Sound Symbolic Execution via Abstract Interpretation and its Application to Security

Ignacio Tiraboschi[2], Tamara Rezk[1], and Xavier Rival[2]

[1] INRIA Sophia Antipolis, France
[2] INRIA Paris, DI ENS, Ecole normale supérieure, Université PSL, CNRS
name.surname@inria.fr

Abstract. Symbolic execution is a program analysis technique commonly utilized to determine whether programs violate properties and, in case violations are found, to generate inputs that can trigger them. Used in the context of security properties such as noninterference, symbolic execution is precise when looking for counter-example pairs of traces when insecure information flows are found, however it is sound only up to a bound thus it does not allow to prove the correctness of programs with executions beyond the given bound. By contrast, abstract interpretation-based static analysis guarantees soundness but generally lacks the ability to provide counter-example pairs of traces. In this paper, we propose to weave both to obtain the best of two worlds. We demonstrate this with a series of static analyses, including a static analysis called RedSoundRSE aimed at verifying noninterference. RedSoundRSE provides both semantically sound results and the ability to derive counter-example pairs of traces up to a bound. It relies on a combination of symbolic execution and abstract domains inspired by the well known notion of reduced product. We formalize RedSoundRSE and prove its soundness as well as its relative precision up to a bound. We also provide a prototype implementation of RedSoundRSE and evaluate it on a sample of challenging examples.

Artifacts Available VMCAI

Artifacts Evaluated Functional VMCAI

1 Introduction

Security properties are notoriously hard to verify. In particular, many security properties are not single-execution properties but hyperproperties [13] (also referred to as relational properties), which means that refuting them sometimes requires *several* executions traces to be provided as a counter-example. In particular, noninterference [26] states that high clearance information should not impact the observation

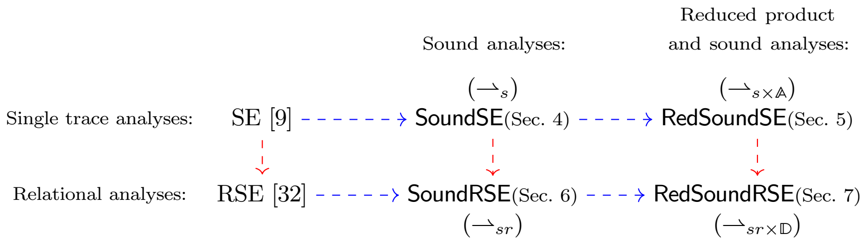

Fig. 1. Relation between different SE analyses. SE [9] is conventional symbolic execution and RSE [32, 34] is its extension to relational properties. Except for RSE with invariants [23], SE and RSE are unsound in general. The rest of the analyses are sound and are our contributions: SoundSE and SoundRSE do not use abstract interpretation whereas RedSoundSE and RedSoundRSE can be combined with different abstract domains. A red dashed line represents a dependency: a relational analysis depends on a single trace analysis. A blue dashed line represents an enhancement of the analysis.

of low clearance users in any execution of the program. It has been the subject of many verification method proposals and tools (e.g. [36, 7, 37, 24, 38, 8, 23, 4, 33]).

Symbolic execution [9, 31] (SE) is typically used to find property violations, and can be applied for policies like noninterference provided some adaptation for relational properties. SE boils down to an execution where variables initially hold symbolic values and get updated with expressions of these symbolic values whereas conditions are evaluated into symbolic path guards. The analysis involves an external tool such as an SMT solver that prunes infeasible paths and attempts to discharge verification conditions on remaining ones. SE attempts to exhaustively cover all executions paths, which is feasible only up to a bound and quickly turns out costly in presence of unbounded loops.

Conventional SE does not over-approximate executions after a fixed bound of iterations. This implies that *soundness* is lost when the program exceeds the exploration bound. Soundness ensures that, when the analysis concludes that the property of interest holds, the concrete semantics of the analyzed program is guaranteed to satisfy it. Since there is no over-approximation, when the property is violated by traces shorter than the exploration bound, tools like SMT solvers can provide instances for the symbolic values and enable the reconstruction of counter-example traces. This is of particular importance to security in order to confirm security violations. We refer to such counter-examples as **refutation models**.

The adaptation of SE to handle relational properties [32, 34] requires to track several traces instead of just one. In the following, we will call this adaptation *relational symbolic execution* (or RSE). Previous work [23] has shown how to combine RSE with loop invariants, provided by the developer, in order to recover soundness at the cost of annotations and loss of precision when invariants are not strong enough.

Abstract interpretation based static analyses [14] (AI) rely on an abstraction defined as a logical approximation relation between concrete behaviors and abstract predicates and produce sound over-approximations of program semantics at the cost of completeness. However, the over-approximation entails that the analysis may fail to conclude positively even when analyzing correct programs. Moreover, most static analysis implementations lack the ability to synthesize counter-example traces.

In this paper, we formalize a combined analysis technique, which aims at bringing together advantages of both symbolic execution and abstract interpretation, in a security setup. We first show how to over-approximate SE in order to keep soundness, and call this analysis SoundSE (see Figure 1). We use SoundSE to show the combination for conventional SE and different abstract domains, calling the resulting analyses RedSoundSE. Our analysis for relational properties is called RedSoundRSE and targets noninterference. It borrows path exploration from relational symbolic execution, parameterized by RedSoundSE, and relies on abstract interpretation based static analysis to report a sound result for all programs. Abstraction enables the early pruning of infeasible paths and the computation of sound over-approximations for program behaviors when the exploration bound is exhausted. To achieve this, RedSoundRSE automatically injects loop invariants computed by abstract domains into a relational store. Not only dependence analysis results can be used to fill security related information where the symbolic execution cannot explore paths fully but also (e.g., numerical) state abstraction information allows to improve the symbolic information extracted from the dependency analysis. Moreover, our analysis allows switching between different abstractions, and tuning specific settings, e.g., loop unroll depth (depth up to which SE is kept precise), which allows the user to change the balance between cost and precision. To summarize, we propose symbolic execution based verification methods that are sound and precise, providing refutation models up to a bound. Our contributions, illustrated in Figure 1, are the following:

1. SoundSE and RedSoundSE: We define a sound SE analysis, and we integrate numerical abstract domains into it to prune reachable paths. As a result, we make SE [9] sound while keeping the ability of the analysis to find counter-examples.
2. SoundRSE and RedSoundRSE: We define a sound relational SE, and we combine it with dependence analysis [4] to enhance the precision of the latter while preserving soundness.
3. We prototype RedSoundRSE together with RedSoundSE in OCaml and show, using a series of challenging examples, that it is able to both soundly decide noninterference for secure programs and synthesize counter-examples of a size up to a given bound for insecure ones.

The structure of the paper is as follows. Section 2 defines a basic language and the noninterference notion used throughout the rest of the paper. Section 3 provides an overview on already defined analyses and highlights the main principles of RedSoundRSE. Section 4 defines SoundSE, a sound single trace symbolic execution that serves as a basis for RedSoundRSE and Section 5 presents RedSoundSE, a new combination of SoundSE with state abstraction. Section 6 presents SoundRSE and Section 7 extends it with a dependence abstraction to obtain RedSoundRSE. Section 8 evaluates our framework on small but challenging examples. Finally, Section 9 discusses related work and Section 10 concludes. The appendix contains all rules of analyses in the paper.

