma s s' \sigma' \in T\}$, and $\alpha_{\mathsf{v}} : \wp(\Sigma \times \Sigma) \to (\Sigma \rightharpoonup \mathbb{O})$ ranks the elements in the domain of r:

$$\alpha_{\mathsf{v}}(\emptyset) \overset{\text{def}}{=} \dot{\emptyset}$$

$$\alpha_{\mathsf{v}}(r)s \overset{\text{def}}{=} \begin{cases} 0 & s \in \Omega_r \\ \sup \left\{ \alpha_{\mathsf{v}}(r)s' + 1 \;\middle|\; \langle s, s'\rangle \in r^i \right\} & s \in \widetilde{\mathrm{pre}}_{r^i}(\mathsf{dom}(\alpha_{\mathsf{v}}(r))) \\ \inf \left\{ \alpha_{\mathsf{v}}(r)s' + 1 \;\middle|\; \mathsf{dom}(\alpha_{\mathsf{v}}(r)) \wedge \langle s, s'\rangle \in r^r \right\} & s \in \mathrm{pre}_{r^r}(\mathsf{dom}(\alpha_{\mathsf{v}}(r))) \\ \text{undefined} & \text{otherwise} \end{cases}$$

with $\dot{\emptyset}$ representing the totally undefined function. The (potential) ranking function is defined incrementally, starting from the final states in Ω_r, where the function has value 0 (and is undefined elsewhere). Then, the domain of the function grows by retracing the program (transitions) backwards and counting the number of performed program steps as value of the function. We model the nondeterminism arising from input transitions *demonically*, requiring all successor states to already be in the domain of the function ($s \in \widetilde{\mathrm{pre}}_{\tau^i}(\mathsf{dom}(\alpha_{\mathsf{v}}(r)))$), and count the *maximum* number of performed program steps. Instead, nondeterminism arising from regular transitions is treated *angelically*, and the potential ranking functions counts the minimum number of steps to termination.

We can now formally define the termination resilience semantics Λ_{tr}.

Definition 3 (Termination Resilience Semantics). *Let $\langle \Sigma, \tau \rangle$ be a transition system. Its termination resilience semantics $\Lambda_{\mathsf{tr}} \in (\Sigma \rightharpoonup \mathbb{O})$ is:*

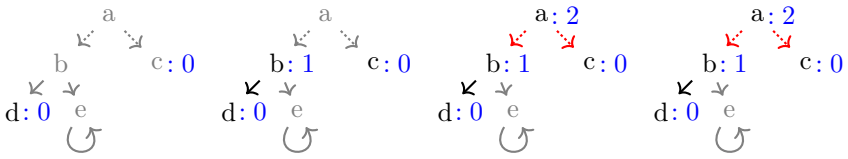
$$\Lambda_{\mathsf{tr}} \overset{def}{=} \alpha_{\mathsf{tr}}(\Lambda) = \mathsf{lfp}^{\sqsubseteq}_{\dot{\emptyset}} \Theta_{\mathsf{tr}}$$

$$\Theta_{\mathsf{tr}}(f) \overset{def}{=} \lambda s. \begin{cases} 0 & s \in \Omega_\tau \\ \mathsf{sup}\left\{ f(s') + 1 \,\middle|\, \langle s, s' \rangle \in \tau^i \right\} & s \in \widetilde{\mathrm{pre}}_{\tau^i}(\mathsf{dom}(f)) \\ \mathsf{inf}\left\{ f(s') + 1 \,\middle|\, s' \in \mathsf{dom}(f), \langle s, s' \rangle \in \tau^r \right\} & s \in \mathrm{pre}_{\tau^r}(\mathsf{dom}(f)) \\ \textit{undefined} & \textit{otherwise} \end{cases}$$

where $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ is the trace semantics (cf. Equation 2).

We write $\Lambda_{\mathsf{tr}}[\![P]\!]$ for the termination resilience semantics of a program P.

Example 5. Let $\langle \Sigma_2, \tau_2 \rangle$ and $\Lambda_{\mathsf{t}2}$ be the transition system from Example 2 and its termination resilience trace semantics, respectively. The fixpoint iterates of its termination resilience semantics $\Lambda_{\mathsf{tr}2} \overset{\text{def}}{=} \alpha_{\mathsf{tr}}(\Lambda_{\mathsf{t}2})$ are the following:



Note that $\Lambda_{\mathsf{tr}2}$ is defined over the state b even if a transition can lead to the diverging trace be^ω because the non-deterministic choice is treated angelically: there is at least one choice that ensures termination (via the trace bd). ◁

The termination resilience semantics $\Lambda_{\mathsf{tr}}[\![P]\!]$ of a program P is sound and complete for verifying Termination Resilience, for an initial set of states $I \subseteq \Sigma$:

Theorem 1 (Soundness&Completeness). $\Lambda[\![P]\!]I \in \mathcal{TR} \Leftrightarrow I \subseteq \mathsf{dom}(\Lambda_{\mathsf{tr}}[\![P]\!])$

```
1x := input;
2z := 10;

while 3(z >= 0) {
  4z := z - x;
}5
```

(a) Program P_1

```
1x = input;
2z := 10;
3c := [-2, 1];
while 4(z >= x) {
  5z := z + c;
}6
```

(b) Program P_2

Fig. 3: Programs akin to the transitions systems in Example 1 (a) and 2 (b).

4 Denotational Termination Resilience Semantics

The formal treatment given in the previous section is language independent. In the following, for simplicity, we introduce a small sequential programming language that we use for illustration throughout the rest of the paper:

$$\begin{aligned} \text{AExp} \ni \mathtt{a} &::= x \mid c \mid -\mathtt{a} \mid \mathtt{a} \diamond \mathtt{a} \\ \text{Stmt} \ni \mathtt{s} &::= {}^{l}\,\mathsf{skip} \mid {}^{l}\,x := \mathsf{input} \mid {}^{l}\,x := [c_1, c_2] \mid {}^{l}\,x := \mathtt{a} \\ &\quad \mid \mathsf{if}\,{}^{l}\,\mathtt{a} \bowtie 0\,\{\mathtt{s}\} \mid \mathsf{while}\,{}^{l}\,\mathtt{a} \bowtie 0\,\{\mathtt{s}\} \mid \mathtt{s}; \mathtt{s} \\ \text{Prog} \ni \mathtt{p} &::= \mathtt{s}^{l} \end{aligned}$$

where $x \in \mathbb{X}$ ranges over program variables, $c \in \mathbb{Z}$, $c_1 \in \mathbb{Z} \cup \{-\infty\}$, $c_2 \in \mathbb{Z} \cup \{+\infty\}$ range over mathematical integer values possibly extended to include (negative or positive) infinity, $\diamond \in \{+, -, *, \dots\}$, and $\bowtie \in \{=, \neq, \leq, \dots\}$. A unique label $l \in \mathcal{L}$ appears within each instruction statement. Variable assignments can involve an untrusted input (${}^{l}\,x := \mathsf{input}$), a (trusted) non-deterministic choice (${}^{l}\,x := [c_1, c_2]$) or an arithmetic expression (${}^{l}\,x := \mathtt{a}$). A program is an instruction statement $\mathtt{s} \in \text{Stmt}$ followed by a label $l \in \mathcal{L}$.

Figure 3 shows two programs written in our small language, both slices of program P from Figure 1. Program P_1 (cf. Figure 3a) is akin to the transition system in Example 1: the input $x \leq 0$ leads to non-termination (state c in Figure 2a), while for $x > 0$ (state b) the program always terminates. Instead, P_2 (cf. Figure 3b) is akin to Example 2: for the input $x \leq 10$ (state b in Figure 2b), termination is possible with $c < 0$ (state d).

In this section, we instantiate the definition of our termination resilience semantics Λ_{tr} with respect to programs $\mathtt{p} \in \text{Prog}$ written in our small language. Then, in the next section, we will define its abstraction $\Lambda^{\natural}_{\mathsf{tr}}$.

Taint Semantics. Let $\mathcal{E} = \mathbb{X} \to \mathbb{Z}$ denote the set of environments, where each $\rho \in \mathcal{E}$ maps variables to their integer values. We adopt the standard semantics of expressions $[\![\mathtt{e}]\!] \colon \mathcal{E} \rightharpoonup \mathcal{P}(\mathbb{Z})$ mapping an expression $\mathtt{e} \in \text{AExp} \cup \{\mathsf{input}, [c_1, c_2]\}$ to the set of all its possible values in a given environment.

