

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 executions. In semantic terms, we treat untrusted variables demonically, since

```

1x := input;
2z := 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 .

termination must be ensured under all value possibilities, while trusted variables 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 7). Both `while` loops in the program are non-terminating, the first when $x \leq 0$, and the second when $(z \geq x \text{ and } c \geq 0)$. 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 [33], 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 [31], 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, $\tau \triangleq \tau^i \uplus \tau^r$ 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 $\Omega_\tau \triangleq \{s \in \Sigma \mid \forall s' \in \Sigma: \langle s, s' \rangle \notin \tau\}$.

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

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, $S^{+\infty} \triangleq S^+ \cup S^\omega$ be the set of all non-empty finite or infinite sequences of elements in S , and $S^{*\infty} \triangleq S^{+\infty} \cup \{\varepsilon\}$. We write $T^+ \triangleq T \cap S^+$, $T^\omega \triangleq T \cap S^\omega$ for the selection of the non-empty finite sequences and the infinite sequences of $T \in \mathcal{P}(S^{+\infty})$, and $T; T' = \{ts't' \in S^{+\infty} \mid s \in S \wedge ts \in T \wedge st' \in T'\}$ 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, i.e., $\langle s, s' \rangle \in \tau$. We write σ_n to denote its 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 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 $\{\sigma^r \mid \sigma \in T\}$). 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 $(\mathcal{P}(\Sigma^{+\infty}), \sqsubseteq, \sqcup, \sqcap, \Sigma^\omega, \Sigma^+)$ [9]:

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

where the computational order is $T_1 \sqsubseteq T_2 = T_1^+ \subseteq T_2^+ \wedge T_1^\omega \supseteq T_2^\omega$. In the following, we write $\Lambda[P]$ to denote the trace semantics of a given program P .



(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 from Example 1 and 2. Input transitions are represented with dashed red lines, while solid black lines represent regular transitions.

Given an initial set of states $I \subseteq \Sigma$, we define $\Lambda[P]I \triangleq \{\sigma \in \Lambda[P] \mid \sigma_0 \in I\}$ 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 $\Lambda_1 = \{\text{ab}, \text{ac}^\omega\}$. \triangleleft

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 $\Lambda_2 = \{\text{abd}, \text{abe}^\omega, \text{ac}\}$. \triangleleft

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:*

$$\mathcal{RNT} \triangleq \{T \in \mathcal{P}(\Sigma^{+\infty}) \mid \exists i \in T^i \forall \sigma \in T: \sigma^i = i \Rightarrow \sigma \in T^\omega\} \quad (2)$$

Note that any set $T \in \mathcal{P}(\Sigma^\omega)$ only containing infinite traces always belongs to \mathcal{RNT} (in such case $\sigma^i = \varepsilon$ for all σ in T), 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 $\text{ac}^\omega \in \Lambda_1$. Thus $\Lambda_1 \in \mathcal{RNT}$. \triangleleft

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:*

$$\mathcal{TR} \triangleq \neg \mathcal{RNT} = \{T \in \mathcal{P}(\Sigma^{+\infty}) \mid \forall i \in T^i \exists \sigma \in T^+ : \sigma^i = i\} \quad (3)$$

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, $\Lambda_2 \in \mathcal{TR}$. \triangleleft

The trace semantics $\Lambda[P]$ of a program P , fully characterizing the behavior of P , is sound and complete for verifying Termination Resilience (and, equivalently, disproving Robust Non-Termination), for an initial set of states $I \subseteq \Sigma$:

$$P \models_I \mathcal{TR} \Leftrightarrow \Lambda[P]I \in \mathcal{TR} \Leftrightarrow \Lambda[P]I \notin \mathcal{RNT} \Leftrightarrow P \not\models_I \mathcal{RNT}. \quad (4)$$

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, in Section 5, we further abstract this semantics into a sound but computable semantics Λ_{tr}^b that yields a static analysis for Termination Resilience. Due to space constraints, proofs are relegated to Appendix D.

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 [30,13] to program states belonging to terminating traces. Specifically, we abstract the trace semantics $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ into the *termination resilience semantics* $\Lambda_{\text{tr}}: \Sigma \rightarrow \mathbb{O}$ (where \mathbb{O} is the set of ordinals), which is the best (potential) ranking function [12,32] for the program states in the traces in Λ .

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

$$\alpha_{\text{tr}}(T) \triangleq \alpha_v \circ \vec{\alpha}(T) \quad (5)$$

where $\vec{\alpha}: \mathcal{P}(\Sigma^{+\infty}) \rightarrow \mathcal{P}(\Sigma \times \Sigma)$ extracts the smallest transition relation $\tau \subseteq \Sigma \times \Sigma$ that generates a given set of traces T : $\vec{\alpha}(T) \triangleq \{\langle s, s' \rangle \mid \exists \sigma, \sigma': \sigma s s' \sigma' \in T\}$, and $\alpha_v: \mathcal{P}(\Sigma \times \Sigma) \rightarrow (\Sigma \rightarrow \mathbb{O})$ ranks the elements in the domain of r :

$$\alpha_v(\emptyset) \triangleq \dot{\emptyset}$$

$$\alpha_v(r)s \triangleq \begin{cases} 0 & s \in \Omega_r \\ \sup \left\{ \alpha_v(r)s' + 1 \mid \langle s, s' \rangle \in r^i \right\} & s \in \widetilde{\text{pre}}_{r,i}(\text{dom}(\alpha_v(r))) \\ \inf \left\{ \alpha_v(r)s' + 1 \mid \langle s, s' \rangle \in r^r \right\} & s \in \text{pre}_{r,r}(\text{dom}(\alpha_v(r))) \\ \text{undefined} & \text{otherwise} \end{cases} \quad (6)$$

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

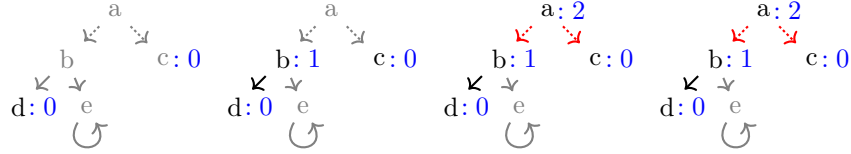
function has value 0 (and is undefined elsewhere). Then, the domain of the function grows by retracing the program (transitions) backwards and counting the number of performed program steps as value of the function. We model the non-determinism arising from input transitions *demonically*, requiring all successor states to already be in the domain of the function ($s \in \widetilde{\text{pre}}_{r,i}(\text{dom}(\alpha_v(r)))$), and count the *maximum* number of performed program steps. Instead, non-determinism 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 $\Lambda_{\text{tr}} \in (\Sigma \rightarrow \mathbb{O})$ is $\Lambda_{\text{tr}} \triangleq \alpha_{\text{tr}}(\Lambda)$, where $\Lambda \in \mathcal{P}(\Sigma^{+\infty})$ is the trace semantics (cf. Equation 2).