2 Language and noninterference security notion

In this section, we introduce the language and security notion for which we formalize our analyses. We let $\mathbb{V}$ and $\mathbb{X}$ be the set of values and program variables respectively, and $\oplus$, $\olessthan$ be binary operators. A boolean expression $\mathbf{b}$ is a comparison operator $\olessthan$ applied to two expressions and evaluates to a boolean value $\mathbb{B}=\{\mathbf{tt},\mathbf{ff}\}$. A statement $\mathbf{s}$ is either a skip, an assignment, a condition, or a loop. Finally, a command $\mathbf{c}$ is a finite sequence of statements. A program P is a pair $(\mathbf{c},L)$ made of a command $\mathbf{c}$ (the body of the program), and a set of low variables $L\subseteq\mathbb{X}$, hence publicly observable (the other variables occurring in the program are high).

$$\mathbf{e} ::= v\ (v\in\mathbb{V}) \mid x\ (x\in\mathbb{X}) \mid \mathbf{e}\oplus\mathbf{e} \qquad\qquad \mathbf{b} ::= \mathbf{e}\olessthan\mathbf{e}$$
$$\mathbf{s} ::= \texttt{skip} \mid x := \mathbf{e} \mid \texttt{if}\ \mathbf{b}\ \texttt{then}\ \mathbf{c}\ \texttt{else}\ \mathbf{c} \mid \texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c} \qquad\qquad \mathbf{c} ::= \mathbf{s} \mid \mathbf{s};\mathbf{c}$$

Semantics. Given a program $(\mathbf{c},L)$, a *state* is a pair $(\mathbf{c},\mu)$, where $\mathbf{c}$ is a command and μ is a function from $\mathbb{X}$ to $\mathbb{V}$, namely a *store*. In particular, a state of the form $(\texttt{skip},\mu)$ is final. We write $\mathbb{M}$ and $\mathbb{S}$ for the set of stores and states respectively. We use $[\texttt{x}\mapsto x,\texttt{y}\mapsto y,...]$ to explicitly enumerate a store's contents, where $x,y,...$ are concrete values. Let $(\rightarrow)\subseteq\mathbb{S}\times\mathbb{S}$ denote the small step operational semantics (which is standard) and $\rightarrow^*$ be its reflexive transitive closure.

Noninterference. Let $=_L$ be the set equality of stores restricted to low variables in L. In the rest of the paper, we focus on termination-insensitive noninterference:

Definition 1 (Termination-insensitive noninterference). *A program* $(\boldsymbol{c},L)$ *is* termination-insensitive noninterferent*, written as* $\mathcal{NI}_{\mathsf{P}}^{T.I}$*, if and only if, for all stores* $\mu_0,\mu_1,\mu_0',\mu_1'\in\mathbb{M}$*,* $\mu_0=_L\mu_1\wedge(\boldsymbol{c},\mu_0)\rightarrow^*(\texttt{skip},\mu_0')\wedge(\boldsymbol{c},\mu_1)\rightarrow^*(\texttt{skip},\mu_1')\Longrightarrow\mu_0'=_L\mu_1'$.

3 Overview

In this section, we demonstrate the principle of the combination of symbolic execution and abstraction performed by RedSoundRSE so as to overcome the limitation of these two approaches taken separately. As in the rest of the paper, we focus on noninterference (NI), although the same principle would apply to other security properties as well.

Examples. We consider the programs displayed in Figure 2. Essentially, programs (a) and (b) are secure with respect to the noninterference policy, where `priv` is high and all other variables are low, whereas (c) is not secure.

In program 2(a), variable `y` gets assigned 5 independently of `priv`, therefore the program is secure. For Program 2(b), let μ_0,μ_1 be two stores such that $\mu_0=_L\mu_1$. Since μ_0 and μ_1 are low-equal executions cannot take different paths, and the loop will be executed the same amount of times. Therefore, the program is secure. Lastly, Program 2(c) is insecure, meaning that it does not satisfy noninterference. We need to provide a counter-example consisting of two executions starting from low-equal stores μ_0,μ_1 such that the corresponding output stores μ_0',μ_1' are not low-equal. We consider the following

```
if (priv > 0)
  y = 5;
else
  y = 5;
```

(a) Secure program

```
while (i < z) {
  i = i + 1;
  priv = priv + 5;
}
```

(b) Secure program

```
while (i > priv) {
  i = i + 1;
  priv = priv + 2;
}
```

(c) Insecure program

```
if (priv < 0) priv = 0;
while (i < 10){
  i += 1; priv += 2;
}
if (priv >= 0) y += 1;
else y = 0;
```

(d) A secure program requiring a numerical domain.

Fig. 2. Example programs. All variables are of type `int`, where variable is `priv` is secure.

	Secure?	RSE	Dependence analysis	RedSoundRSE
Program 2(a)	Yes	✓ Secure	✗ False alarm	✓ Secure
Program 2(b)	Yes	✗ False alarm	✓ Secure	✓ Secure
Program 2(c)	No	✓ Refutation model	✓ Alarm	✓ Refutation model

Table 1. Analysis results compared. Symbol ✓ (resp., ✗) denotes a semantically correct (resp., incorrect) analysis outcome, with either a proof of security, a (possibly false) alarm, or a refutation model.

stores: $\mu_0 = [\mathtt{i} \mapsto 0, \mathtt{priv} \mapsto 0]$, and $\mu_1 = [\mathtt{i} \mapsto 0, \mathtt{priv} \mapsto -1]$. Finally, calculated output stores are such that $\mu'_0(\mathtt{i}) = 0 \neq \mu'_1(\mathtt{i}) = 1$, thus the program violates noninterference.

In the next paragraphs we study the result of verification methods for these three programs.

Verification based on relational symbolic execution. A symbolic store, referred to as ρ, maps variables to symbolic expressions of the initial values of the variables. To avoid confusion, we use an italic typewriter font for these symbolic values while program variables appear in straight typewriter font. For instance, $\mathtt{\textit{y}}$ denotes the initial value of `y`. Relational symbolic execution describes pairs of executions using symbolic conditions over the initial values of variables and pairs of symbolic stores. Symbolic stores are not enough to abstract executions, since they cannot express constraints. Constraints are then provided by a *symbolic path* π that contextualizes the store. A pair (ρ, π) of a symbolic store and a symbolic path is referred to as a *symbolic precise store*.

As an example, we consider Program 2(a). Relational symbolic execution uncovers four pairs of paths depending on the sign of the initial values of `priv` in both executions. For instance, one of the diverging paths produces $\pi = (\mathtt{\textit{priv}}_0 > 0 \wedge \mathtt{\textit{priv}}_1 \leq 0) \Longrightarrow ([\mathtt{y}_0 \mapsto 5, \ldots], [\mathtt{y}_1 \mapsto 5, \ldots])$, where $\mathtt{y}_0$ and $\mathtt{y}_1$ denote the program variable `y` in both executions and $\mathtt{\textit{priv}}_0, \mathtt{\textit{priv}}_1$ the initial symbolic values of `priv`. This symbolic precise store shows no information flow to `y` since any SMT solver can prove $\mathtt{y}_0 = \mathtt{y}_1$. The other three pairs of paths lead to a similar result, thus the program is proved secure.

For Program 2(b), the loop has an unbounded number of iterations, but relational symbolic execution can only cover finitely many unrollings of the loop. This prevents RSE to prove that Program 2(b) is secure.

For Program 2(c), RSE will only explore the loop up to a bound. Assuming the bound is one (any positive value would prove similar), it can determine that the program does not satisfy NI by calculating a concrete trace that violates the property. This counter-example trace is calculated by an SMT solver, for instance $\boldsymbol{i}_0 = \boldsymbol{i}_1 = 0$, $\boldsymbol{priv}_0 = 1$ and $\boldsymbol{priv}_1 = -1$ corresponds to the counter-example given previously.