Program states are pairs $\mathcal{L} \times \mathcal{E}$ of a label $l \in \mathcal{L}$ and an environment $\rho \in \mathcal{E}$ defining the values of the program variables at the program point designated

$$
\begin{aligned}
\mathfrak{T}[\![^{l}\,\mathsf{skip}]\!]\mathrm{T} &\stackrel{\text{def}}{=} \mathrm{T}\\
\mathfrak{T}[\![^{l}\,x := \mathsf{input}]\!]\mathrm{T} &\stackrel{\text{def}}{=} \mathrm{T} \cup \{x\}\\
\mathfrak{T}[\![^{l}\,x := [c_1, c_2]]\!]\mathrm{T} &\stackrel{\text{def}}{=} \mathrm{T} \setminus \{x\}\\
\mathfrak{T}[\![^{l}\,x := \mathsf{a}]\!]\mathrm{T} &\stackrel{\text{def}}{=} \begin{cases} \mathrm{T} \cup \{x\} & \mathfrak{T}[\![\mathsf{a}]\!]\mathrm{T}\\ \mathrm{T} \setminus \{x\} & \text{otherwise} \end{cases}\\
\mathfrak{T}[\![\mathsf{if}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathtt{s}\}]\!]\mathrm{T} &\stackrel{\text{def}}{=} \begin{cases} \mathfrak{T}[\![\mathtt{s}]\!]\mathrm{T} \cup \textsc{assigned}(\mathtt{s}) \cup \mathrm{T} & \mathfrak{T}[\![\mathsf{a}]\!]\mathrm{T}\\ \mathfrak{T}[\![\mathtt{s}]\!]\mathrm{T} \cup \mathrm{T} & \text{otherwise} \end{cases}\\
\mathfrak{T}[\![\mathsf{while}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathtt{s}\}]\!]\mathrm{T} &\stackrel{\text{def}}{=} \mathsf{lfp}^{\subseteq}_{\mathrm{T}}\,\mathfrak{T}[\![\mathsf{if}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathtt{s}\}]\!]\\
\mathfrak{T}[\![\mathtt{s}_1; \mathtt{s}_2]\!]\mathrm{T} &\stackrel{\text{def}}{=} \mathfrak{T}[\![\mathtt{s}_2]\!](\mathfrak{T}[\![\mathtt{s}_1]\!]\mathrm{T})
\end{aligned}
$$

Fig. 4: Taint semantics $\mathfrak{T}[\![\mathtt{s}]\!] : \mathcal{P}(\mathbb{X}) \to \mathcal{P}(\mathbb{X})$, where $\textsc{assigned}(\mathtt{s})$ is the set of variables assigned with an arithmetic expression within the statement $\mathtt{s}$.

by l. To track the effect of input transitions on program states, we need a *taint semantics* $\mathfrak{T}[\![\mathtt{s}]\!] : \mathcal{P}(\mathbb{X}) \to \mathcal{P}(\mathbb{X})$ that collects variables that depend on untrusted inputs, i.e., that are *tainted*, after the execution of a program statement (cf. Figure 4). The taint semantics $\mathfrak{T}[\![\mathsf{a}]\!] \colon \mathcal{P}(\mathbb{X}) \to \{\mathsf{true}, \mathsf{false}\}$ of expressions is:

$$
\mathfrak{T}[\![\mathsf{a}]\!]\mathrm{T} \stackrel{\text{def}}{=} \exists \rho \in \mathcal{E} \colon \exists V \in \mathbb{Z}^{|\mathbb{X}|} \colon V \in [\![\mathsf{a}]\!]\rho \wedge \forall \rho' \neq_{\mathrm{T}} \rho \colon V \notin [\![\mathsf{a}]\!]\rho' \tag{6}
$$

where $\rho' \neq_{\mathrm{T}} \rho \Leftrightarrow \exists \emptyset \subset \mathrm{T}' \subseteq \mathrm{T} : (\forall x \notin \mathrm{T}' \colon \rho(x) = \rho'(x)) \wedge (\forall x \in \mathrm{T}' \colon \rho(x) \neq \rho'(x))$ denotes program environments that differ only on (a non-empty subset of) the tainted variables in T. An expression is tainted if there exists a value of its semantics $[\![\mathsf{a}]\!]$ that is only possible in an environment ρ with a certain value V for the set T of tainted variables, i.e., the value of the expression depends on untrusted program inputs. Note that this definition differs from a classical non-interference-like definition (e.g., $\mathfrak{T}[\![\mathsf{a}]\!]\mathrm{T} \stackrel{\text{def}}{=} \forall \rho, \rho' \in \mathcal{E} \colon \forall x \notin \mathrm{T} \colon \rho(x) = \rho'(x) \wedge [\![\mathsf{a}]\!]\rho \neq [\![\mathsf{a}]\!]\rho'$) to also take into account the non-determinism arising from trusted variables (i.e., $[\![\mathsf{a}]\!]\rho \neq [\![\mathsf{a}]\!]\rho'$ could be the consequence of non-deterministic value choices rather than untrusted inputs). For simplicity, in this definition we deliberately do not take into account the reachable environments at each program label. A more precise definition would be straightforward but needlessly cumbersome for our purposes. The taint semantics $\mathfrak{T}[\![\mathtt{p}]\!] \in \mathcal{P}(\mathbb{X})$ of a program $\mathtt{p} \in \mathrm{Prog}$ is $\mathfrak{T}[\![\mathtt{p}]\!] = \mathfrak{T}[\![\mathtt{s}^{l}]\!] \stackrel{\text{def}}{=} \mathfrak{T}[\![\mathtt{s}]\!]\emptyset$. By pointwise-lifting of $\mathfrak{T}$ with respect to the program labels $\mathcal{L}$, we obtain the lifted taint semantics $\dot{\mathfrak{T}} \colon \mathcal{L} \to \mathcal{P}(\mathbb{X})$ mapping each label to the set of variables that are can be tainted at that label.

Denotational Termination Resilience Semantics. We can now define in Figure 5 the termination resilience semantics $\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!] : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$ for

$$\Lambda_{\mathsf{tr}}[\![{}^{l}\,\mathtt{skip}]\!]f \stackrel{\text{def}}{=} \lambda\rho \in dom(f).f(\rho)+1$$

$$\Lambda_{\mathsf{tr}}[\![{}^{l}\,x := \mathsf{e}]\!]f \stackrel{\text{def}}{=} \lambda\rho.\begin{cases}\sup\{f(\rho[x \leftarrow v])+1 \mid v \in [\![\mathsf{e}]\!]\rho\} & \mathsf{e} = \mathsf{input} \vee A\\ \inf\left\{f(\rho[x \leftarrow v])+1 \;\middle|\; \begin{array}{c} v \in [\![\mathsf{e}]\!]\rho \wedge \\ \rho[x \leftarrow v] \in \mathsf{dom}(f)\end{array}\right\} & \mathsf{e} = [c_1, c_2] \vee B\\ \text{undefined} & \text{otherwise}\end{cases}$$

$$A \Leftrightarrow (\mathsf{e} = \mathsf{a} \wedge \mathfrak{T}[\![\mathsf{a}]\!](\dot{\mathfrak{T}}(l)) \wedge \exists v \in [\![\mathsf{a}]\!]\rho \wedge \forall v \in [\![\mathsf{a}]\!]\rho, \rho[x \leftarrow v] \in \mathsf{dom}(f))$$

$$B \Leftrightarrow (\mathsf{e} = \mathsf{a} \wedge \neg\mathfrak{T}[\![\mathsf{a}]\!](\dot{\mathfrak{T}}(l)) \wedge \exists v \in [\![\mathsf{a}]\!]\rho : \rho[x \leftarrow v] \in \mathsf{dom}(f))$$

$$\Lambda_{\mathsf{tr}}[\![\mathtt{if}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathtt{s}\}]\!]f \stackrel{\text{def}}{=} \Theta_{\mathsf{tr}}(f)$$

$$\Theta_{\mathsf{tr}} \stackrel{\text{def}}{=} \lambda X.\lambda\rho.\begin{cases}\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]X(\rho)+1 & \forall v \in [\![\mathsf{a}]\!]\rho : v \bowtie 0 \wedge \rho \in \mathsf{dom}(\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]X(\rho))\\ f(\rho)+1 & \forall v \in [\![\mathsf{a}]\!]\rho : v \not\bowtie 0 \wedge \rho \in \mathsf{dom}(f(\rho))\\ \text{undefined} & \text{otherwise}\end{cases}$$

$$\Lambda_{\mathsf{tr}}[\![\mathtt{while}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathtt{s}\}]\!]f \stackrel{\text{def}}{=} \mathsf{lfp}^{\sqsubseteq}_{\dot{\emptyset}}\,\Theta_{\mathsf{tr}}$$

$$\Lambda_{\mathsf{tr}}[\![\mathtt{s}_1;\mathtt{s}_2]\!]f \stackrel{\text{def}}{=} \Lambda_{\mathsf{tr}}[\![\mathtt{s}_1]\!](\Lambda_{\mathsf{tr}}[\![\mathtt{s}_2]\!]f)$$

Fig. 5: Termination resilience semantics $\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!] : (\mathcal{E} \rightharpoonup \mathbb{O}) \to (\mathcal{E} \rightharpoonup \mathbb{O})$. Since $[\![\mathsf{a}]\!]\rho$ is always a singleton $\{v\}$, we abuse notation and write $[\![\mathsf{a}]\!]\rho$ to directly denote v.

each program statement. Each transfer function $\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]$ takes as input a (potential) ranking function that is valid *after* the execution of the statement $\mathtt{s}$ and returns a (potential) ranking function that is valid *before* the execution of $\mathtt{s}$.