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

Example 5. Let $\langle \Sigma_2, \tau_2 \rangle$ and Λ_{t_2} be the transition system from Example 2 and its termination resilience trace semantics, respectively. The fixpoint iterates of its termination resilience semantics Λ_{tr_2} are the following:



Note that Λ_{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). \triangleleft

The termination resilience semantics $\Lambda_{\text{tr}}[P]$ 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). $\Lambda[P]I \in \mathcal{TR} \Leftrightarrow I \subseteq \text{dom}(\Lambda_{\text{tr}}[P])$

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:

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

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

<pre> ¹x := input; ²z := 10; while ³(z >= 0) { ⁴z := z - x; ⁵} </pre>	<pre> ¹x = input; ²z := 10; ³c := [-2, 1]; while ⁴(z >= x) { ⁵z := z + c; ⁶} </pre>
(a) Program P_1	(b) Program P_2

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

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 ($^l x := \text{input}$), a (trusted) non-deterministic choice ($^l x := [c_1, c_2]$) or an arithmetic expression ($^l x := a$). A program is an instruction statement $s \in \text{Stmt}$ 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 $p \in \text{Prog}$ written in our small language. Then, in the next section, we will define its abstraction $\Lambda_{\text{tr}}^{\sharp}$.

Taint Semantics. An environment $\rho \in \mathcal{E} \triangleq \mathbb{X} \rightarrow \mathbb{Z}$ maps variables to their value. The semantics $\llbracket e \rrbracket : \mathcal{E} \rightarrow \mathcal{P}(\mathbb{Z})$ of an expression $e \in \text{AExp} \cup \{\text{input}, [c_1, c_2]\}$ yields the possible values of e in a given environment (cf. Appendix A).

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 by l . To keep track of the effect of input transitions on program states, we need a *taint semantics* $\mathfrak{T}[\mathbf{s}] : \mathcal{P}(\mathbb{X}) \rightarrow \mathcal{P}(\mathbb{X})$ that collects the set of variables that depend on untrusted inputs, i.e., are *tainted*, after the execution of a program statement (cf. Appendix B). By pointwise-lifting of \mathfrak{T} with respect to the program labels \mathcal{L} , we obtain the lifted taint semantics $\dot{\mathfrak{T}} : \mathcal{L} \rightarrow \mathcal{P}(\mathbb{X})$ mapping each label to the set of variables that are tainted at that label.

Denotational Termination Resilience Semantics. We can now define in Figure 4 the termination resilience semantics $\Lambda_{\text{tr}}[\mathbf{s}] : (\mathcal{E} \rightarrow \mathbb{O}) \rightarrow (\mathcal{E} \rightarrow \mathbb{O})$ for each program statement. Each transfer function $\Lambda_{\text{tr}}[\mathbf{s}]$ takes as input a (potential) ranking function that is valid *after* the execution of the statement \mathbf{s} and returns a (potential) ranking function that is valid *before* the execution of \mathbf{s} .

$$\begin{aligned}
\Lambda_{\text{tr}}[\text{skip}]^l f &\triangleq \lambda \rho \in \text{dom}(f). f(\rho) + 1 \\
\Lambda_{\text{tr}}[x := e]^l f &\triangleq \lambda \rho. \begin{cases} \sup\{f(\rho[x \leftarrow v]) + 1 \mid v \in \llbracket e \rrbracket \rho\} & e = \text{input} \vee A \\ \inf \left\{ f(\rho[x \leftarrow v]) + 1 \mid \begin{array}{l} v \in \llbracket e \rrbracket \rho \wedge \\ \rho[x \leftarrow v] \in \text{dom}(f) \end{array} \right\} & e = [c_1, c_2] \vee B \\ \text{undefined} & \text{otherwise} \end{cases} \\
A &\Leftrightarrow (e = a \wedge \mathfrak{T}[\llbracket a \rrbracket](\mathfrak{S}(l)) \wedge \exists v \in \llbracket a \rrbracket \rho \wedge \forall v \in \llbracket a \rrbracket \rho, \rho[x \leftarrow v] \in \text{dom}(f)) \\
B &\Leftrightarrow (e = a \wedge \neg \mathfrak{T}[\llbracket a \rrbracket](\mathfrak{S}(l)) \wedge \exists v \in \llbracket a \rrbracket \rho : \rho[x \leftarrow v] \in \text{dom}(f)) \\
\Lambda_{\text{tr}}[\text{if }^l a \bowtie 0 \{s\}] f &\triangleq \Theta_{\text{tr}}(f) \\
\Theta_{\text{tr}} &\triangleq \lambda X. \lambda \rho. \begin{cases} \Lambda_{\text{tr}}[\llbracket s \rrbracket] X(\rho) + 1 & \forall v \in \llbracket a \rrbracket \rho : v \bowtie 0 \wedge \rho \in \text{dom}(\Lambda_{\text{tr}}[\llbracket s \rrbracket] f(\rho)) \\ f(\rho) + 1 & \forall v \in \llbracket a \rrbracket \rho : v \not\bowtie 0 \wedge \rho \in \text{dom}(f(\rho)) \\ \text{undefined} & \text{otherwise} \end{cases} \\
\Lambda_{\text{tr}}[\text{while }^l a \bowtie 0 \{s\}] f &\triangleq \text{lfp}_{\emptyset} \Theta_{\text{tr}} \\
\Lambda_{\text{tr}}[s_1; s_2] f &\triangleq \Lambda_{\text{tr}}[\llbracket s_1 \rrbracket](\Lambda_{\text{tr}}[\llbracket s_2 \rrbracket] f)
\end{aligned}$$

Fig. 4: Termination resilience semantics $\Lambda_{\text{tr}}[\llbracket s \rrbracket] : (\mathcal{E} \multimap \mathbb{O}) \rightarrow (\mathcal{E} \multimap \mathbb{O})$. Since $\llbracket a \rrbracket \rho$ is always a singleton $\{v\}$, we abuse notation and write $\llbracket a \rrbracket \rho$ to directly denote v .