Verification based on dependence abstraction. Many static analyses that work for noninterference rely on some form of dependence abstraction as formalized in, e.g., [4] or [28]. We briefly summarize the abstraction of [4]. We assume an ordered set of security levels $\{\mathbb{L},\mathbb{H}\}$ and that each value fed into a program via an input variable is given a security level. A dependency, noted as $l \rightsquigarrow \texttt{x}$ with $l \in \{\mathbb{L},\mathbb{H}\}$, expresses the agreement of `x` in both executions when observing from level l. This analysis, based on abstract interpretation, is *sound*.

We now discuss the analysis of some programs in Figure 2. For Program 2(a), the analysis determines that the assignments are conditioned by the value of `priv`, which is initially high. Then, the dependency $\mathbb{L} \rightsquigarrow \texttt{y}$ is dropped, indicating that `y` can potentially disagree between executions. In Program 2(b), the loop condition is only influenced by `i` and `z`, which are low. Then, the assignment of low variables is not affected, and `i` and `z` remain low, allowing to prove noninterference.

Lastly, Program 2(c) is not secure, and since dependence analysis is sound, the analysis discards dependency $\mathbb{L} \rightsquigarrow \texttt{i}$ based on the illicit flow of information.

Combination of relational symbolic execution and dependence abstraction. As observed in Table 1, relational symbolic execution fails to handle precisely program 2(b) whereas dependence abstraction fails to verify program 2(a) and provides no counter-example for program 2(c). The purpose of RedSoundRSE is to use both techniques in an alternating manner in order to increase precision and prune branches.

To achieve this, RedSoundRSE borrows from relational symbolic execution the precise analysis of assignment and condition commands, as well as the unrolled iterates of loop commands. In particular, the analysis of programs 2(a) and 2(c) is carried out as shown above. However, when the unrolling bound is reached, dependence analysis is used as a means to compute in finite time sound information about any number of further loop iterations. Indeed, when the dependence information proves that a loop induces no dependency of a given low variable on any high variable, it is possible to assume the equality of the variable in the symbolic store. This new value may not be expressed precisely in terms of the initial values, hence it may be approximated with a fresh symbol. This occurs for variable `i` in program 2(b).

As seen, RedSoundRSE analyzes the first three examples of Figure 2 precisely.

Refinement of symbolic execution based on state abstraction. Program Figure 2(d), previously not considered, cannot be proved NI by just using symbolic execution and dependence analysis. This program is secure since the assignment of `i` does not depend on `priv`, and `y` is conditioned by `priv` which is always positive after the loop.

As in Program 2(b), the loop causes the symbolic execution to stop at the unrolling bound. Dependence information allows to prove that there is no information flow to `i` and also that the value of `y` at line 8 does not depend on `priv`. However, the condition at line 9 depends on `priv`, thus dependence analysis will not prove that the assignments at lines 10 and 12 do not leak information. Symbolic execution does not succeed either as it lacks the ability to reason over the value of `priv` at the loop exit.

Such information may be computed using a reachability static analysis. In particular, a classical static analysis based on the abstract domains of intervals [14] computes ranges for all numeric variables and concludes in this case that `priv` is positive, hence only the true branch of the condition may be taken. Integrating non-relational abstract domains allows the analyzer to increase precision by automatically pruning paths.

This combination of AI and SE is referred to as RedSoundSE, defined in Section 5, and is later integrated into the final analysis RedSoundRSE.

4 SoundSE: Sound symbolic execution

We now define a type of symbolic execution, named SoundSE, as it serves as a basis for not only SoundRSE but also RedSoundSE—the product of SoundSE with abstract domains.

Symbolic execution states. The core principle of symbolic execution is to map program variables into expressions made of *symbolic values* that denote the initial value of the program variables. We let $\overline{\mathbb{V}} = \{\boldsymbol{x}, \boldsymbol{y}, ...\}$ denote the set of symbolic values and note for clarity $\boldsymbol{x}$ the symbolic value associated to program variable $\mathtt{x}$ (not to be confused with concrete values). A *symbolic store* is a function ρ from program variables to *symbolic expressions* the set of which is noted $\mathbb{E}$, namely expressions defined like the programming language expressions using symbolic values instead of program variables. We write $\overline{\mathbb{M}} = \mathcal{P}(\mathbb{X} \to \mathbb{E})$ for the set of symbolic stores and write $[\mathtt{x} \rightsquigarrow \langle \boldsymbol{x} \rangle, ...]$ for an explicitly given symbolic store. To tie properly symbolic stores and concrete stores, we need to relate symbolic values and concrete values. To this end, we let a *valuation* be a function $\nu : \overline{\mathbb{V}} \longrightarrow \mathbb{V}$. Moreover, given a symbolic expression ε, we let $[\![\varepsilon]\!]$ be a partial function that maps a valuation ν to the value obtained when evaluating the expression obtained by replacing each symbolic value $\boldsymbol{x}$ in e with $\nu(\boldsymbol{x})$. We can now express the concretization of symbolic stores:

Definition 2 (Symbolic store concretization). *The* symbolic store concretization*,* $\gamma_{\overline{\mathbb{M}}} : \overline{\mathbb{M}} \longrightarrow \mathcal{P}(\mathbb{M} \times (\overline{\mathbb{V}} \to \mathbb{V}))$*, maps a symbolic store to the set of pairs made of a store and a valuation that realize it, i.e.* $\gamma_{\overline{\mathbb{M}}}(\rho) = \{(\mu, \nu) \mid \forall \mathtt{x} \in \mathbb{X}, \mu(\mathtt{x}) = [\![\rho(\mathtt{x})]\!](\nu)\}$.

To precisely characterize the outcome of an execution path, a symbolic store is too abstract. Hence, SE also utilizes a symbolic expression to constrain the store, referred to as *symbolic path*, that accounts for the conditions encountered during a path. A *symbolic precise store* is a pair $\kappa = (\rho, \pi)$ where $\rho \in \overline{\mathbb{M}}$ and π is a symbolic path. We write $\mathbb{K}$ for the set of symbolic precise stores. Their meaning is defined as follows:

Definition 3 (Symbolic precise store concretization). *The* symbolic precise store concretization*,* $\gamma_{\mathbb{K}} : \mathbb{K} \longrightarrow \mathcal{P}(\mathbb{M} \times (\overline{\mathbb{V}} \to \mathbb{V}))$*, is defined by* $\gamma_{\mathbb{K}}(\rho, \pi) = \{(\mu, \nu) \in \gamma_{\overline{\mathbb{M}}}(\rho) \mid [\![\pi]\!](\nu) = \boldsymbol{tt}\}$.

$$\textsc{s-assign}\ \frac{(\mathbf{e},\rho)\vdash_{\mathrm{s}}\varepsilon}{(\mathtt{x}:=\mathbf{e},(\rho,\pi))\rightharpoonup_{s}(\mathtt{skip},(\rho[\mathtt{x}\rightsquigarrow\langle\varepsilon\rangle],\pi))}$$

$$\textsc{s-if-t}\ \frac{(\mathbf{b},\rho)\vdash_{\mathrm{s}}\beta \qquad \pi'\triangleq\pi\wedge\beta \qquad \mathbf{may}(\pi')}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1,(\rho,\pi))\rightharpoonup_{s}(\mathbf{c}_0,(\rho,\pi))}$$

$$\textsc{s-loop-t}\ \frac{(\mathbf{b},\rho)\vdash_{\mathrm{s}}\beta \qquad \pi'\triangleq\pi\wedge\beta \qquad \mathbf{may}(\pi')}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c},(\rho,\pi))\rightharpoonup_{s}(\mathbf{c};\ \mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c},(\rho,\pi))}$$

$$\textsc{s-loop-f}\ \frac{(\mathbf{b},\rho)\vdash_{\mathrm{s}}\beta \qquad \pi'\triangleq\pi\wedge\neg\beta \qquad \mathbf{may}(\pi')}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c},(\rho,\pi))\rightharpoonup_{s}(\mathtt{skip},(\rho,\pi))}$$

Fig. 3. Symbolic execution step relation: a few selected rules

Example 1 (Symbolic precise store). We consider Program 2(a). Symbolic execution needs to cover two paths corresponding to each of the branches of the condition statement, i.e., depending on the sign of $\boldsymbol{priv}$. Therefore, symbolic execution should produce the precise stores $(\rho_0, \boldsymbol{priv} > 0)$ and $(\rho_1, \boldsymbol{priv} \le 0)$, where $\rho_0 = \rho_1 = [\mathtt{y}\rightsquigarrow\langle 5\rangle, \mathtt{priv}\rightsquigarrow\langle\boldsymbol{priv}\rangle]$.