If the right-hand side of an assignment is an untrusted input (${}^{l}\,x := \mathsf{input}$) or an expression a (${}^{l}\,x := \mathsf{a}$) that depends on tainted variables ($\mathfrak{T}[\![\mathsf{a}]\!](\dot{\mathfrak{T}}(l))$), the termination resilience semantics $\Lambda_{\mathsf{tr}}[\![{}^{l}\,x := \mathsf{a}]\!]$ returns a (potential) ranking function only defined over the environments ρ that, when subject to the assignment ($\rho[x \leftarrow v]$ with $v \in [\![\mathsf{a}]\!]\rho$), *always* belong to the domain of the given (potential) ranking function f (case A). The value of f for these environments is increased by one, to take into account another program execution step before termination, and the value of the resulting ranking function is the least upper bound of these values (to count the maximum number of steps to termination). Otherwise – if the right-hand side of the assignment is a non-deterministic choice (${}^{l}\,x := [c_1, c_2]$) or an expression a that does not depend on tainted variables ($\neg\mathfrak{T}[\![\mathsf{a}]\!](\dot{\mathfrak{T}}(l))$) – the resulting function is defined over the environments ρ for which *at least one* value $v \in [\![\mathsf{a}]\!]\rho$ yields an environment $\rho[x \leftarrow v]$ in $\mathsf{dom}(f)$ (case B). Its value is the greatest lower bound of the value of f on these environments plus one (to count the minimum number of steps to termination).

The termination resilience semantics $\Lambda_{\mathsf{tr}}[\![\mathtt{if}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathtt{s}\}]\!]$ for `if` statements, via Θ_{tr}, defines a potential ranking function over all environments in $\mathsf{dom}(\Lambda_{\mathsf{tr}}[\![\mathtt{s}]\!]f)$ or $\mathsf{dom}(f)$, i.e., all environments potentially leading to a terminating execution, depending on whether they satisfy or not satisfy the guard $\mathsf{a} \bowtie 0$. The termination

resilience $\Lambda_{\mathsf{tr}}[\![\texttt{while}^{\,l}\,\mathsf{a} \bowtie 0\,\{\mathsf{s}\}]\!]$ for `while` statements is the least fixpoint of Θ_{tr} with respect to the computational order $f_1 \sqsubseteq f_2 \overset{\text{def}}{\Leftrightarrow} \mathsf{dom}(f_1) \subseteq \mathsf{dom}(f_2) \wedge \forall x \in \mathsf{dom}(f_1) : f_1(x) \leq f_2(x)$.

The termination resilience semantics $\Lambda_{\mathsf{tr}}[\![\mathsf{p}]\!] \in (\mathcal{E} \rightharpoonup \mathbb{O})$ of a program $\mathsf{p} \in \text{Prog}$ is determined by $\Lambda_{\mathsf{tr}}[\![\mathsf{s}]\!]$ taking as input the constant function equal to zero.

Definition 4 (Denotational Termination Resilience Semantics). *Let* $\mathsf{p} \in \text{Prog}$ *be a program. Its denotational termination resilience semantics is:*

$$\Lambda_{\mathsf{tr}}[\![\mathsf{p}]\!] = \Lambda_{\mathsf{tr}}[\![\mathsf{s}^l]\!] \overset{def}{=} \Lambda_{\mathsf{tr}}[\![\mathsf{s}]\!](\lambda\rho.0). \tag{7}$$

5 Abstract Termination Resilience Semantics

In this section, we propose a sound abstraction $\Lambda^\natural_{\mathsf{tr}}[\![\mathsf{p}]\!]$ of the denotational termination resilience semantics $\Lambda_{\mathsf{tr}}[\![\mathsf{p}]\!]$ defined in Section 4 by leveraging (and extending) Urban and Miné's decision tree numerical abstract domain $\mathcal{T}$ [34] to abstract potential ranking functions by means of piecewise-defined partial functions. Specifically, we consider the concretization-based abstraction [11] $\langle \mathcal{E} \rightharpoonup \mathbb{O}, \preceq \rangle \xleftarrow{\gamma} \langle \mathcal{T}, \preceq_T \rangle$ where the approximation order $\preceq$ is defined as follows:

$$f_1 \preceq f_2 \Leftrightarrow \mathsf{dom}(f_1) \supseteq \mathsf{dom}(f_2) \wedge \forall x \in \mathsf{dom}(f_2) \colon f_1(x) \leq f_2(x). \tag{8}$$

We define $\Lambda^\natural_{\mathsf{tr}}[\![\mathsf{p}]\!]$ so $\Lambda_{\mathsf{tr}}[\![\mathsf{p}]\!] \preceq \gamma(\Lambda^\natural_{\mathsf{tr}}[\![\mathsf{p}]\!])$ meaning that $\Lambda^\natural_{\mathsf{tr}}[\![\mathsf{p}]\!]$ *over-approximates* the value of $\Lambda_{\mathsf{tr}}[\![\mathsf{p}]\!]$ and *under-approximates* its domain $\mathsf{dom}(\Lambda_{\mathsf{tr}}[\![\mathsf{p}]\!])$. In this way, the abstraction $\Lambda^\natural_{\mathsf{tr}}[\![\mathsf{p}]\!]$ provides *sufficient preconditions* for termination resilience.

Remark 1. Note that Urban and Miné's abstract termination semantics [34] — let us denote it $\Lambda^\natural_{\mathsf{t}}[\![\mathsf{p}]\!]$ — is already a sound but coarse over-approximation of $\Lambda^\natural_{\mathsf{tr}}[\![\mathsf{p}]\!]$: $\Lambda_{\mathsf{tr}}[\![P]\!] \preceq \gamma(\Lambda^\natural_{\mathsf{t}}[\![\mathsf{p}]\!])$. Our objective in this section is to build upon and relax $\Lambda^\natural_{\mathsf{t}}[\![\mathsf{p}]\!]$ for reasoning about Termination Resilience.

5.1 Decision Tree Abstract Domain.

In the following, we recall the decision tree abstract domain only insofar as it is necessary for our analysis. We do not redefine aspects of the domain that remain unchanged, and refer to [33] for the full formal definitions.

An element $t \in \mathcal{T}$ is a piecewise-defined partial function represented by a full binary tree, where each node — denoted $\text{NODE}\{c\}\colon t_1, t_2$, with $t_1, t_2 \in \mathcal{T}$ — is labeled by a constraint $c \in \mathcal{C}$ over the program variables (where $\mathcal{C}$ is the set of allowed constraints, e.g., linear constraints), and each leaf — denoted $\text{LEAF}\colon f$ — is labeled by a function $f \in \mathcal{F}$ of the program variables (where $\mathcal{F}$ is the set of allowed functions, e.g., affine functions). Nodes recursively partition the space of possible values of the program variables: constraint labeling nodes are satisfied by the left subtrees of the nodes, while the right subtrees satisfy their negation. The leaves of the tree represent the value of the function within each partition.