If the right-hand side of an assignment is an untrusted input ($^l x := \text{input}$) or an expression a ($^l x := a$) that depends on tainted variables ($\mathfrak{T}[\llbracket a \rrbracket](\mathfrak{S}(l))$), the termination resilience $\Lambda_{\text{tr}}[x := a]^l$ returns a (potential) ranking function only defined over the environments ρ that, when subject to the assignment ($\rho[x \leftarrow v]$ with $v \in \llbracket a \rrbracket \rho$), *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 (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 ($\neg \mathfrak{T}[\llbracket a \rrbracket](\mathfrak{S}(l))$) – the resulting function is defined over the environments ρ for which *at least one* value $v \in \llbracket a \rrbracket \rho$ yields an environment $\rho[x \leftarrow v]$ in $\text{dom}(f)$ (case B). Its value is the greatest lower bound of the value of f on these environments plus one (minimum number of steps to termination). We refer to Example 9 and 11 in Appendix C for an example.

The termination resilience semantics $\Lambda_{\text{tr}}[\text{if }^l a \bowtie 0 \{s\}]$ for **if** statements, via Θ_{tr} , defines a potential ranking function over all environments in $\text{dom}(\Lambda_{\text{tr}}[\llbracket s \rrbracket] f)$ or $\text{dom}(f)$, i.e., all environments potentially leading to a terminating execution, depending on whether they satisfy or not satisfy the guard $a \bowtie 0$. The termination resilience $\Lambda_{\text{tr}}[\text{while }^l a \bowtie 0 \{s\}]$ for **while** statements is the least fixpoint of Θ_{tr} with respect to the computational order $f_1 \sqsubseteq f_2 \Leftrightarrow \text{dom}(f_1) \subseteq \text{dom}(f_2) \wedge \forall x \in \text{dom}(f_1) : f_1(x) < f_2(x)$. We refer to Example 10 in Appendix C for an example.

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

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

$$\Lambda_{\text{tr}}[\mathbf{p}] = \Lambda_{\text{tr}}[\mathbf{s}^l] \triangleq \Lambda_{\text{tr}}[\mathbf{s}](\lambda\rho.0). \quad (7)$$

5 Abstract Termination Resilience Semantics

In this section, we propose a sound abstraction $\Lambda_{\text{tr}}^{\sharp}[\mathbf{p}]$ of the denotational termination resilience semantics $\Lambda_{\text{tr}}[\mathbf{p}]$ defined in Section 4 by leveraging (and extending) Urban and Miné’s decision tree numerical abstract domain \mathcal{T} [33] to abstract potential ranking functions by means of piecewise-defined partial functions. Specifically, we consider the concretization-based abstraction [11] $\langle \mathcal{E} \rightarrow \mathbb{O}, \preceq \rangle \xleftarrow{\gamma} \langle \mathcal{T}, \preceq_T \rangle$ where the approximation order \preceq is defined as follows:

$$f_1 \preceq f_2 \Leftrightarrow \text{dom}(f_1) \supseteq \text{dom}(f_2) \wedge \forall x \in \text{dom}(f_2): f_1(x) \leq f_2(x). \quad (8)$$

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

Remark 1. Note that Urban and Miné’s abstract termination semantics [33] — let us denote it $\Lambda_{\text{t}}^{\sharp}[\mathbf{p}]$ — is already a sound but coarse over-approximation of $\Lambda_{\text{tr}}^{\sharp}[\mathbf{p}]$: $\Lambda_{\text{tr}}[\mathbf{p}] \preceq \gamma(\Lambda_{\text{t}}^{\sharp}[\mathbf{p}])$. Our objective in this section is to build upon and relax $\Lambda_{\text{t}}^{\sharp}[\mathbf{p}]$ 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 [32] 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 $\text{NODE}\{c\}: t_1, t_2$, with $t_1, t_2 \in \mathcal{T}$ — is labeled by a constraint $c \in \mathcal{C}$ over the program variables, and each leaf — denoted $\text{LEAF}: f$ — is labeled by a function $f \in \mathcal{F}$ of the program variables. 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.

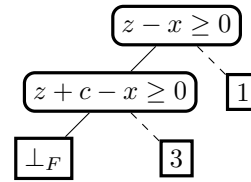


Fig. 5: Decision tree inferred by our analysis at label 4 of program P_2 (cf. Figure 3b).

Algorithm 1 Decision Tree Assignment

```

1: function  $\text{ASSIGN}_T \llbracket^l x := e \rrbracket(t)$ 
2:   if  $\text{isLeaf}(t)$  then
3:     return  $\text{LEAF} : \text{ASSIGN}_F \llbracket^l x := e \rrbracket(t.f)$ 
4:   else if  $\text{isNode}(t)$  then
5:      $I \leftarrow \text{ASSIGN}_C \llbracket^l x := e \rrbracket(t.c)$ 
6:      $J \leftarrow \text{ASSIGN}_C \llbracket^l x := e \rrbracket(\neg t.c)$ 
7:     if  $\text{isEmpty}(I) \wedge \text{isEmpty}(J)$  then
8:       return  $\text{ASSIGN}_T \llbracket^l x := e \rrbracket(t.l) \vee_T \text{ASSIGN}_T \llbracket^l x := e \rrbracket(t.r)$ 
9:     else if  $\text{isEmpty}(I) \wedge \perp_C \in J$  then
10:      return  $\text{ASSIGN}_T \llbracket^l x := e \rrbracket(t.l)$ 
11:     else if  $\perp_C \in I \wedge \text{isEmpty}(J)$  then
12:      return  $\text{ASSIGN}_T \llbracket^l x := e \rrbracket(t.r)$ 
13:     else
14:        $t_1 \leftarrow \text{PRUNET}(\text{ASSIGN}_T \llbracket^l x := e \rrbracket(t.l), I)$ 
15:        $t_2 \leftarrow \text{PRUNET}(\text{ASSIGN}_T \llbracket^l x := e \rrbracket(t.r), J)$ 
16:     return  $t_1 \vee_T t_2$ 

```

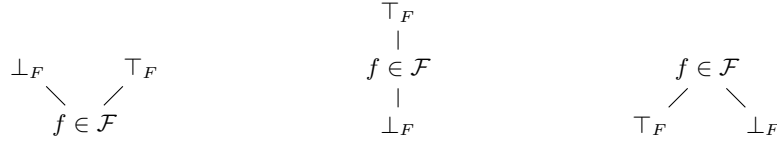
(a) Approximation order \preceq_F (b) Computational order \sqsubseteq_F (c) Resilience order \leq_F

Fig. 6: Hasse diagrams for difference orders of decision tree leaves.

Figure 5 shows an example of decision tree. The leaf with value \perp_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 5 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 $c < 0$ (cf. Example 9 in Appendix C): the decision tree is thus a sound approximation with respect to \preceq (cf. Equation 8).