Symbolic execution step. The main piece of the symbolic execution algorithm is the step relation, which closely follows the small step semantics of the programs. We define it by a transition relation $\rightharpoonup_s$ between *symbolic execution states* that are made of a program command and a symbolic precise store. Before we write down the analysis $\rightharpoonup_s$, we need a few definitions.

First, we define the symbolic evaluation of an expression or condition in a symbolic store, which produces a symbolic expression. We note $(\mathbf{e},\rho)\vdash_{\mathrm{s}}\varepsilon$ the evaluation of $\mathbf{e}$ into symbolic expression ε in symbolic store ρ. Usually, this evaluation step boils down to the substitution of the variables in $\mathbf{e}$ with the symbolic expressions they are mapped to in ρ, possibly with some simplifications.

Second, we define the conservative satisfiability test of a symbolic path. This step is usually performed by an external tool such as an SMT solver, so we do not detail its internals here. We note that this test may conservatively return as a result that a symbolic path *may* be satisfiable. We note $\mathbf{may}(\pi)$ when π may be satisfiable.

We now turn to the rules in Figure 3. Rule s-assign simply updates the symbolic store with a new symbolic expression for the assigned variable. In rule s-if-t, if the guard evaluation β is satisfiable, the true branch is accessed and β is added to the symbolic path. Finally, rules s-loop-t and s-loop-f follow similar principles as rule s-if-t in the case of loops. We formalize the soundness of execution steps:

Theorem 1 (Soundness of a single symbolic execution step). *Let $(\boldsymbol{c},\mu)$ and $(\boldsymbol{c}',\mu')\in\mathbb{S}$ be two states such that $(\boldsymbol{c},\mu)\rightarrow(\boldsymbol{c}',\mu')$, $\kappa\in\mathbb{K}$ a symbolic precise store, and ν be a valuation such that $(\mu,\nu)\in\gamma_{\mathbb{K}}(\kappa)$. Then, there exists a symbolic precise store κ' such that $(\mu',\nu)\in\gamma_{\mathbb{K}}(\kappa')$ and $(\boldsymbol{c},\kappa)\rightharpoonup_s(\boldsymbol{c}',\kappa')$.*

Sound depth bounded symbolic execution. Clearly, the exhaustive application of the symbolic execution step relation defined in Figure 3 would not terminate. Therefore, common symbolic execution tools typically abort the exploration when they reach some

$$\textsc{s-next}\ \frac{(\mathbf{c},\kappa)\rightharpoonup_s(\mathbf{c}',\kappa') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w)=(\mathbf{tt},w')}{(\mathbf{c},\kappa,w,b)\rightharpoonup_s(\mathbf{c}',\kappa',w',b)}$$

$$\textsc{s-approx-many}\ \frac{(\mathbf{c},\kappa)\rightharpoonup_s(\mathbf{c}',\kappa') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w)=(\mathbf{ff},w') \qquad \rho''=\mathfrak{modif}(\rho,\mathbf{c})}{(\mathbf{c},(\rho,\pi),w,b)\rightharpoonup_s(\mathtt{skip},(\rho'',\pi),w',\mathbf{ff})}$$

Fig. 4. SoundSE: Sound bounded symbolic execution step relation

sort of bound on execution lengths. This result is clearly unsound as longer executions are simply ignored. Alternatively, it is possible to over-approximate the set of precise stores that may be reachable when the bound is met. We formalize this approach here.

Essentially, symbolic states need to be augmented with two additional pieces of information, namely a boolean so-called *precision flag* which states whether symbolic execution has performed any over-approximation due to exhausting the bound, and a bound control field, called *counter*. We define set $\mathbb{W}$ as the set of counters, with a special element $w_0 \in \mathbb{W}$ that denotes the initial counter status with respect to bound control. To operate over counters, we require a function $\mathfrak{step}$ which inputs two commands $\mathbf{c}$, $\mathbf{c}'$, and a counter w. It produces a result of the form (b,w') where b is a boolean, and w' is the next counter. Value b is $\mathbf{tt}$ if and only if a step from $\mathbf{c}$ to $\mathbf{c}'$ can be done without exhausting the iteration bounds, and with the new counter w'. If b is $\mathbf{ff}$, the iteration bound has been reached and the state needs to be over approximated.

To perform the over approximation, a function $\mathfrak{modif}$ is required. The function inputs a symbolic store and a command, and returns a new symbolic store ρ' such that:

- ρ' maps each program variable that is considered to be "modified" (by a sound over approximation of the set) in $\mathbf{c}$ to a *fresh* symbolic value;
- ρ' maps all the other program variables to their image in the original store.

Example 2 (Loop iteration bounding). The most typical way to bound symbolic execution limits the number of iteration of each loop to pre-defined number k. Then, $\mathbb{W}$ consists of stacks of integers, w_0 is the empty stack, and $\mathfrak{step}$ adds a zero on top of the stack when entering a new loop and pops the value on top of the stack when exiting a loop. More importantly, it increments the value n at the top of the stack when $n \leq k$ and moving to the next iteration (rule s-loop-t); on the other hand, when $n > k$, it pops n and returns the $\mathbf{ff}$ precision flag.

To ensure termination, $\mathbb{W}$ and $\mathfrak{step}$ should satisfy the following *well-foundedness* property: for any infinite sequence of commands $(\mathbf{c}_i)_i$ the infinite sequence $(w_i)_i$ defined by $\mathfrak{step}(\mathbf{c}_i,\mathbf{c}_{i+1},w_i)=(\mathbf{tt},w_{i+1})$ should be stationary, which we assume here.

Based on these definitions, *depth bounded symbolic execution* is defined by a transition relation over 4-tuples made of a command, a symbolic state, an element of $\mathbb{W}$, and a boolean, referred to as symbolic state. We overload the notation $\rightharpoonup_s$ for this relation, which is defined based on the previously defined $\rightharpoonup_s$. The rules are provided in Figure 4:

– Rule s-next carries out an atomic step of symbolic execution that requires no over approximation; function $\mathfrak{step}$ returns the precision flag b and a new counter;

– Rule s-approx-many carries out a global approximation step; indeed, as $\mathfrak{step}$ returns $\mathbf{ff}$, the function $\mathfrak{modif}$ is applied to the symbolic state to over-approximate

the effect of an arbitrary number of steps of execution of **c**; alongside with the new counter state the **ff** precision is propagated forward.
Under the well-foundedness assumption, exhaustive iteration of the available symbolic execution rules from any initial symbolic state will terminate and produce finitely many symbolic states. To express the soundness of this algorithm, we need to account for the creation of symbolic values by function $\mathfrak{modif}$, which means that valuations also need to be extended. To this end, we note $\nu \preceq \nu'$ when the domain of valuation ν is included into that of ν' and when both ν and ν' agree on the intersection of their domains. We now obtain the following soundness statement:

Theorem 2 (Soundness of any sequence of single symbolic execution steps). *Let* $(\boldsymbol{c},\mu)\in\mathbb{S}$ *be a state and* μ' *be a store such that* $(\boldsymbol{c},\mu)\rightarrow^*(\texttt{skip},\mu')$. *Let* $\kappa\in\mathbb{K}$ *be a symbolic precise store and* ν *be a valuation such that* $(\mu,\nu)\in\gamma_{\mathbb{K}}(\kappa)$. *Let* $w\in\mathbb{W}$ *be a counter. Then, there exists a symbolic precise store* κ'*, a valuation* ν'*, and a counter* $w'\in\mathbb{W}$ *such that* $\nu\preceq\nu'$*,* $(\mu',\nu')\in\gamma_{\mathbb{K}}(\kappa')$*, and* $(\boldsymbol{c},\kappa,w,b)\rightharpoonup_s^*(\texttt{skip},\kappa',w',b')$*.*

The proof of this theorem follow from Theorem 1 (steps where $\mathfrak{step}$ returns **tt**), and a global induction on the command **c** when rule S-APPROX-MANY applies.

Example 3 (Symbolic execution). For program 2(a), symbolic execution returns the symbolic stores shown in Example 1. We assume the bounding of Example 2 and consider program 2(b). Then, symbolic execution generates the symbolic store $[\mathtt{z}\rightsquigarrow\langle\boldsymbol{z}\rangle,\mathtt{i}\rightsquigarrow\langle\boldsymbol{i}'\rangle,\mathtt{priv}\rightsquigarrow\langle\boldsymbol{priv}'\rangle]$ with precision flag **ff**, and where $\boldsymbol{i}'$, $\boldsymbol{priv}'$ are fresh symbolic values generated by rule S-APPROX-MANY.

Refutation up to a bound. A very desirable feature of symbolic execution is the ability to produce counter-examples up to a bound. This feature stems from a bounded refutation result, which states that, when symbolic execution produces a final state for which the final precision flag is **tt**, and such that the symbolic path is satisfiable, then a matching concrete execution can be found. From the final state, the SMT solver can compute a refutation model.

Theorem 3 (Refutation up to a bound). *Let* **c** *be a command,* $\kappa,\kappa'\in\mathbb{K}$ *be two precise stores,* $w,w'\in\mathbb{W}$*, such that* $(\boldsymbol{c},\kappa,w,\boldsymbol{tt})\rightharpoonup_s^*(\texttt{skip},\kappa',w',\boldsymbol{tt})$*. Then, for all* $(\mu',\nu')\in\gamma_{\mathbb{K}}(\kappa')$*, it exists* $(\mu,\nu)\in\gamma_{\mathbb{K}}(\kappa)$ *such that* $(\boldsymbol{c},\mu)\rightarrow^*(\texttt{skip},\mu')$*.*

This result follows from the fact that rule S-APPROX-MANY is never applied in the symbolic execution and from an induction on the sequence of S-NEXT steps.

Example 4 (Symbolic execution completeness up to a bound). We consider the cases discussed in Example 3. Using the bounding of Example 2, the result produced for program 2(a) is complete whereas that for program 2(b) generates some final symbolic state with precision flag **ff**, hence for which Theorem 3 does not apply.

5 RedSoundSE: Sound SE combined with abstract states

We now extend SoundSE with the ability to use the properties inferred by abstract interpretation. This combined symbolic execution is referred to as RedSoundSE, making reference to the reduced product between SoundSE and an AI based analysis.

$$\text{A-ASSIGN}\ \frac{a' \triangleq \mathfrak{assign}_{\mathtt{x},\mathbf{e}}(a)}{(\mathtt{x}:=\mathbf{e},a) \rightharpoonup_{\mathbb{A}} (\mathtt{skip},a')} \qquad \text{A-IF-T}\ \frac{a' \triangleq \mathfrak{guard}_{\mathbf{b}}(a) \qquad a' \neq \bot}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1,a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}_0,a')}$$

(a) Abstract execution step selected rules

$$\text{S-A-NEXT}\ \frac{\begin{array}{c}(\mathbf{c},\kappa,w,b) \rightharpoonup_s (\mathbf{c}',\kappa',w',b)\\ \mathfrak{step}(\mathbf{c},\mathbf{c}',w)=(\mathbf{tt},w') \qquad (\mathbf{c},a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}',a') \qquad (\kappa'',a'') \triangleq \mathfrak{reduction}(\kappa',a')\end{array}}{(\mathbf{c},\kappa,a,w,b) \rightharpoonup_{s\times\mathbb{A}} (\mathbf{c}',\kappa'',a'',w',b)}$$

$$\text{S-A-APPROX-MANY}\ \frac{\begin{array}{c}(\mathbf{c},\kappa) \rightharpoonup_s (\mathbf{c}',\kappa') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w)=(\mathbf{ff},w')\\ \kappa''=\mathfrak{modif}(\kappa,\mathbf{c}) \qquad a'=[\![\mathbf{c}]\!]^{\sharp}_{\mathbb{A}}(a) \qquad (\kappa''',a''') \triangleq \mathfrak{reduction}(\kappa'',a')\end{array}}{(\mathbf{c},(\kappa,a),w,b) \rightharpoonup_{s\times\mathbb{A}} (\mathtt{skip},(\kappa''',a'''),w',\mathbf{ff})}$$

(b) Product of symbolic execution and static analysis

Fig. 5. Abstract execution step and product with symbolic execution

Abstraction of store and static analysis. In the following, we assume that an *abstract domain* [14] $\mathbb{A}$ describing sets of stores is fixed, together with a concretization function $\gamma_{\mathbb{A}}:\mathbb{A}\longrightarrow\mathcal{P}(\mathbb{M})$. We assume the existence of an element $\bot\in\mathbb{A}$ such that $\gamma_{\mathbb{A}}(\bot)=\emptyset$. Additionally, we require the two following sound abstract post-condition functions for basic operations. Function $[\![\ ]\!]$ will be overloaded to replace any variable $\mathtt{x}$ for its mapped value in a store $\mu(\mathtt{x})$:

– *abstract assignment* $\mathfrak{assign}_{\mathtt{x},\mathbf{e}}:\mathbb{A}\longrightarrow\mathbb{A}$ is parameterized by a variable $\mathtt{x}$ and an expression $\mathbf{e}$ and is such that $\forall a\in\mathbb{A},\ \{\mu[\mathtt{x}\mapsto[\![\mathbf{e}]\!](\mu)]\mid\mu\in\gamma_{\mathbb{A}}(a)\}\subseteq\gamma_{\mathbb{A}}(\mathfrak{assign}_{\mathtt{x},\mathbf{e}}(a))$.

– *abstract condition* $\mathfrak{guard}_{\mathbf{b}}:\mathbb{A}\longrightarrow\mathbb{A}$ is parameterized by a boolean expression $\mathbf{b}$ and is such that $\forall a\in\mathbb{A},\ \{\mu\in\gamma_{\mathbb{A}}(a)\mid[\![\mathbf{b}]\!](\mu)=\mathbf{tt}\}\subseteq\gamma_{\mathbb{A}}(\mathfrak{guard}_{\mathbf{b}}(a))$.