Algorithm 1 Decision Tree Assignment

```
1: function ASSIGN_T⟦l x := e⟧(t)
2:     if isLeaf(t) then
3:         return LEAF: ASSIGN_F⟦l x := e⟧(t.f)
4:     else if isNode(t) then
5:         I ← ASSIGN_C⟦l x := e⟧(t.c)
6:         J ← ASSIGN_C⟦l x := e⟧(¬t.c)
7:         if isEmpty(I) ∧ isEmpty(J) then
8:             return ASSIGN_T⟦l x := e⟧(t.l) ϒ_T ASSIGN_T⟦l x := e⟧(t.r)
9:         else if isEmpty(I) ∧ ⊥_C ∈ J then
10:            return ASSIGN_T⟦l x := e⟧(t.l)
11:        else if ⊥_C ∈ I ∧ isEmpty(J) then
12:            return ASSIGN_T⟦l x := e⟧(t.r)
13:        else
14:            t_1 ← PRUNE_T(ASSIGN_T⟦l x := e⟧(t.l), I)
15:            t_2 ← PRUNE_T(ASSIGN_T⟦l x := e⟧(t.r), J)
16:            return t_1 ϒ_T t_2
```

Figure 6 shows an example of a decision tree. The leaf with value $\bot_F$ explicitly represents the undefined partition of the (partial) function. Undefined functions can also be represented by the leaf value $\top_F$ in case of an irrecoverable loss of precision of the analysis [8]. The decision tree in Figure 6 is automatically inferred by our termination resilience static analysis at label 4 of program P_2 in Figure 3b. It represents a function with constant value 1 when $z < x$ (at most one step to termination), with constant value 3 when $z \geq x \wedge z + c < x$ (at most three steps to termination) and undefined otherwise (termination is not proved when $z + c \geq x$), while the concrete termination resilience semantics at the same label 4 is defined when $z < x$ or $c < 0$: the decision tree is thus a sound approximation with respect to $\preceq$ (cf. Equation 8).

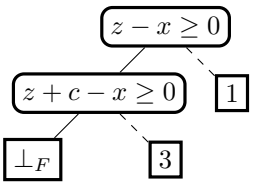
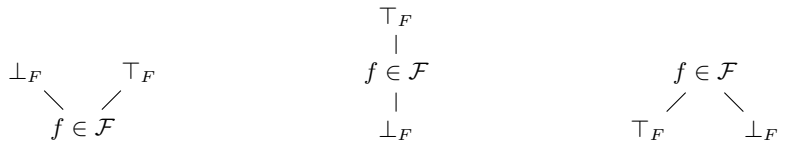


Fig. 6: Decision tree inferred by our analysis at label 4 of program P_2 (cf. Figure 3b).

The partitioning induced by decision tree constraints is dynamic: the analysis begins at the end of the program with the decision tree LEAF: 0 (i.e., zero program executions steps to termination) and proceeds *backwards*; during the analysis, constraints are added by boolean conditions or when merging control flows, and are modified by variable assignments.

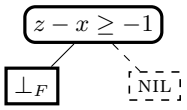
Algorithm 1 shows the decision tree assignment operator $\text{ASSIGN}_T\llbracket^l x := \mathsf{e}\rrbracket$ originally defined in [33]. The assignment is performed by recursively descending the given decision tree t (cf. Lines 14 and 15, the left subtree is accessed with $t.l$ and the right one with $t.r$). At the leaves, the assignment is handled by the auxiliary operator $\text{ASSIGN}_F\llbracket^l x := \mathsf{e}\rrbracket$ operating on functions (accessed by $t.f$, cf. Line 3): after the assignment, the value of defined functions is increased by one to account for one more step to termination. For example, let us consider the



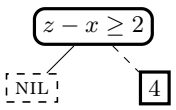
(a) Approximation order $\preceq_F$ (b) Computational order $\sqsubseteq_F$ (c) Resilience order $\leq_F$

Fig. 7: Hasse diagrams for difference orders of decision tree leaves.

decision tree in Figure 6 and the assignment $c := [-2, 1]$ at label 3 in Figure 3b. The result of the assignment on LEAF : 3 is LEAF : 4, simply increasing the value of its function by one, while LEAF : $\bot_F$ remains unchanged. Node constraints (accessed with $t.c$) and their negation ($\neg t.c$) are handled by the standard assignment operator $\text{ASSIGN}_C[\![^l\, x := \mathsf{e}]\!]$ of an auxiliary underlying numerical abstract domain used to manage constraints in $\mathcal{C}$, which produces a set $I \in \mathcal{P}(\mathcal{C})$ (cf. Line 5) and a set $J \in \mathcal{P}(\mathcal{C})$ (cf. Line 6) of linear constraint conjuncts resulting from (over-approximating) the assignment to $t.c$ and $\neg t.c$, respectively. For example, considering again the decision tree in Figure 6, the result of the assignment $c := [-2, 1]$ on the constraint $z + c - x \geq 0$ and its negation $z + c - x < 0$ yields $I = \{z - x \geq -1\}$ and $J = \{z - x < 2\}$. The set of constraints I and J are added (by PRUNE_T) to the subtrees resulting from the recursive calls (cf. Lines 14 and 15). In our example, the pruning of the LEAF : $\bot_F$ with the set of constraints I results in the decision tree shown in Figure 8a, while the pruning of LEAF : 4 with J results in the tree in Figure. 8b.



(a) Pruning of LEAF: $\bot_F$ with $I = \{z - x \geq -1\}$.



(b) Pruning of LEAF: 4 with $J = \{z - x < 2\}$.

Fig. 8: Decision trees resulting from PRUNE_T.

The special NIL node denotes partitions that are outside the domain of definition of the decision tree. Note that it differs from leaves labeled with $\bot_F$ or $\top_F$, which denote undefined partitions within the domain of definition of the tree. Finally, the resulting trees are joined by the approximation join $\curlyvee_T$ (cf. Line 16), which merges the domains of definitions of the joined trees (essentially removing any NIL nodes) and retains the leaves labeled with an undefined function ($\bot_F$ or $\top_F$) in either of the joined decision trees, cf. the Hasse diagram in Figure 7a. Continuing the example, joining the trees in Figure 8 yields the tree in Figure 9.

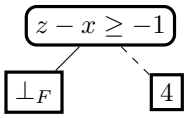


Fig. 9: Approximation join of Figure 8a and Figure 8b.

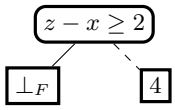


Fig. 10: Resilience join of Figure 8a and Figure 8b.

In case both I and J are empty (cf. Line 7), neither the constraint $t.c$ nor its negation $\neg t.c$ exists anymore and thus the subtrees resulting from the recursive calls are joined by the approximation join $\curlyvee_T$. In case I is empty and J is an unsatisfiable set of constraints (i.e., the unsatisfiable constraint $\bot_C$ belongs to J, cf. Line 9), it means that $\neg t.c$ is no longer satisfiable and thus only the left subtree of the decision tree is kept (cf. Line 10). Similarly, in case I is an unsatisfiable set of constraints and J is empty (cf. Line 11), only the right subtree of the decision tree is kept (cf. Line 12).

To minimize the cost of the analysis and to enforce its termination, a widening operator $\triangledown_T$ limits the height of the decision trees and the number of induced partitions. Abstract fixpoint iterations with widening follow the computational order $\sqsubseteq_T$, which preserves decision tree leaves labeled with a defined function over undefined leaves labeled with $\bot_F$ (i.e., the analysis grows the domain of the inferred ranking function at each iteration), but preserves leaves labeled with $\top_F$ over all other leaves (i.e., the domain of the inferred function shrinks when the analysis loses too much precision), cf. Figure 7b.

5.2 Non-Deterministic Assignments.

To leverage the decision tree abstract domain for termination resilience we need to extend it to distinguish between variables assigned with untrusted or (trusted) non-deterministic values. Termination must be ensured for *all* possible values of untrusted variables, while *any* non-deterministic value that ensures termination is enough to satisfy termination resilience. We thus introduce a new *resilience join* operator $\veebar_T$ between decision trees that retains the leaves labeled with a *defined* function in either of the joined decision trees, cf. Figure 7c.

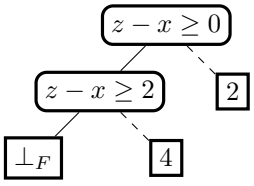


Fig. 11: Decision tree inferred at label 3 of program P_2.

We implement $\textsc{assign}_T[\![^l\, x := [c_1, c_2]]\!]$ for non-deterministic assignments following Algorithm 1, but using the resilience join $\veebar_T$ instead of the approximation join $\curlyvee_T$ at Lines 8 and 16. Let us consider again the left subtree in Figure 6 and the non-deterministic variable assignment $c := [-2, 1]$ at label 3 in Figure 3b. Joining the trees in Figure 8 resulting from the $\textsc{prune}_T$ at Lines 14 and 15 yields the decision tree in Figure 10, which indicates that the termination of program P_2 is possible when $z - x < 2$, for at least *some* values of c chosen at label 3 (notably,

for $c \in \{-2, -1\}$, but the analysis does not explicitly provide these values). The final decision tree at program label 3 is shown in Figure 11.