The partitioning induced by decision tree constraints is dynamic: the analysis begins at the end of the program with the decision tree $\text{LEAF} : 0$ (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 $\text{ASSIGN}_T \llbracket^l x := e \rrbracket$ originally defined in [32]. 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 handled by the auxiliary operator $\text{ASSIGN}_F \llbracket^l x := e \rrbracket$ 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 decision tree in Figure 5 and the assignment $c := [-2, 1]$ at label 3 in Figure 3b. The result of the assignment on $\text{LEAF} : 3$ is $\text{LEAF} : 4$, simply increasing the value of its function by one, while $\text{LEAF} : \perp_F$ remains unchanged. Node constraints (accessed with $t.c$) and their negation ($\neg t.c$) are handled by the auxiliary operator $\text{ASSIGN}_C \llbracket^l x := e \rrbracket$ of the underlying numerical abstract domain \mathcal{C} used to manage constraints, 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 constraints resulting from (over-approximating) the assignment to $t.c$ and $\neg t.c$, respectively. For example, considering again the decision tree in Figure 5, 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 $\text{LEAF} : \perp_F$ with the set of constraints I results in the decision tree shown in Figure 7a), while the pruning of $\text{LEAF} : 4$ with J results in the tree in Figure 7b).



Fig. 7: 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 \perp_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 γ_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 (\perp_F or \top_F) in either of the joined decision trees, cf. the Hasse diagram in Figure 6a. Continuing the example, joining the trees in Figure 7 yields the tree in Figure 8.

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 γ_T . In case I is empty and J is an unsatisfiable set of constraints (i.e., the unsatisfiable constraint \perp_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

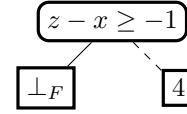


Fig. 8: Approximation join of Figure 7a and Figure 7b.

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 ∇_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 \perp_F (i.e., the analysis grows the domain of the inferred raking 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 6b.

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 \forall_T between decision trees that retains the leaves labeled with a *defined* function in either of the joined decision trees, cf. Figure 6c.

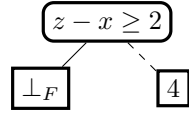


Fig. 9: Resilience join of Figure 7a and Figure 7b.

We implement $\text{ASSIGN}_T \llbracket^l x := [c_1, c_2] \rrbracket$ for non-deterministic assignments following Algorithm 1, but using the resilience join \forall_T instead of the approximation join ∇_T at Lines 8 and 16. Let us consider again the left subtree in Figure 5 and the non-deterministic variable assignment $c := [-2, 1]$ at label 3 in Figure 3b. Joining the trees in Figure 7 resulting from the PRUNE_T at Lines 14 and 15 yields the decision

tree in Figure 9, 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 10.

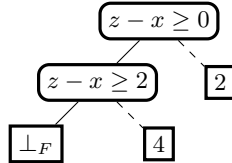


Fig. 10: Decision tree inferred by our analysis at label 3 of program P_2 .

Note that this seemingly anodyne modification is actually insidious. Let us consider the decision tree in Figure 11 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 \vee_F to

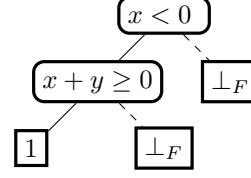


Fig. 11: Unfilled decision tree.

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 11 requires $y > 0$ but this constraint is missing from the decision tree because 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 redundant constraints. 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 \mathcal{C} to find these redundant constraints and adds them to the decision tree t .

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 4):

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

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 assignment with a non-tainted right-hand side expression (i.e., an expression that does not depend from 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 12 shows the most precise decision trees that can be inferred at label 4 and 6.

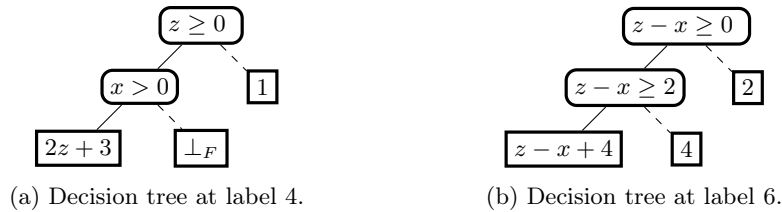


Fig. 12: Abstract termination resilience semantics for program P .

At program label 3, the boolean condition $a * a \geq 0$ cannot be precisely rep-

$$\begin{aligned}
\Lambda_{tr}^{\natural}[\![^l \text{skip}]\!]t &\triangleq \text{SKIP}_T(t) \\
\Lambda_{tr}^{\natural}[\![^l x := e]\!]t &\triangleq \text{ASSIGN}_T[\![^l x := e]\!](t) \\
\Lambda_{tr}^{\natural}[\![\text{if } ^l a \bowtie 0 \{s\}]\!]t &\triangleq \Theta_{tr}^{\natural}(t) \\
\Theta_{tr}^{\natural} &\triangleq \lambda X. \text{FILTER}_T[\![a \bowtie 0]\!](\Lambda_{tr}^{\natural}[s]X) \curlyvee_T \text{FILTER}_T[\![a \not\bowtie 0]\!](t) \\
\Lambda_{tr}^{\natural}[\![\text{while } ^l a \bowtie 0 \{s\}]\!]t &\triangleq \text{LFP}^{\natural} \Theta_{tr}^{\natural} \\
\Lambda_{tr}^{\natural}[s_1; s_2]t &\triangleq \Lambda_{tr}^{\natural}[s_2](\Lambda_{tr}^{\natural}[s_1]t)
\end{aligned}$$

Fig. 13: Abstract Program Semantics for Termination Resilience

resented by a (set of) linear constraints. Thus, no further constraints are added before joining the trees. Since the decision tree at label 6 is totally defined (cf. Figure 12b), using the resilience join would also yield a decision tree that is totally defined (cf. Figure 6c), implying that termination resilience always holds. 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 [32]. We leave finding less conservative approximations for future work.

5.3 Abstract Termination Resilience Semantics.

We can finally define in Figure 13 the *abstract termination resilience semantics* $\Lambda_{tr}^{\natural}[s] : \mathcal{T} \rightarrow \mathcal{T}$ for each program statement s . The SKIP_T operator simply increases by one the value of the functions labeling the leaves of the decision tree. The assignment operator $\text{ASSIGN}_T[\![^l x := e]\!]$ is implemented by Algorithm 1, modified as discussed in Section 5.2 for non-deterministic variable assignments. The semantics of conditional **if** statements Θ_{tr}^{\natural} employs the approximation join \curlyvee_T as discussed above, after handling the boolean conditions with the FILTER_T operator. The semantics for **while** loops iterates Θ_{tr}^{\natural} with widening starting from the totally undefined decision tree $\text{LEAF} : \perp_F$ until an abstract fixpoint is reached.