Based on these operations, the definition of a *sound abstract execution step* relation $\rightharpoonup_{\mathbb{A}}$ is straightforward. We show two rules in Figure 5(a). The rules match those of $\rightarrow$ (Section 2) and are sound with respect to it. In the following, $\mathbb{A}$ is assumed to be a parameter of the analysis. It may consist of any numerical abstraction, such as the interval abstract domain [14] or the domain of convex polyhedra [16]. Moreover, the application of standard widening technique [14] allows to define a *static analysis* function $[\![\mathbf{c}]\!]^{\sharp}_{\mathbb{A}}:\mathbb{A}\longrightarrow\mathbb{A}$ that is sound in the sense that, for all command $\mathbf{c}$ and all abstract state a, $\{\mu'\in\mathbb{M}\mid\exists\mu\in\gamma_{\mathbb{A}}(a),(\mathbf{c},\mu)\rightharpoonup^{*}_{s}(\mathtt{skip},\mu')\}\subseteq\gamma_{\mathbb{A}}([\![\mathbf{c}]\!]^{\sharp}_{\mathbb{A}}(a))$

Reduced product of symbolic precise stores and abstract states. Reduced product [15] aims at expressing precisely conjunctions of constraints expressed in distinct abstract domains. We let a precise product store be a pair $(\kappa,a)\in\mathbb{K}\times\mathbb{A}$. In our case, the definition needs to be adapted slightly as symbolic execution and abstract domain $\mathbb{A}$ do not abstract exactly the same objects:

Definition 4 (Product domain). *The* product abstract domain *consists of the set* $\mathbb{K}\times\mathbb{A}$ *and the concretization function* $\gamma_{\mathbb{K}\times\mathbb{A}}:\mathbb{K}\times\mathbb{A}\longrightarrow\mathcal{P}(\mathbb{M}\times(\overline{\mathbb{V}}\rightarrow\mathbb{V}))$ *defined as follows:* $\gamma_{\mathbb{K}\times\mathbb{A}}:(\kappa,a)\longmapsto\{(\mu,\nu)\in\gamma_{\overline{\mathbb{M}}}(\kappa)\mid\mu\in\gamma_{\mathbb{A}}(a)\}$

In a precise product store (κ,a), the goal is to enhance precision by exchanging information between κ and a. This is done through a **reduction** function, which rewrites an abstract element with another of equal concretization, but that supports more precise analysis operations. This implies that $(\gamma_{\mathbb{K}\times\mathbb{A}} \circ \mathfrak{reduction})(\kappa,a) = \gamma_{\mathbb{K}\times\mathbb{A}}(\kappa,a)$. This requires the abstract domain $\mathbb{A}$ to support a function $\mathfrak{constr}$ that maps an abstract state a to a logical formula over program variables and entailed by a, namely such that, if $\mu \in \gamma_{\mathbb{A}}(a)$ then μ satisfies formula $\mathfrak{constr}(a)$. Some abstract domains—specifically intervals and abstract polyhedra—utilize an internal representation based on conjunction of constraints, in which case $\mathfrak{constr}$ is trivial. Then, $\mathfrak{reduction} : \mathbb{K}\times\mathbb{A} \longrightarrow \mathbb{K}\times\mathbb{A}$ is defined by:

$$\mathfrak{reduction}((\rho,\pi),a) \triangleq ((\rho,\pi'),a) \quad \text{where} \quad \pi' \triangleq \pi \wedge \mathfrak{constr}(a)[\vec{\mathtt{x}} \mapsto \rho(\vec{\mathtt{x}})]$$

Note that $[\vec{\mathtt{x}} \mapsto \rho(\vec{\mathtt{x}})]$ in the above definition, symbolizes the replacement of each program variable present in $\mathfrak{constr}(a)$ into its definition in ρ; this step follows from the fact that a constrains program variables whereas π constrains valuations. This general reduction function may be refined into a more precise one, where the resulting symbolic path is simplified, possibly to the **ff** formula. Furthermore, this reduction only modifies the symbolic path π, but it is possible to define a reduction operation that also rewrites the abstract state a.

Reduced product symbolic execution. The product analysis, namely RedSoundSE, takes the form of an extension of the symbolic execution function of Figure 4. The new states are still 4-tuples, but the symbolic precise store component κ is now replaced with a precise product store (κ,a). The transition relation $\rightharpoonup_{s\times\mathbb{A}}$ between such states consists of two rules that are shown in Figure 5(b) and that extend those in Figure 4. In rule S-A-MANY (applied when exploration bound is met) aside from $\mathfrak{modif}$, the loop is calculated over the abstract state and then the reduction function is applied.

In both cases, the sound $\mathfrak{reduction}$ operator may be applied. In practice, for the sake of efficiency, it can be computed and applied in a lazy manner that is, only for specific steps (typically S-A-MANY and for branching commands).

Example 5 (Product analysis). For program 2(d), assuming $\mathtt{i} < 10$, and then when exiting the loop, an intervals abstract state will hold two constraints $a = \{\mathtt{i} = 10; \mathtt{priv} \geq 2\}$. Assuming a symbolic precise store $\kappa = (\rho,\pi)$ with $\rho = [\mathtt{i} \to \boldsymbol{i}\,;\ \mathtt{priv} \to \boldsymbol{priv}]$, the abstract constraints can be fitted to a symbolic path π' as follows: $\pi' \triangleq \pi \wedge \boldsymbol{i} = 10 \wedge \boldsymbol{priv} \geq 2$. A more detailed execution trace is given in Appendix A.

Soundness and refutation property. The RedSoundSE analysis defined in the previous paragraph satisfies the same soundness (Theorem 2) and refutation (Theorem 3) properties as standard symbolic execution, so we do not give the theorems again.

6 SoundRSE: Sound relational symbolic execution

As discussed in Section 3, security properties like noninterference require to reason over *pairs* of execution traces thus we now set up a *sound relational symbolic execution* technique that constructs pairs of executions. This analysis will be regarded as SoundRSE.

Assumption. To keep notations lighter, we assume in this section and the next that the bounding counter step function $\mathfrak{step}$ only affects loops, namely $\mathfrak{step}(\mathbf{c},\mathbf{c}',w)=w$ whenever $\mathbf{c}$ is not a loop command. Moreover, we do not include the product with the numerical abstract state (as in Section 5) in the following definitions. Since it can be added in a seamless manner, we omit it here to keep formal statements lighter.

Precise relational stores. We first define the notions of relational expression, relational store, and precise relational store.

Definition 5 (Relational and precise relational stores). *A* relational symbolic expression *is an element defined by the grammar: $\tilde{\varepsilon} ::= \langle\varepsilon\rangle | \langle\varepsilon \mid \varepsilon\rangle$ where ε ranges over the set $\mathbb{E}$ of symbolic expressions. We write $\mathbb{E}_2$ for the set of relational symbolic expressions. A* relational symbolic store *$\tilde{\rho}$ is a function from variables to relational symbolic expressions. We let $\overline{\mathbb{M}}_2 = \mathbb{X} \rightarrow \mathbb{E}_2$ stand for their set. Finally, a* precise relational store *$\tilde{\kappa}$ is a pair $(\tilde{\rho},\pi) \in \mathbb{K}_2$.*

Before we define concretizations of $\overline{\mathbb{M}}_2$ and $\mathbb{K}_2$, we need to introduce two operations:

– The *projections* Π_0,Π_1 map relational symbolic stores into symbolic stores. They are defined in a pointwise manner, as follows: if $\tilde{\rho}(\mathtt{x})=\langle\varepsilon\rangle$ then $\Pi_0(\tilde{\rho})(\mathtt{x})=\Pi_1(\tilde{\rho})(\mathtt{x})=\varepsilon$ and if $\tilde{\rho}(\mathtt{x})=\langle\varepsilon_0 \mid \varepsilon_1\rangle$, then $\Pi_0(\tilde{\rho})(\mathtt{x})=\varepsilon_0$ and $\Pi_1(\tilde{\rho})(\mathtt{x})=\varepsilon_1$. We overload the Π_0,Π_1 notation and also apply it to double symbolic expressions: $\Pi_0(\langle\varepsilon\rangle)=\Pi_1(\langle\varepsilon\rangle)=\varepsilon$ and if $\tilde{\varepsilon}=\langle\varepsilon_0 \mid \varepsilon_1\rangle$, then $\Pi_0(\tilde{\varepsilon})=\varepsilon_0$ and $\Pi_1(\tilde{\varepsilon})=\varepsilon_1$.