Note that this seemingly anodyne modification is actually insidious. Let us consider the decision tree in Figure 12 and the variable assignment $x := [-\infty, +\infty]$, and assume that the variable y is untrusted. The result of the assignment on the decision tree constraints is the empty set of constraints and, since the variable x is assigned with a non-deterministic value, we can employ the resilience join $\veebar_F$ to join all leaves after the assignment is performed on them. This yields the final decision tree LEAF: 2 (obtained from LEAF: 1 after the assignment), which is not sound! In fact, the sufficient precondition for Termination Resilience given by the decision tree in Figure 12 requires $y > 0$ but this constraint is missing from the decision tree because it is redundant. As a consequence, when the assignment is performed and the other constraints $x < 0$ and $x + y \geq 0$ disappear, the result is an unsound decision tree indicating that termination resilience always holds. To avoid this issue, before non-deterministic variable assignments, decision trees need to be *filled* to explicitly contain *at least* the strongest redundant univariate constraints over the untrusted program variables. We further modify Algorithm 1 to insert a call to $\text{FILL}_T(t)$ at the very beginning (*before* Line 2), where FILL_T leverages the underlying numerical abstract domain used to manage $\mathcal{C}$ to find these redundant univariate constraints and adds them to the decision tree t. Additional redundant constraints may be added, though this needlessly increases the analysis time.

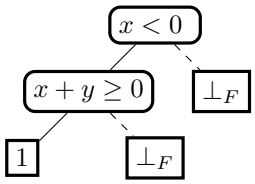


Fig. 12: Unfilled decision tree.

The ASSIGN_T operator for non-deterministic variable assignments, implemented by Algorithm 1 modified as discussed above, is a sound over-approximation of the termination resilience semantics (cf. Figure 5):

Proposition 1. $\Lambda_{\text{tr}}[\![^l\, x := [c_1, c_2]]\!]\gamma(t) \preceq \gamma(\textit{ASSIGN}_T[\![^l\, x := [c_1, c_2]]\!]t)$

Resilience Join and Non-Tainted Program Variables. At this point, one may be tempted to extend the use of the resilience join to all variable assignments with a non-tainted right-hand side expression (i.e., an expression that does not depend on untrusted program variables) or, similarly, to `if` and `while` statements with non-tainted boolean conditions. However, this change alone would be unsound. Consider program P in Figure 1 and let the boolean condition at label 3 be $a * a \geq 0$ where a is assigned with a non-deterministic value before x. Figure 13 shows the most precise decision trees that can be inferred at label 4 and 6.
At program label 3, the boolean condition $a * a \geq 0$ cannot be precisely represented by a (set of) linear constraints. Thus, no further constrains are added before joining the trees. Since the decision tree at label 6 is totally defined (cf. Figure 13b), using the resilience join would also yield a decision tree that is totally defined (cf. Figure 7c), implying that termination resilience always holds.

(a) Decision tree at label 4. (b) Decision tree at label 6.

Fig. 13: Abstract termination resilience semantics for program P.

$$\Lambda^\natural_{\mathsf{tr}}[\![^l \text{ skip}]\!]t \stackrel{\text{def}}{=} \text{SKIP}_T(t)$$

$$\Lambda^\natural_{\mathsf{tr}}[\![^l\, x := \mathsf{e}]\!]t \stackrel{\text{def}}{=} \text{ASSIGN}_T[\![^l\, x := \mathsf{e}]\!](t)$$

$$\Lambda^\natural_{\mathsf{tr}}[\![\text{if}^{\,l}\, \mathsf{a} \bowtie 0\, \{\mathtt{s}\}]\!]t \stackrel{\text{def}}{=} \Theta^\natural_{\mathsf{tr}}(t)$$

$$\Theta^\natural_{\mathsf{tr}} \stackrel{\text{def}}{=} \lambda X.\text{FILTER}_T[\![\mathsf{a} \bowtie 0]\!](\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}]\!]X) \curlyvee_T \text{FILTER}_T[\![\mathsf{a} \not\bowtie 0]\!](t)$$

$$\Lambda^\natural_{\mathsf{tr}}[\![\text{while}^{\,l}\, \mathsf{a} \bowtie 0\, \{\mathtt{s}\}]\!]t \stackrel{\text{def}}{=} \text{LFP}^\natural\ \Theta^\natural_{\mathsf{tr}}$$

$$\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}_1; \mathtt{s}_2]\!]t \stackrel{\text{def}}{=} \Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}_2]\!](\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}_1]\!]t)$$

Fig. 14: Abstract Program Semantics for Termination Resilience

However, program label 6 is not reachable and, indeed, termination resilience does not hold: the program is robustly non-terminating when $x \leq 0$.

To ensure soundness in the general case of non-tainted variables and expressions, the resilience join should retain leaves labeled with a defined function only when the domain partition that they represent is actually reachable. In practice, we ensure this by conservatively under-approximating the weakest liberal precondition of program statements by means of the approximation join [33]. We leave finding less conservative approximations for future work.

5.3 Abstract Termination Resilience Semantics.

We can finally define in Figure 14 the *abstract termination resilience semantics* $\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{s}]\!] : \mathcal{T} \to \mathcal{T}$ for each program statement $\mathtt{s}$. The SKIP_T operator simply increases by one the value of the functions labeling the leaves of the decision tree. The assignment operator $\text{ASSIGN}_T[\![^l\, x := \mathsf{e}]\!]$ is implemented by Algorithm 1, modified as discussed in Section 5.2 for non-deterministic variable assignments. The semantics of conditional `if` statements $\Theta^\natural_{\mathsf{tr}}$ employs the approximation join $\curlyvee_T$ as discussed above, after handling the boolean conditions with the FILTER_T operator (due to space limitations, we refer to [33] for its definition). The semantics for `while` loops iterates $\Theta^\natural_{\mathsf{tr}}$ with widening starting from the totally undefined decision tree $\text{LEAF} : \bot_F$ until an abstract fixpoint is reached.

The *abstract termination resilience semantics* $\Lambda^\natural_{\mathsf{tr}}[\![\mathtt{p}]\!] \in \mathcal{T}$ for a $\mathtt{p} \in \text{Prog}$ starts the termination resilience analysis with a decision tree containing a single leaf labeled with the zero function:

Definition 5 (Abstract Termination Resilience Semantics). *Let* $\mathtt{p} \in \mathit{Prog}$ *be a program. Its abstract termination resilience semantics is:*

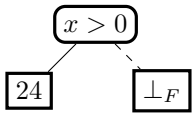
$$\Lambda^{\natural}_{\mathsf{tr}}[\![\mathtt{p}]\!] = \Lambda^{\natural}_{\mathsf{tr}}[\![\mathtt{s}^l]\!] \stackrel{def}{=} \Lambda^{\natural}_{\mathsf{tr}}[\![\mathtt{s}]\!](\mathit{LEAF} : 0). \tag{9}$$

The termination resilience analysis is sound with respect to the approximation order $\preceq$ (cf. Equation 8):

Theorem 2 (Soundness). $\Lambda_{\mathsf{tr}}[\![P]\!] \preceq \gamma(\Lambda^{\natural}_{\mathsf{tr}}[\![P]\!])$

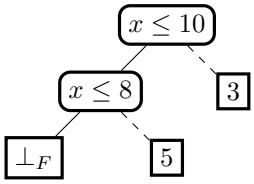
Corollary 1 (Soundness). $\Lambda[\![P]\!]I \in \mathcal{TR} \Leftarrow I \subseteq \mathsf{dom}(\gamma(\Lambda^{\natural}_{\mathsf{tr}}[\![P]\!]))$

Example 6. Let us consider again P_1 in Figure 3a. Its most precise termination resilience semantics at label 3 matches the semantics of program P at label 4 shown in Figure 13a. The assignment at label 2 – substituting z with 10 and incrementing the value of the functions labeling the defined leaves – yields



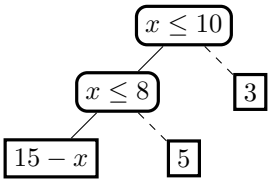
The assignment at label 1 yields $\textsc{leaf} : \bot_F$. Its concretization $\gamma(\textsc{leaf}\colon \bot_F)$ is the totally undefined function $\dot{\emptyset}$. We are thus unable to prove termination resilience. In this case, this is a true positive since indeed $\Lambda_{\mathsf{tr}}[\![P_1]\!] \notin \mathcal{TR}$. ◁

Example 7. Let us consider again program P_2 in Figure 3b. Its abstract termination resilience semantics at program label 3 is shown in Figure 11. The assignment at label 2 yields



and the assignment at label 1 yields $\textsc{leaf} : \bot_F$. We are again unable to prove termination resilience since $\gamma(\textsc{leaf}\colon \bot_F)$ is the totally undefined function $\dot{\emptyset}$. This, however, is a false alarm since, in fact, $\Lambda_{\mathsf{tr}}[\![P_2]\!] \in \mathcal{TR}$.