The *abstract termination resilience semantics* $\Lambda_{tr}^{\natural}[p] \in \mathcal{T}$ for a $p \in \text{Prog}$ 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 $p \in \text{Prog}$ be a program. Its abstract termination resilience semantics is:*

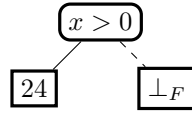
$$\Lambda_{tr}^{\natural}[p] = \Lambda_{tr}^{\natural}[s'] \triangleq \Lambda_{tr}^{\natural}[s](\text{LEAF} : 0). \quad (9)$$

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

Theorem 2 (Soundness). $\Lambda_{\text{tr}}\llbracket P \rrbracket \preceq \gamma(\Lambda_{\text{tr}}^{\sharp}\llbracket P \rrbracket)$

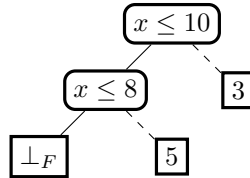
Corollary 1 (Soundness). $\Lambda\llbracket P \rrbracket I \in \mathcal{TR} \Leftarrow I \subseteq \text{dom}(\gamma(\Lambda_{\text{tr}}^{\sharp}\llbracket P \rrbracket))$

Example 6. Let us consider again program P_1 in Figure 3a and its most precise termination resilience semantics at program label 3 shown in Figure 12a. The assignment at label 2 – substituting z with 10 and increasing by one the value of the functions labeling the defined leaves – yields



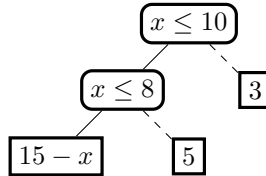
Finally, the assignment at label 1 yields $\text{LEAF} : \perp_F$. Its concretization $\gamma(\text{LEAF} : \perp_F)$ is the totally undefined function \emptyset . We are thus unable to prove termination resilience. In this case, this is a true positive since $\Lambda_{\text{tr}}\llbracket P_1 \rrbracket \notin \mathcal{TR}$ (cf. Example 11). \triangleleft

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 10. The assignment at label 2 yields



and the assignment at label 1 yields $\text{LEAF} : \perp_F$. We are again unable to prove termination resilience since $\gamma(\text{LEAF} : \perp_F)$ is the totally undefined function \emptyset . This, however, is a false alarm since $\Lambda_{\text{tr}}\llbracket P_2 \rrbracket \in \mathcal{TR}$ (cf. Example 11).

Let us consider instead the most precise abstract termination resilience semantics shown in Figure 12b. The assignment at label 2 yields



and the assignment at label 1 then yields $\text{LEAF} : \omega$. In this case, we prove termination resilience since $\gamma(\text{LEAF} : \omega)$ yields $\lambda\rho.\omega$, which is exactly $\Lambda_{\text{tr}}\llbracket P_2 \rrbracket$. \triangleleft

6 Implementation and Experimental Evaluation

Implementation. We implemented our termination resilience analysis in the open-source tool `FUNCTION-TR`¹. `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 [34], and tune the precision of the analysis by adjusting the widening delay. We use a simple implementation of the `FILLT` 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. 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`², the benchmarks of the state-of-the-art non-termination analyzer `Pulse` [28], and the recent survey on non-termination bugs by Shi et al. [29]. From `SV-COMP`, we collected 39 programs from the categories *crafted-lit*, *restricted-15*, and *nla*, all labeled as non-terminating in the associated `.yaml` files (average length of 23 lines of code). From the `Pulse` benchmarks [28] we retained 10 programs (averaging 20 lines of code), and from the survey benchmarks [29] 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 (`[-∞, +∞]`) values. In doing so, we obtained a benchmark of 268 test cases. Note that variable initializations also occur within loops for 58 of 268 test cases (32 from `SV-COMP`, 12 from `Pulse` [28], 14 from Shi et al. [29]). The experiments were conducted on a 64-bit 8-Core CPU (AMD® Ryzen 7 pro 5850u) with 16GB of RAM on Ubuntu 20.04.

¹ <https://zenodo.org/records/17123230>

² <https://sv-comp.sosy-lab.org/2024/>

Benchmark	Configuration	Property	Verified	Alarms	TO	Time
SV-COMP	FUNCTION-TR-Boxes	Termination	0	150	0	3 s
		Termination Resilience	58	92	0	3 s
	FUNCTION-TR-Polyhedra	Termination	0	150	0	6 s
		Termination Resilience	99	51	0	14 s
Pulse [28]	FUNCTION-TR-Boxes	Termination	0	42	0	0.5 s
		Termination Resilience	23	19	0	0.5 s
	FUNCTION-TR-Polyhedra	Termination	0	42	0	7 s
		Termination Resilience	23	19	0	17 s
Shi et al. [29]	FUNCTION-TR-Boxes	Termination	0	86	0	2 s
		Termination Resilience	58	28	0	2 s
	FUNCTION-TR-Polyhedra	Termination	0	86	0	54 s
		Termination Resilience	58	20	8	257 s

Fig. 14: Evaluation Results.

Figure 14 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. 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, up to 70% using decision tree with polyhedral constraints. This shows that that our termination resilience analysis is an effective way to triage and prioritize alarms related to program non-termination.

The full comparison between FUNCTION-TR-Boxes and FUNCTION-TR-Polyhedra is shown in Figure 15, where green stars (\star) label test cases where the configurations have the same precision (i.e., both prove Termination Resilience or raise an alarm), while red crosses (\times) and blue circles (\bullet) 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 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.

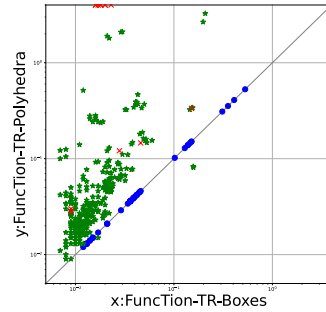


Fig. 15: FUNCTION-TR-Boxes vs FUNCTION-TR-Polyhedra.

Triage of bugs found by non-termination analyzers. We evaluated the ability of FUNCTION-TR to filter out non-critical non-termination bugs found by the following non-termination analyzers: Pulse, Dynamite and Ultimate-Automizer.

We reused the benchmark in which Raad et al. [28] reported non-termination bugs found by Pulse and Dynamite [21]. The programs in this benchmark use non-linear arithmetic and are extracted from the *termination-nla* 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 the associated variant 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 [28], we also ran the non-termination analysis of Ultimate-Automizer [7].