– The *pairing* $(\!|\rho_0 \mid \rho_1|\!)$ of two symbolic stores ρ_0 and ρ_1 is a relational symbolic store defined such that, for all variable $\mathtt{x}$,

$$(\!|\rho_0 \mid \rho_1|\!)(\mathtt{x}) = \begin{cases} \langle\varepsilon\rangle & \text{if } \rho_0(\mathtt{x}) \text{ and } \rho_1(\mathtt{x}) \text{ are provably equal to } \varepsilon \in \mathbb{E} \\ \langle\rho_0(\mathtt{x}) \mid \rho_1(\mathtt{x})\rangle & \text{otherwise} \end{cases}$$

where the notion of "provably equal" may boil down to syntactic equality of symbolic expressions or involve an external proving tool.
We can now define the concretization functions:

Definition 6 (Concretization functions). *The* concretization of relational stores $\gamma_{\overline{\mathbb{M}}_2}$ *and* concretization of precise relational stores $\gamma_{\mathbb{K}_2}$ *are defined by:*

$$\begin{array}{rll} \gamma_{\overline{\mathbb{M}}_2} : \overline{\mathbb{M}}_2 & \longrightarrow & \mathcal{P}(\mathbb{M} \times \mathbb{M} \times (\overline{\mathbb{V}} \rightarrow \mathbb{V})) \\ \tilde{\rho} & \longmapsto & \{(\mu_0,\mu_1,\nu) \mid \forall \mathtt{x} \in \mathbb{X}, \forall i \in \{0,1\}, \mu_i(\mathtt{x}) = [\![\Pi_i(\tilde{\rho})(\mathtt{x})]\!](\nu)\} \\ \gamma_{\mathbb{K}_2} : \mathbb{K}_2 & \longrightarrow & \mathcal{P}(\mathbb{M} \times \mathbb{M} \times (\overline{\mathbb{V}} \rightarrow \mathbb{V})) \\ (\tilde{\rho},\pi) & \longmapsto & \{(\mu_0,\mu_1,\nu) \in \gamma_{\overline{\mathbb{M}}_2}(\tilde{\rho}) \mid [\![\pi]\!](\nu) = \boldsymbol{tt}\}. \end{array}$$

Example 6. We consider program 2(a) ($\mathsf{SoundSE}$ was discussed in Example 1). To cover pairs of executions that start with the same value for low variable $\mathtt{y}$ but possibly distinct values for high variable $\mathtt{priv}$, relational symbolic execution should cover four pairs of paths. These four paths have the same relational symbolic store $[\mathtt{priv} \rightsquigarrow \langle \boldsymbol{\mathtt{priv}}_0 \mid \boldsymbol{\mathtt{priv}}_1\rangle, \mathtt{y} \rightsquigarrow \langle 5\rangle]$ and differ only in the symbolic path components. For instance, when the first execution takes the true branch of the condition and the second the false branch, the symbolic path is $\boldsymbol{\mathtt{priv}}_0 > 0 \wedge \boldsymbol{\mathtt{priv}}_1 \leq 0$.

$$\text{SR-EXIT}\ \frac{}{(\texttt{skip}\bowtie\texttt{skip},(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\texttt{skip},(\tilde{\rho},\pi),w,b)}$$

$$\text{SR-COMP-R}\ \frac{(\mathbf{c}_1,(\mathit{\Pi}_1(\tilde{\rho}),\pi),w,b)\rightharpoonup_{s}(\mathbf{c}_1',(\rho_1',\pi'),w',b')}{(\texttt{skip}\bowtie\mathbf{c}_1,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\texttt{skip}\bowtie\mathbf{c}_1',(\llparenthesis\mathit{\Pi}_0(\tilde{\rho})\,|\,\rho_1'\rrparenthesis,\pi'),w',b')}$$

$$\text{SR-COMP-L}\ \frac{(\mathbf{c}_0,(\mathit{\Pi}_0(\tilde{\rho}),\pi),w,b)\rightharpoonup_{s}(\mathbf{c}_0',(\rho_0',\pi'),w',b')}{(\mathbf{c}_0\bowtie\mathbf{c}_1,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathbf{c}_0'\bowtie\mathbf{c}_1,(\llparenthesis\rho_0'\,|\,\mathit{\Pi}_1(\tilde{\rho})\rrparenthesis,\pi'),w',b')}$$

$$\text{SR-IF-TF}\ \frac{(\mathbf{b},\tilde{\rho})\vdash_{\mathrm{sr}}\tilde{\beta}\qquad \pi'=\pi\wedge\mathit{\Pi}_0(\tilde{\beta})\wedge\neg\mathit{\Pi}_1(\tilde{\beta})\qquad \mathbf{may}(\pi')}{(\texttt{if}\ \mathbf{b}\ \texttt{then}\ \mathbf{c}_0\ \texttt{else}\ \mathbf{c}_1,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathbf{c}_0\bowtie\mathbf{c}_1,(\tilde{\rho},\pi'),w,b)}$$

$$\text{SR-APPROX-MANY}\ \frac{\begin{array}{c}\mathfrak{step}(\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c},(\mathbf{c};\ \texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}),w)=(\mathbf{ff},w')\\ \tilde{\rho}''=\mathfrak{modif}(\tilde{\rho},\mathbf{c})\qquad(\mathbf{b},\tilde{\rho}'')\vdash_{\mathrm{sr}}\langle\beta_0,\beta_1\rangle\qquad\pi'\triangleq\pi\wedge\neg\beta_0\wedge\neg\beta_1\end{array}}{(\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c},(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\texttt{skip},(\tilde{\rho}'',\pi'),w',\mathbf{ff})}$$

Fig. 6. SoundRSE: a few selected rules of the relational symbolic execution step relation.

Relational symbolic execution algorithm. Since SoundRSE aims at describing pairs of executions, it should account for the case where the two executions follow different control flow paths. Thus, a relational symbolic state may consist of a single command when both executions follow the same path, or two commands when they diverge. We respectively note these two kinds of states $(\mathbf{c},\tilde{\kappa},w,b)$ and $((\mathbf{c}_0\bowtie\mathbf{c}_1);\ \mathbf{c}_2,\tilde{\kappa},w,b)$; in the latter, $\mathbf{c}_0$ (resp., $\mathbf{c}_1$) denotes the control state of the first (resp., second) execution, which they later meet in $\mathbf{c}_2$. The components w and b have the same meaning as in Section 4. Initial states are of the former sort.

We write $\rightharpoonup_{sr}$ for the relational symbolic execution step relation. A representative selection of the rules are shown in Figure 6. Rule SR-APPROX-MANY describes a case where approximation is performed so as to ensure termination and uses the straightforward extension of $\mathfrak{modif}$ to relational symbolic states.

Soundness and refutation property. SoundRSE inherits similar soundness and refutation properties as SoundSE, as shown in the following theorems.