Let us consider instead the most precise abstract termination resilience semantics shown in Figure 13b. The assignment at label 2 yields



and the assignment at label 1 then yields $\textsc{leaf} : \omega$. In this case, we prove termination resilience since $\gamma(\textsc{leaf}\colon \omega)$ yields $\lambda\rho.\omega$, which is exactly $\Lambda_{\mathsf{tr}}[\![P_2]\!]$. ◁

6 Implementation and Experimental Evaluation

Implementation. We implemented our termination resilience analysis in the open-source tool FuncTion-TR[26]. FuncTion-TR is implemented in OCaml and relies on the Apron numerical abstract domain library [18] to manage decision tree constraints and functions labeling the tree leaves. It is also possible to activate the extension to lexicographic ranking functions [35], and tune the precision of the analysis by adjusting the widening delay. We use a simple implementation of the FILL_T operation (cf. Section 5.2), which adds constraints, obtained from Apron's interval domain, bounding the values of the variables assigned with a non-deterministic value.

The analysis takes as input a numerical program written in a C-like syntax and outputs TRUE if all leaves of the inferred decision tree for the program are labelled with defined functions, i.e., Termination Resilience holds for any initial program state. Otherwise, it outputs UNKNOWN to indicate that Robust Non-Termination may hold (i.e., FuncTion-TR raises a potential Robust Non-Termination alarm). In this case, the defined partitions of the inferred decision tree are a *sufficient precondition* for Termination Resilience.

Experimental Evaluation. Note that always terminating programs trivially satisfy Termination Resilience. For this reason, in our evaluation, we deliberately focus on programs that admit both terminating and non-terminating executions, thereby targeting the class of benchmarks where termination resilience is a non-trivial property and where our analysis can be meaningfully evaluated.

We selected a total of 62 programs from three sources: SV-COMP 2024[1], the benchmarks of the state-of-the-art non-termination analyzer Pulse [29], and the recent survey on non-termination bugs by Shi et al. [30]. From SV-COMP, we collected 39 programs from the sets *termination-crafted-lit*, *termination-restricted-15*, and *termination-nla* in the Termination category, all labeled as non-terminating in the associated `.yml` files (average length of 23 lines of code). From the Pulse benchmarks [29] we retained 10 programs (averaging 20 lines of code), and from the survey benchmarks [30] we retained 13 (averaging 25 lines of code – each program is a simplified extract derived from real-world open-source software that exhibited a non-termination bug). We excluded 16 (resp. 36) always terminating programs from these two latter sources. Furthermore, we discarded 33 (resp. 42) programs making use of pointers, bitwise operations, or recursive functions, which are currently not fully supported by our prototype tool.

We constructed variants of these 62 programs, for all possible combinations of variable initializations (e.g., `__VERIFIER_nondet_int()` for programs in SV-COMP) with untrusted (`input`) or trusted ($[-\infty, +\infty]$) values. In doing so, we obtained a benchmark of 278 test cases (150 from SV-COMP, 42 from Pulse [29], 86 from Shi et al. [30]). Note that variable initializations also occur within loops for 54 of 278 test cases (28 from SV-COMP, 12 from Pulse [29], 14 from Shi et al. [30]). The experiments were conducted on a 64-bit 8-Core CPU (AMD® Ryzen

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

Table 1: Evaluation results.

Benchmark	Configuration	Property	Verified	Alarms	TO	Time (s)
SV-COMP	FuncTion-TR-Boxes	Termination	0	150	0	3.0
		Termination Resilience	58	92	0	3.0
	FuncTion-TR-Polyhedra	Termination	0	150	0	6.0
		Termination Resilience	99	51	0	14.0
Pulse [29]	FuncTion-TR-Boxes	Termination	0	42	0	0.5
		Termination Resilience	23	19	0	0.5
	FuncTion-TR-Polyhedra	Termination	0	42	0	7.0
		Termination Resilience	23	19	0	17.0
Shi et al. [30]	FuncTion-TR-Boxes	Termination	0	86	0	2.0
		Termination Resilience	58	28	0	2.0
	FuncTion-TR-Polyhedra	Termination	0	86	0	54.0
		Termination Resilience	58	20	8	257.0

7 pro 5850u) with 16GB of RAM on Ubuntu 20.04. Note that, for the moment, our tool is single-threaded and uses only one CPU core.

Table 1 summarizes the results of the evaluation with FuncTion-TR configured to use the boxes (FuncTion-TR-Boxes) or polyhedra (FuncTion-TR-Polyhedra) abstract domain to manage decision tree constraints. We configured the analysis to perform widening after two iterations, and we left the extension with lexicographic functions disabled. We used a timeout of 120s (wall time) per test case. Activating the extension or increasing the widening delay makes no difference on these benchmarks aside from an increased execution time. As expected, proving Termination Resilience instead of Termination (i.e., Termination Resilience where all variables are considered untrusted) considerably reduces the number of alarms, almost 65% using decision tree with polyhedral constraints. This shows that our termination resilience analysis is an effective way to triage and prioritize alarms related to program non-termination. The execution time reported in the last column of Table 1 corresponds to the total analysis time over all test cases that did not time out.

The full comparison between FuncTion-TR-Boxes and FuncTion-TR-Polyhedra is shown in Figure 15, where the axes denote execution (wall) time in seconds, and green stars (⋆) label test cases where the configurations have the same precision (i.e., both prove Termination Resilience or raise an alarm), while red crosses (×) and blue circles (•) label test cases where FuncTion-TR-Polyhedra and FuncTion-TR-Boxes are more precise (i.e., prove Termination Resilience for more test cases), respectively. The presence of blue circles may be surprising: in few test cases, using a more precise numerical domain to manage decision

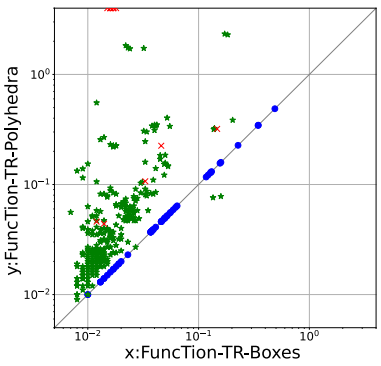


Fig. 15: FuncTion-TR-Boxes vs FuncTion-TR-Polyhedra.

tree constraints actually causes a loss of precision when widening is performed (because the widening heuristic leverages adjacent decision tree partitions, and more partitions are formed with more precise decision tree constraints), while using a less precise numerical domain allows the widening to generalize better. We leave the implementation of better widening heuristics [8] for future work.

Triage of Alarms Found by Non-Termination Analyzers. We evaluated the ability of FuncTion-TR to filter out non-critical non-termination alarms found by the non-termination analyzers Pulse, Dynamite and Ultimate-Automizer.

We reused the benchmark in which Raad et al. [29] reported non-termination bugs found by Pulse and Dynamite [21]. The programs in this benchmark use non-linear arithmetic and are extracted from the set *termination-nla* in the Termination category of SV-COMP. We excluded 9 out of 40 non-terminating programs that rely on unsigned int, since we assume a mathematical integer semantics. As in the previous evaluation, for each test filename.c, we generated variants for each possible combination of variable initialization which gave us 86 programs to test (with an average of 54 lines of code). In addition to the result of [29], we also ran the non-termination analysis of Ultimate-Automizer [7].

In Table 2, the suffixes of the test names, generated by the regular expression $-[\mathrm{r}+\mathrm{i}]^*$, describe the sequence of variable initializations. For instance the test `filename-ri.c` is the variant of `filename.c` in which we used a trusted (non-deterministic) value ($[-\infty, +\infty]$) for the first variable initialization and an untrusted (input) value for the second. A non-termination bug found by Pulse (column P in Table 2), Dynamite (column D in Table 2) or Ultimate-Automizer (column U in Table 2) is denoted ⚠, an empty cell indicates that the tool is inconclusive. Note that the tool results should not be compared as they consider different integer semantics. For FuncTion-TR, ✓ means that it proved Termination Resilience (returns TRUE), while ❗(FP) or ❗(TP) denotes the cases where FuncTion-TR raised a false alarm (upon manual inspection, Termination Resilience, in fact, holds) or a true alarm (upon manual inspection, Termination Resilience indeed does not hold), respectively.