In Figure 16, the suffixes of the test names, generated by the regular expression $-[r + i]^*$, describe the sequence of variable initializations. For instance the test filename-ri.c is the variant of filename.c in which we used a non-deterministic choice $([-\infty, +\infty])$ for the first initialization and input for the second. A non-termination bug found by Pulse, Dynamite or Ultimate-Automizer is denoted **▲**, an empty cell indicates that the tool is inconclusive. For FUNCTION-TR **✓** means that it proved Termination Resilience (returns TRUE), while **!(FP)** and **!(TP)** denotes the cases where FUNCTION-TR raised a false alarm³ and where it raised a true alarm⁴, respectively.

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

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 [28,36,17,5,21,22,20,19, etc.]. However, to

³ FUNCTION-TR raises an alarm on a program where Termination Resilience holds.

⁴ FUNCTION-TR raises an alarm on a program where Termination Resilience does not hold.

Test name	Pulse Dynamite Ultimate-Automizer	FUNCTION-TR
bresenham1-both-nt-ii	▲	!(TP)
bresenham1-both-nt-ir	▲	!(TP)
bresenham1-both-nt-ri	▲	!(TP)
bresenham1-both-nt-rr	▲	!(FP)
cohencl1-both-nt-i	▲	!(TP)
cohencl1-both-nt-r	▲	!(TP)
cohencl2-both-nt-i		!(TP)
cohencl3-both-nt-i		!(TP)
cohencl3-both-nt-r		!(TP)
cohencl4-both-nt-i		!(TP)
cohencl4-both-nt-r		!(TP)
cohencl5-both-nt-i	▲	!(TP)
cohencl5-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	▲	✓
dijkstra6-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)

Test name	Pulse Dynamite Ultimate-Automizer	FUNCTION-TR
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)
sqr11-both-nt-i	▲	!(TP)
sqr11-both-nt-r	▲	!(TP)
sqr12-both-nt-ii		!(TP)
sqr12-both-nt-ir		!(FP)
sqr12-both-nt-ri		!(FP)
sqr12-both-nt-rr		!(FP)

Fig. 16: Triage of non-termination alarms by FUNCTION-TR

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 [27,26] 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 ac-

commodate 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 [32,33] and inherits its imprecision (cf. Section 6). It also means that we could benefit from the extensions of the domain. Urban et al.[35] 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, \dots, p_n , a program satisfies $\langle\langle p_1, \dots, p_n \rangle\rangle \phi$ 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

In this paper, we have proposed a novel property, Termination Resilience, and an abstract interpretation-based static analysis to infer sufficient preconditions ensuring it. To this end, we have enhanced an existing abstract domain representing piecewise-defined functions with non-trivial transformer operators. Constructing a benchmark from SV-COMP 2024 and from the state-of-the-art, we have shown that our static analysis is able to reduce by up to 70% 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 decision tree abstract domain to analyze 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 Dangel, 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 Dangel, 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 Fix-points. 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 Malik, 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. 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.
27. 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.
28. Azalea Raad, Julien Vanegue, and Peter W. O’Hearn. Non-termination proving at scale. *Proc. ACM Program. Lang.*, 8(OOPSLA2):246–274, 2024.
29. Xiuhuan 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.
30. Alan Turing. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69, 1949.
31. 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.
32. Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs (Analyse Statique par Interprétation Abstraite de Propriétés Temporelles Fonctionnelles des Programmes)*. PhD thesis, École Normale Supérieure, Paris, France, 2015.

33. 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.
34. 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.
35. 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.
36. 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.

$$\begin{aligned}
\llbracket x \rrbracket \rho &\triangleq \{\rho(x)\} \\
\llbracket c \rrbracket \rho &\triangleq \{c\} \\
\llbracket -a \rrbracket \rho &\triangleq \{-n \mid n \in \llbracket a \rrbracket \rho\} \\
\llbracket a_1 \diamond a_2 \rrbracket \rho &\triangleq \{n_1 \diamond n_2 \mid n_1 \in \llbracket a_1 \rrbracket \rho, n_2 \in \llbracket a_2 \rrbracket \rho\} \\
\llbracket \text{input} \rrbracket \rho &\triangleq \mathbb{Z} \\
\llbracket [c, c'] \rrbracket \rho &\triangleq \{z \in \mathbb{Z} \mid c \leq z \leq c'\}
\end{aligned}$$

Fig. 17: Semantics $\llbracket e \rrbracket : \mathcal{E} \rightarrow \mathcal{P}(\mathbb{Z})$ of an expression $e \in \text{AExp} \cup \{\text{input}, [c_1, c_2]\}$.

$$\begin{aligned}
\mathfrak{T}[\text{skip}]^l T &\triangleq T \\
\mathfrak{T}[x := \text{input}]^l T &\triangleq T \cup \{x\} \\
\mathfrak{T}[x := [c_1, c_2]]^l T &\triangleq T \setminus \{x\} \\
\mathfrak{T}[x := a]^l T &\triangleq \begin{cases} T \cup \{x\} & \mathfrak{T}[a] T \\ T \setminus \{x\} & \text{otherwise} \end{cases} \\
\mathfrak{T}[\text{if } a \bowtie 0 \{s\}]^l T &\triangleq \begin{cases} \mathfrak{T}[s] T \cup \text{ASSIGNED}(s) \cup T & \mathfrak{T}[a] T \\ \mathfrak{T}[s] T \cup T & \text{otherwise} \end{cases} \\
\mathfrak{T}[\text{while } a \bowtie 0 \{s\}]^l T &\triangleq \text{lfp}_T \mathfrak{T}[\text{if } a \bowtie 0 \{s\}]^l \\
\mathfrak{T}[s_1; s_2] T &\triangleq \mathfrak{T}[s_2](\mathfrak{T}[s_1] T)
\end{aligned}$$

Fig. 18: Taint semantics $\mathfrak{T}[s] : \mathcal{P}(\mathbb{X}) \rightarrow \mathcal{P}(\mathbb{X})$, where $\text{ASSIGNED}(s)$ is the set of variables assigned with an arithmetic expression within the statement s .

A Arithmetic Expressions Semantics

The semantics $\llbracket e \rrbracket : \mathcal{E} \rightarrow \mathcal{P}(\mathbb{Z})$ of an expression $e \in \text{AExp} \cup \{\text{input}, [c_1, c_2]\}$ is defined in Figure 17.

B Taint Semantics

B.1 A Naïve Taint Semantics

The *taint semantics* $\mathfrak{T}[s] : \mathcal{P}(\mathbb{X}) \rightarrow \mathcal{P}(\mathbb{X})$ is defined in Figure 18.