From this evaluation, we conclude that our prototype FuncTion-TR is able to filter out non-critical non-termination alarms found by non-termination analyzers. More precisely, Termination Resilience is proved for 16 programs where Pulse or Dynamite or Ultimate-Automizer found a non-terminating execution, hence FuncTion-TR removes 16 alarms. Moreover, Termination Resilience is proved for 4 programs where Pulse, Dynamite and Ultimate-Automizer did not find any non-termination bug. Finally, by manual inspection, we report that 58 out of the 62 alarms raised by FuncTion-TR are true positives. This result was expected: most of the non-terminating programs from the SV-COMP category *termination-nla* are constructed from terminating ones by making them always diverging, which is a case of Robust Non-Termination where FuncTion-TR must raise an alarm. Dually, 4 of the 62 alarms are false positives. In our benchmark it is mostly due to the inherited imprecision of the decision tree abstract domain on programs exhibiting non-linear arithmetic operations or, more frequently, when the widening operator does not generalize well.

Table 2: Triage of non-termination alarms by FuncTion-TR.

Test Case	P	D	U	FuncTion-TR
bresenham1-both-nt-ii	⚠			!(TP)
bresenham1-both-nt-ir	⚠			!(TP)
bresenham1-both-nt-ri	⚠			!(TP)
bresenham1-both-nt-rr	⚠			!(FP)
cohencu1-both-nt-i	⚠	⚠		!(TP)
cohencu1-both-nt-r	⚠	⚠		!(TP)
cohencu2-both-nt-i				!(TP)
cohencu2-both-nt-r				!(TP)
cohencu3-both-nt-i				!(TP)
cohencu3-both-nt-r				!(TP)
cohencu4-both-nt-i				!(TP)
cohencu4-both-nt-r				!(TP)
cohencu5-both-nt-i		⚠		!(TP)
cohencu5-both-nt-r		⚠		!(TP)
dijkstra1-both-nt-i	⚠		⚠	!(TP)
dijkstra1-both-nt-r	⚠		⚠	✓
dijkstra2-both-nt-i	⚠		⚠	!(TP)
dijkstra2-both-nt-r	⚠		⚠	✓
dijkstra3-both-nt-i	⚠		⚠	!(TP)
dijkstra3-both-nt-r	⚠		⚠	✓
dijkstra4-both-nt-i	⚠		⚠	!(TP)
dijkstra4-both-nt-r	⚠		⚠	✓
dijkstra5-both-nt-i	⚠		⚠	!(TP)
dijkstra5-both-nt-r	⚠		⚠	✓
dijkstra4-both-nt-i	⚠		⚠	!(TP)
dijkstra6-both-nt-r	⚠		⚠	✓
egcd2-both-nt-ii	⚠	⚠		!(TP)
egcd2-both-nt-ir	⚠	⚠		✓
egcd2-both-nt-ri	⚠	⚠		✓
egcd2-both-nt-rr	⚠	⚠		✓
egcd3-both-nt-ii	⚠	⚠		!(TP)
egcd3-both-nt-ir	⚠	⚠		✓
egcd3-both-nt-ri	⚠	⚠		!(TP)
egcd3-both-nt-rr	⚠	⚠		✓
egcd-both-nt-ii	⚠	⚠		!(TP)
egcd-both-nt-ir	⚠	⚠		✓
egcd-both-nt-ri	⚠	⚠		✓
egcd-both-nt-rr	⚠	⚠		✓
freire1-both-nt-i	⚠			!(TP)
freire1-both-nt-r	⚠			!(TP)
geo1-both-nt-ii		⚠		!(TP)
geo1-both-nt-ir		⚠		!(TP)
geo1-both-nt-ri		⚠		!(TP)
geo1-both-nt-rr		⚠		!(TP)
geo2-both-nt-ii		⚠		!(TP)
geo2-both-nt-ir		⚠		!(TP)
geo2-both-nt-ri		⚠		!(TP)
geo2-both-nt-rr		⚠		!(TP)
geo3-both-nt-ii		⚠		!(TP)
geo3-both-nt-ir		⚠		!(TP)
geo3-both-nt-ri		⚠		!(TP)
geo3-both-nt-rr		⚠		!(TP)
hard2-both-nt-i		⚠		!(TP)
hard2-both-nt-r		⚠		!(TP)
hard-both-nt-ii	⚠	⚠		!(TP)
hard-both-nt-ir	⚠	⚠		✓
hard-both-nt-ri	⚠	⚠		!(TP)
hard-both-nt-rr	⚠	⚠		✓
prod4br-both-nt-ii				!(TP)
prod4br-both-nt-ir				✓
prod4br-both-nt-ri				!(TP)
prod4br-both-nt-rr				✓
prodbin-both-nt-ii				!(TP)
prodbin-both-nt-ir				✓
prodbin-both-nt-ri				!(TP)
prodbin-both-nt-rr				✓
ps2-both-nt-i		⚠		!(TP)
ps2-both-nt-r		⚠		!(TP)
ps3-both-nt-i				!(TP)
ps3-both-nt-r				!(TP)
ps4-both-nt-i		⚠		!(TP)
ps4-both-nt-r		⚠		!(TP)
ps5-both-nt-i				!(TP)
ps5-both-nt-r				!(TP)
ps6-both-nt-i				!(TP)
ps6-both-nt-r				!(TP)
sqrt1-both-nt-i		⚠		!(TP)
sqrt1-both-nt-r		⚠		!(TP)
sqrt2-both-nt-ii				!(TP)
sqrt2-both-nt-ir				!(FP)
sqrt2-both-nt-ri				!(FP)
sqrt2-both-nt-rr				!(FP)

Data Availability. The instrumented programs used for the experimental evaluation are provided (alongside with the tool) in our artifact on Zenodo [26].

7 Related Work

Several approaches have been proposed in the literature to prove program termination [3,2,4,6,14,23,24,19, etc.], or the existence of diverging executions (potential non-termination) in programs [29,37,17,5,21,22,20,19, etc.]. However, to the best of our knowledge, none of the other works studies termination or non-termination in the presence of untrusted inputs and trusted variables. We believe an interesting avenue for future research is to extend these approaches, with the aim of developing new methods to prove Termination Resilience or the existence of Robust Non-Termination bugs.

This work is inspired by the work of Girol et al. [15,16], where they introduce the notion of Robust Reachability of a *safety* bug. They additionally propose symbolic execution and bounded model checking techniques to find robustly reachable bugs. Instead, we introduce the notion of Robust Non-Termination, the

robust occurrence of a *liveness* bug, and its negation Termination Resilience. We propose a static analysis approach to verify Termination Resilience and detect potential Robust Non-Termination bugs.

In a similar spirit as Girol et al., Parolini and Miné have introduced the notion of *safety* Non-Exploitability [28,27] and proposed a static analysis, combining taint and reachable values information, to prove that the untrusted inputs (called an attacker in their work) cannot trigger or silence a safety bug in a program. In our work, we focus on *liveness* bugs (notably, non-termination bugs) and we are only concerned with the untrusted inputs triggering these bugs, not silencing them. We also observe that the analysis of Parolini and Miné assumes full knowledge of trusted variables values at the time untrusted inputs values are chosen. In contrast, our semantic and static analysis frameworks naturally accommodate models with incomplete knowledge, including blind models in which the values of the untrusted inputs are chosen before the values of the trusted variables are observed (as in the programs in Figure 3).

Our work builds on the decision tree abstract domain [33,34] and inherits its imprecision (cf. Section 6). It also means that we could benefit from the extensions of the domain. Urban et al. [36] proposed a static analysis of CTL properties using the decision trees abstract domain augmented with the new abstract operators. Later, Moussaoui-Remil et al. [25], proposed an analysis operating on decision trees to infer minimal sets of variables that need to be constrained in order to ensure a CTL property. We could leverage this work to infer minimal sets of untrusted variables that need to be strained to ensure termination resilience.

Termination Resilience can be formulated within the Alternating-Time Temporal Logic (ATL) [1] whose formulas are the same as those of CTL (in which termination is expressible), except that the universal (A) and existential (E) operators are generalized with a selective quantifier $\langle\!\langle . \rangle\!\rangle$ over paths that are possible outcomes of games. Given n players $p_1, \ldots, p_n$, a program satisfies $\langle\!\langle p_1, \ldots, p_n \rangle\!\rangle \phi$ if and only if there exists a strategy for the n players to force the program to satisfy ϕ. Termination Resilience is expressed as the existence of a strategy for the trusted variables that ensures program termination. Hence, an abstract interpretation of ATL properties seems to be a natural generalization of our work toward an analysis of resilience CTL properties.