Variable assignments add the assigned variable to the set of tainted variables if the assigning expression is an untrusted input ($^l x := \text{input}$) or ($^l x := a$) if its value depends on already tainted variables.

We define the taint semantics $\mathfrak{T}[a] : \mathcal{P}(\mathbb{X}) \rightarrow \{\text{true}, \text{false}\}$ of expressions as:

$$\mathfrak{T}[a] T \triangleq \exists \rho \in \mathcal{E} : \exists V \in \mathbb{Z}^{\mathbb{X}} : V \in \llbracket a \rrbracket \rho \wedge \forall \rho' \neq_T \rho : V \notin \llbracket a \rrbracket \rho' \quad (10)$$

where $\rho' \neq_T \rho \Leftrightarrow \exists \emptyset \subset T' \subseteq T : (\forall x \notin T' : \rho(x) = \rho'(x)) \wedge (\forall x \in T' : \rho(x) \neq \rho'(x))$ 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 $\llbracket a \rrbracket$ 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., $\mathfrak{T}\llbracket a \rrbracket T \triangleq \forall \rho, \rho' \in \mathcal{E} : \forall x \notin T : \rho(x) = \rho'(x) \wedge \llbracket a \rrbracket \rho \neq \llbracket a \rrbracket \rho'$) to also take into account the non-determinism arising from trusted variables (i.e., $\llbracket a \rrbracket \rho \neq \llbracket a \rrbracket \rho'$ could be the consequence of non-deterministic value choices rather than untrusted inputs). In the next section, we propose a more precise taint semantics that takes into account the reachable environments.

The taint semantics of compound statements (if and while), if their guard $a \bowtie 0$ is tainted, also adds to the tainted variables the set $\text{ASSIGNED}(s)$ of variables assigned with arithmetic expression $(^l x := a)$ in their body s . The taint semantics of while statements is the least fixpoint of the taint semantics of if statements: $\lambda X. \mathfrak{T}[\text{if } ^l a \bowtie 0 \{s\}] X$ is monotone and $\langle \mathcal{P}(\mathbb{X}), \subseteq \rangle$ is a finite ordered set, so the fixpoint exists. The taint semantics $\mathfrak{T}[p] \in \mathcal{P}(\mathbb{X})$ of a program $p \in \text{Prog}$ is then $\mathfrak{T}[p] = \mathfrak{T}[s^l] \triangleq \mathfrak{T}[s] \emptyset$.

Example 8 (Taint Semantics). Let us consider again the example programs in Figure 3. The set $\mathfrak{T}(2)$ of tainted variables at label 2 is $\{x\}$ for both programs.

For P_1 , the fixpoint iterations at label 3 add z because at label 4 it is assigned with an arithmetic expression that depends on already tainted variables ($\mathfrak{T}[z - x]\{x\}$ is true); the fixpoint is thus $\{x, z\}$.

For P_2 , the iterations at label 4 also add z but, this time, because the boolean condition of the loop depends on tainted variables ($\mathfrak{T}[z \geq x]\{x\}$ is true) and z belongs to $\text{ASSIGNED}(^5 z := z + c)$; the fixpoint is thus again $\{x, z\}$. \triangleleft

B.2 A More Precise Taint Semantics

A more precise taint semantics $\mathfrak{T}[s] : \mathcal{P}(\mathbb{X}) \times \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathbb{X}) \times \mathcal{P}(\mathcal{E})$ is defined in Figure 19: it additionally considers the set $R \in \mathcal{P}(\mathcal{E})$ of reachable environments at each program label. Accordingly, it relies on a more precise taint semantics of expressions:

$$\mathfrak{T}[a](T, R) \triangleq \exists \rho \in R, \exists V \in \mathbb{Z}^{\mathbb{X}} : \llbracket a \rrbracket \rho = V \wedge \forall \rho' \neq_T \rho, \llbracket a \rrbracket \rho' \neq V \quad (11)$$

An expression a is tainted if there exists a value of its semantics that is only possible in a *reachable* environment $\rho \in R$ with a certain value V for the set T of tainted variables, i.e., the value of a depends on untrusted program inputs.

The taint semantics $\mathfrak{T}[p] \in \mathcal{P}(\mathbb{X}) \times \mathcal{P}(\mathcal{E})$ of a program $p \in \text{Prog}$ is then $\mathfrak{T}[p] = \mathfrak{T}[s^l] \triangleq \mathfrak{T}[s](\emptyset, \mathcal{E})$. In this case, the lifted taint semantics $\mathfrak{T} : \mathcal{L} \rightarrow \mathcal{P}(\mathbb{X})$ mapping each program label to the set of variables that are tainted at that label is obtained by taking the first component of the pointwise-lifting of \mathfrak{T} with respect to the program labels \mathcal{L} .

$$\begin{aligned}
\mathfrak{T}[\text{skip}]^l(T, R) &\triangleq (T, R) \\
\mathfrak{T}[\text{input}]^l(T, R) &\triangleq T \cup \{x\} \\
\mathfrak{T}[x := a]^l(T, R) &\triangleq \text{let } T' = \begin{cases} T \cup \{x\} & \mathfrak{T}[a](T, R) \\ T \setminus \{x\} & \text{otherwise} \end{cases} \\
&\quad \text{in} \\
&\quad \text{let } R' = \{\rho[x \leftarrow v] \mid \rho \in R, v \in \llbracket a \rrbracket \rho\} \text{ in} \\
&\quad \text{in } (T', R') \\
\mathfrak{T}[\text{if }^l a \bowtie 0 \{s\}](T, R) &\triangleq \text{let } T', R' \triangleq \mathfrak{T}[s](T, \{\rho \in R \mid \exists v \in \llbracket a \rrbracket \rho, v \bowtie 0\}) \text{ in} \\
&\quad \begin{cases} T' \cup \text{ASSIGNED}(R') \cup T, R \cup R' & \mathfrak{T}[a](T, R) \\ T' \cup T, R \cup R' & \text{otherwise} \end{cases} \\
\mathfrak{T}[s_1; s_2](T, R) &\triangleq \mathfrak{T}[s_2](\mathfrak{T}[s_1](T, R)) \\
\mathfrak{T}[\text{while }^l a \bowtie 0 \{s\}](T, R) &\triangleq \text{lfp}_{\mathcal{T}}^{\subseteq} \mathfrak{T}[\text{if }^l a \bowtie 0 \{s\}](X, R)
\end{aligned}$$

Fig. 19: Taint semantics $\mathfrak{T}[\mathbf{s}] : \mathcal{P}(\mathbb{X}) \times \mathcal{P}(\mathcal{E}) \rightarrow \mathcal{P}(\mathbb{X}) \times \mathcal{P}(\mathcal{E})$.