8 Conclusion and Future Work

We have proposed a novel program property, called Termination Resilience, and an abstract interpretation-based static analysis to infer sufficient preconditions ensuring it. To this end, we have enhanced the decision tree abstract domain of piecewise-defined functions [33,34] with non-trivial transformer operators. In our evaluation, we have shown that our static analysis is able to reduce by almost 65% the alarms related to program non-termination.

For future work, an interesting direction is a static analysis for ATL properties with an initial focus on the resilience of CTL properties. We also plan to extend the analysis to data structure-manipulating programs.

References

1. Rajeev Alur, Thomas A Henzinger, and Orna Kupferman. Alternating-time temporal logic. *Journal of the ACM (JACM)*, 49(5):672–713, 2002.
2. Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. Correctness witnesses: exchanging verification results between verifiers. In Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 326–337. ACM, 2016.
3. Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. Witness validation and stepwise testification across software verifiers. In Elisabetta Di Nitto, Mark Harman, and Patrick Heymans, editors, *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 721–733. ACM, 2015.
4. Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety verification and refutation by k-invariants and k-induction. In Sandrine Blazy and Thomas P. Jensen, editors, *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015, Proceedings*, volume 9291 of *Lecture Notes in Computer Science*, pages 145–161. Springer, 2015.
5. Krishnendu Chatterjee, Ehsan Kafshdar Goharshady, Petr Novotný, and Dorde Zikelic. Proving non-termination by program reversal. In Stephen N. Freund and Eran Yahav, editors, *PLDI '21: 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, Virtual Event, Canada, June 20-25, 2021*, pages 1033–1048. ACM, 2021.
6. Hong-Yi Chen, Cristina David, Daniel Kroening, Peter Schrammel, and Björn Wachter. Bit-precise procedure-modular termination analysis. *ACM Trans. Program. Lang. Syst.*, 40(1):1:1–1:38, 2018.
7. Yu-Fang Chen, Matthias Heizmann, Ondřej Lengál, Yong Li, Ming-Hsien Tsai, Andrea Turrini, and Lijun Zhang. Advanced automata-based algorithms for program termination checking. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 135–150, 2018.
8. Nathanaël Courant and Caterina Urban. Precise Widening Operators for Proving Termination by Abstract Interpretation. In Axel Legay and Tiziana Margaria, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, volume 10205 of *Lecture Notes in Computer Science*, pages 136–152, 2017.
9. Patrick Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Theoretical Computer Science*, 277(1-2):47–103, 2002.
10. Patrick Cousot and Radhia Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.
11. Patrick Cousot and Radhia Cousot. Abstract Interpretation Frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

12. Patrick Cousot and Radhia Cousot. An Abstract Interpretation Framework for Termination. In John Field and Michael Hicks, editors, *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*, pages 245–258. ACM, 2012.
13. Robert W. Floyd. Assigning Meanings to Programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.
14. Mirco Giacobbe, Daniel Kroening, and Julian Parsert. Neural termination analysis. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 633–645. ACM, 2022.
15. Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not All Bugs Are Created Equal, But Robust Reachability Can Tell the Difference. In Alexandra Silva and K. Rustan M. Leino, editors, *Computer Aided Verification - 33rd International Conference, CAV 2021, Virtual Event, July 20-23, 2021, Proceedings, Part I*, volume 12759 of *Lecture Notes in Computer Science*, pages 669–693. Springer, 2021.
16. Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Introducing robust reachability. *Formal Methods Syst. Des.*, 63(1):206–234, 2024.
17. Jera Hensel, Constantin Mensendiek, and Jürgen Giesl. Aprove: Non-termination witnesses for C programs - (competition contribution). In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part II*, volume 13244 of *Lecture Notes in Computer Science*, pages 403–407. Springer, 2022.
18. Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.
19. Naoki Kobayashi, Kento Tanahashi, Ryosuke Sato, and Takeshi Tsukada. HFL(Z) validity checking for automated program verification. *Proc. ACM Program. Lang.*, 7(POPL):154–184, 2023.
20. Daniel Larraz, Kaustubh Nimkar, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving non-termination using max-smt. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 779–796. Springer, 2014.
21. Ton Chanh Le, Timos Antonopoulos, Parisa Fathololumi, Eric Koskinen, and ThanhVu Nguyen. Dynamite: dynamic termination and non-termination proofs. *Proc. ACM Program. Lang.*, 4(OOPSLA):189:1–189:30, 2020.
22. Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 489–498. ACM, 2015.

23. Viktor Malík, Frantisek Necas, Peter Schrammel, and Tomás Vojnar. 2ls: Arrays and loop unwinding - (competition contribution). In Sriram Sankaranarayanan and Natasha Sharygina, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 29th International Conference, TACAS 2023, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Paris, France, April 22-27, 2023, Proceedings, Part II*, volume 13994 of *Lecture Notes in Computer Science*, pages 529–534. Springer, 2023.
24. Ravindra Metta, Hrishikesh Karmarkar, Kumar Madhukar, R. Venkatesh, and Supratik Chakraborty. PROTON: probes for termination or not (competition contribution). In Bernd Finkbeiner and Laura Kovács, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 30th International Conference, TACAS 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part III*, volume 14572 of *Lecture Notes in Computer Science*, pages 393–398. Springer, 2024.
25. Naïm Moussaoui Remil, Caterina Urban, and Antoine Miné. Automatic detection of vulnerable variables for CTL properties of programs. In Nikolaj S. Bjørner, Marijn Heule, and Andrei Voronkov, editors, *LPAR 2024: Proceedings of 25th Conference on Logic for Programming, Artificial Intelligence and Reasoning, Port Louis, Mauritius, May 26-31, 2024*, volume 100 of *EPiC Series in Computing*, pages 116–126. EasyChair, 2024.
26. Naïm Moussaoui Remil and Caterina Urban. Termination resilience static analysis (artifact). `https://zenodo.org/records/17176433`, 2025.
27. Francesco Parolini. *Static Analysis for Security Properties of Software by Abstract Interpretation. (Analyse statique des propriétés de sécurité des logiciels par interprétation abstraite).* PhD thesis, Sorbonne University, Paris, France, 2024.
28. Francesco Parolini and Antoine Miné. Sound Abstract Nonexploitability Analysis. In Rayna Dimitrova, Ori Lahav, and Sebastian Wolff, editors, *Verification, Model Checking, and Abstract Interpretation - 25th International Conference, VMCAI 2024, London, United Kingdom, January 15-16, 2024, Proceedings, Part II*, volume 14500 of *Lecture Notes in Computer Science*, pages 314–337. Springer, 2024.
29. Azalea Raad, Julien Vanegue, and Peter W. O'Hearn. Non-termination proving at scale. *Proc. ACM Program. Lang.*, 8(OOPSLA2):246–274, 2024.
30. Xiuhan Shi, Xiaofei Xie, Yi Li, Yao Zhang, Sen Chen, and Xiaohong Li. Large-scale analysis of non-termination bugs in real-world OSS projects. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 256–268. ACM, 2022.
31. Alan Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.
32. Caterina Urban. FuncTion: An Abstract Domain Functor for Termination - (Competition Contribution). In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 464–466. Springer, 2015.
33. Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs (Analyse Statique par Interprétation Abstraite de Pro-*

priétés Temporelles Fonctionnelles des Programmes). PhD thesis, École Normale Supérieure, Paris, France, 2015.

34. Caterina Urban and Antoine Miné. A Decision Tree Abstract Domain for Proving Conditional Termination. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis - 21st International Symposium, SAS 2014, Munich, Germany, September 11-13, 2014. Proceedings*, volume 8723 of *Lecture Notes in Computer Science*, pages 302–318. Springer, 2014.
35. Caterina Urban and Antoine Miné. An Abstract Domain to Infer Ordinal-Valued Ranking Functions. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, volume 8410 of *Lecture Notes in Computer Science*, pages 412–431. Springer, 2014.
36. Caterina Urban, Samuel Ueltschi, and Peter Müller. Abstract interpretation of CTL properties. In Andreas Podelski, editor, *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings*, volume 11002 of *Lecture Notes in Computer Science*, pages 402–422. Springer, 2018.
37. Yao Zhang, Xiaofei Xie, Yi Li, Sen Chen, Cen Zhang, and Xiaohong Li. Endwatch: A practical method for detecting non-termination in real-world software. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 686–697. IEEE, 2023.