C Denotational Termination Resilience Semantics

Example 9 (Variable Assignment Termination Resilience Semantics). Let us consider again the program P_2 in Figure 3b. Let

$$f \triangleq \lambda\rho. \begin{cases} 1 & \rho(z) < \rho(x) \\ \frac{2\rho(z) - 2\rho(x)}{-\rho(c)} + 3 & \rho(z) \geq \rho(x) \wedge \rho(c) < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

be the fixpoint at program label 4 and let us consider the non-deterministic variable assignment ${}^3c := [-2, 1]$. We have $\llbracket [-2, 1] \rrbracket \rho = \{-2, -1, 0, 1\}$ for any $\rho \in \mathcal{E}$ but, if $z \geq x$, $\rho[c \leftarrow v] \in \text{dom}(f)$ only when $v \in \llbracket [-2, 1] \rrbracket \rho$ is negative, i.e., when $v = -1$ or $v = -2$. Thus, the termination resilience semantics at label 3 is

$$\lambda\rho. \begin{cases} 2 & \rho(z) < \rho(x) \\ \rho(z) - \rho(x) + 4 & \text{otherwise} \end{cases}$$

since $\inf\{\frac{2\rho(z) - 2\rho(x)}{1} + 4, \frac{2\rho(z) - 2\rho(x)}{2} + 4\} = \rho(z) - \rho(x) + 3$. \triangleleft

Example 10 (While Statement Termination Resilience Semantics). Let us consider again the program P_1 in Figure 3a. The fixpoint iterations at label 3 start with the totally undefined function $f_3^0 \triangleq \emptyset$. Let f be the constant function equal to zero $\lambda\rho.0$. The first fixpoint iteration yields

$$f_3^1 \triangleq \lambda\rho. \begin{cases} 1 & \rho(z) < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

via the second condition defining $A_{\text{tr}}[\![\text{if}^l \mathbf{a} \bowtie 0 \{\mathbf{s}\}]\!]$. The assignment at label 4 yields the function

$$\lambda\rho. \begin{cases} 2 & \rho(z) - \rho(x) < 0 \\ \text{undefined} & \text{otherwise} \end{cases}$$

and the second fixpoint iteration thus yields

$$f_3^2 \triangleq \lambda\rho. \begin{cases} 1 & \rho(z) < 0 \\ 3 & 0 \leq \rho(z) < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

via the second condition defining $A_{\text{tr}}[\![\text{if}^l \mathbf{a} \bowtie 0 \{\mathbf{s}\}]\!]$ for $\rho(z) < 0$ and the first condition for $0 \leq \rho(z) < \rho(x)$. Finally,

$$f_3 \triangleq \lambda\rho. \begin{cases} 1 & \rho(z) < 0 \\ \frac{2\rho(z)}{\rho(x)} + 3 & 0 \leq \rho(z) \wedge 0 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

is the least fixpoint at program label 3. \triangleleft

Example 11 (Termination Resilience Semantics). Let us consider again the programs P_1 and P_2 in Figure 3.

For program P_1 , continuing Example 10, we obtain

$$\lambda\rho. \begin{cases} \frac{20}{\rho(x)} + 4 & 0 < \rho(x) \\ \text{undefined} & \text{otherwise} \end{cases}$$

at program label 2 and, finally, \emptyset at label 1, i.e., $A_{\text{tr}}[\![P_1]\!] \notin \mathcal{TR}$.

For program P_2 , continuing Example 9, we get

$$\lambda\rho. \begin{cases} 3 & 10 < \rho(x) \\ 15 - \rho(x) & \text{otherwise} \end{cases}$$

at label 2 and, finally $\lambda\rho.\omega$ at label 1, i.e., $A_{\text{tr}}[\![P_2]\!] \in \mathcal{TR}$. \triangleleft

D Proofs

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

Proof (Sketch). The proof uses Park's fixpoint induction principle, along the lines of the soundness and completeness proofs for the potential and definite termination semantics of Cousot and Cousot [12,32]. \square

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

Proof (Sketch). Let $C \triangleq \Lambda_{\text{tr}}[[^l x := [c_1, c_2]]\gamma(t)$ and let $A \triangleq \gamma(\text{ASSIGN}_T[[^l x := [c_1, c_2]]t)$. We prove that $\text{dom}(A) \subseteq \text{dom}(C)$ and $\forall \rho \in \text{dom}(A): C(\rho) \leq A(\rho)$.

Let us assume by absurd that $\text{dom}(C) \subset \text{dom}(A)$. Then, there exists $\rho \in \mathcal{E}$ such that $\rho \in \text{dom}(A)$ and $\rho \notin \text{dom}(C)$. Since $\rho \in \text{dom}(A)$ then, by definition of $\text{ASSIGN}_T[[^l x := [c_1, c_2]]$ (i.e., Algorithm 1 modified as discussed above), we have $\exists v \in [[c_1, c_2]]\rho: \rho[x \leftarrow v] \in \text{dom}(\gamma(t))$. Thus, by definition of $\Lambda_{\text{tr}}[[^l x := [c_1, c_2]]$ (cf. Figure 4), $\rho \in \text{dom}(C)$, which is absurd. Therefore, $\text{dom}(A) \subseteq \text{dom}(C)$.

Let us now assume by absurd that $\exists \rho \in \text{dom}(A): C(\rho) > A(\rho)$. By definition of $\Lambda_{\text{tr}}[[^l x := [c_1, c_2]]$, we have $C(\rho) = \inf \{ \gamma(t)(\rho[x \leftarrow v]) + 1 \mid v \in [[e]]\rho \}$. Moreover, by the definition of $\text{ASSIGN}_T[[^l x := [c_1, c_2]]$ and, notably, the definition of the resilience join \vee_F , we have $\inf \{ \gamma(t)(\rho[x \leftarrow v]) + 1 \mid v \in [[e]]\rho \} \leq A(\rho)$ since $\rho \in \text{dom}(A)$. Thus, $C(\rho) \leq A(\rho)$, which is absurd. Therefore, we have $\forall \rho \in \text{dom}(A): C(\rho) \leq A(\rho)$ and this concludes the proof. \square

Theorem 2 (Soundness). $\Lambda_{\text{tr}}[[P]] \preceq \gamma(\Lambda_{\text{tr}}^{\natural}[[P]])$

Proof (Sketch). The proof follows from Proposition 1 for $\Lambda_{\text{tr}}^{\natural}[[^l x := [c_1, c_2]]$. For all other statements, we refer to the proofs in [32] (cf. Remark 1). \square