Theorem 4 (Soundness). *Let $\tilde{\kappa}\in\mathbb{K}_2$, $w\in\mathbb{W}$, and $b\in\mathbb{B}$. We let $(\mu_0,\mu_1,\nu)\in\gamma_{\mathbb{K}_2}(\tilde{\kappa})$ and assume that stores μ_0',μ_1' are such that $(\boldsymbol{c},\mu_0)\rightarrow^*(\texttt{skip},\mu_0')$ and $(\boldsymbol{c},\mu_1)\rightarrow^*(\texttt{skip},\mu_1')$. Then, there exists $\tilde{\kappa}'\in\mathbb{K}_2$, a valuation ν', and a counter state $w'\in\mathbb{W}$ such that $\nu\preceq\nu'$, $(\mu_0',\mu_1',\nu')\in\gamma_{\mathbb{K}}(\tilde{\kappa}')$, and $(\boldsymbol{c},\tilde{\kappa},w,b)\rightharpoonup_{sr}^*(\texttt{skip},\tilde{\kappa}',w',b')$.*

Theorem 5 (Refutation up to a bound). *Let $\boldsymbol{c}$ be a command, $\tilde{\kappa},\tilde{\kappa}'\in\mathbb{K}_2$ be two precise stores, $w,w'\in\mathbb{W}$, such that $(\boldsymbol{c},\kappa,w,\boldsymbol{tt})\rightharpoonup_{sr}^*(\texttt{skip},\kappa',w',\boldsymbol{tt})$. Then, for all $(\mu_0',\mu_1',\nu')\in\gamma_{\mathbb{K}_2}(\kappa')$, it exists $(\mu_0,\mu_1,\nu)\in\gamma_{\mathbb{K}_2}(\kappa)$ such that $(\boldsymbol{c},\mu_0)\rightarrow^*(\texttt{skip},\mu_0')$ and $(\boldsymbol{c},\mu_1)\rightarrow^*(\texttt{skip},\mu_1')$.*

SoundRSE-based analysis and noninterference. We now assume a program $(\mathbf{c},L)$, and show the application of SoundRSE analysis to attempt proving noninterference. The analysis proceeds according to the following steps:

1. Construction of the initial store $\tilde{\rho}_0$ such that, for all variables $\mathtt{x}$ present in $\mathbf{c}$, $\tilde{\rho}_0(\mathtt{x})=\langle\boldsymbol{x}\rangle$ (resp., $\tilde{\rho}_0(\mathtt{x})=\langle\boldsymbol{x}_0\,|\,\boldsymbol{x}_1\rangle$) if $\mathtt{x}\in L$ (resp., $\mathtt{x}\notin L$), and where $\boldsymbol{x}$ is a fresh symbolic value (resp., $\boldsymbol{x}_0,\boldsymbol{x}_1$ are fresh symbolic values).

2. Exhaustive application of semantic rules from initial state $(\mathbf{c},(\tilde{\rho}_0,\mathbf{tt}),w_0,\mathbf{tt})$; we let $\mathcal{O}$ stand for the set of final precise relational stores with their precision flags: $\mathcal{O} \triangleq \{(\tilde{\kappa},b) \mid \exists w \in \mathbb{W}, (\mathbf{c},(\tilde{\rho}_0,\mathbf{tt}),w_0,\mathbf{tt}) \rightharpoonup_{sr} (\mathtt{skip},\tilde{\kappa},w,b)\}$.
3. *Attempt to prove noninterference* for each symbolic path in $\mathcal{O}$ using an external tool, such as an SMT solver; more precisely, given $((\tilde{\rho},\pi),b) \in \mathcal{O}$,
   - if π is not satisfiable, the path is infeasible and can be ignored;
   - if it can be proved that for all variables $\mathtt{x} \in L$, there is a unique value, i.e., $\Pi_0(\tilde{\rho})(\mathtt{x}) = \Pi_1(\tilde{\rho})(\mathtt{x})$, then the program is noninterferent;
   - if a valuation ν can be found, such that $[\![\pi]\!](\nu) = \mathbf{tt}$ (the path is satisfiable), and there exists a variable $\mathtt{x} \in L$ such that $[\![\Pi_0(\tilde{\rho})(\mathtt{x})]\!](\nu) \neq [\![\Pi_1(\tilde{\rho})(\mathtt{x})]\!](\nu)$, and $b = \mathbf{tt}$, then ν provides a counter-example refuting noninterference;
   - finally, if $b = \mathbf{ff}$ and neither of the above cases occurs, no conclusive answer can be given for this path.

To summarize, the analyser either proves noninterference (when all paths are either not satisfiable or noninterferent), or it provides a valuation that refutes noninterference (when such a valuation can be found for at least one path), or it does not conclude. When a refutation is found, this refutation actually defines a real attack.

Example 7 (Noninterference). In the case of program 2(a), all paths are low-equal. The analysis of program 2(c) computes at least one interferent path if the unrolling bound is set to any strictly positive integer; in that case, a model such as the one presented in Section 3 can be synthesized by even basic SMT solvers. Finally, the program of Figure 2(d) can be proved noninterferent with relational symbolic execution combined with a reduced product with a value abstract domain such as intervals (Section 5).

7 RedSoundRSE: Product of SoundRSE with Dependence AI

As observed in Section 3 some programs like that of Figure 2(b) can be analyzed more precisely using conventional dependence analysis than by bounded symbolic execution (Section 4). In this section, we set up a novel form of product of abstractions, so as to benefit from this increase in precision. This notion of product is generic and does not require to fix a specific dependency abstraction. We refer to the final analysis presented in this section as RedSoundRSE.

Dependence abstraction and static analysis. Although dependence abstractions may take many forms, they all characterize information flows that can be observed by comparing pairs of executions. For instance, [4] uses a lattice of security levels and abstract elements map each level to a set of variables. These variables are left unmodified when the input value of variables of higher levels change. Other works use relational abstract domains, where relational means that relations are maintained *across pairs of executions*. Therefore, we can characterize such analyses with an abstraction of pairs of stores:

Definition 7 (Dependence abstraction and analysis). *A* dependence abstraction *is defined by an abstract lattice $\mathbb{D}$ from security levels to variables and a con-*

cretization function

$$\begin{aligned}\gamma_{\mathbb{D}}:\ &\mathbb{D} \longrightarrow \mathcal{P}(\mathbb{M}\times\mathbb{M})\\ &d \longmapsto \{(\mu_0,\mu_1)\in\mathbb{M}\times\mathbb{M} \mid \mu_0 =_{d(\mathbb{L})} \mu_1\}\end{aligned}$$

A sound dependency analysis *is defined by a function* $[\![\boldsymbol{c}]\!]^{\sharp}_{\mathbb{D}}:\mathbb{D}\rightarrow\mathbb{D}$ *such that, for all* $d\in\mathbb{D}$*,* $(\mu_0,\mu_1)\in\gamma_{\mathbb{D}}(d)$*,* $\{(\mu'_0,\mu'_1)\in\mathbb{M}\times\mathbb{M} \mid \forall i\in\{0,1\},(\boldsymbol{c},\mu_i)\rightarrow(\texttt{skip},\mu'_i)\}\subseteq\gamma_{\mathbb{D}}\circ[\![\boldsymbol{c}]\!]^{\sharp}_{\mathbb{D}}(d)$*.*

Example 8 (Standard dependence based abstraction [4]). The abstraction of [4] is an instance of Definition 7. Let $\{\mathbb{L},\mathbb{H}\}$ be the set of security levels. Assume an initial abstract state d that captures pairs of concrete stores that are low equal for some program $(\mathbf{c},L)$. By applying the dependence analysis, if the final dependence state has a low dependency for each initially low variable, the program is noninterferent.

In practice such information is computed by forward abstract interpretation, using syntactic dependencies for expressions and conditions, and conservatively assuming conditions may generate (implicit) flows to any operation that they guard.

We note that Definition 7 accounts not only for dependence abstractions such as that of [4]. In particular, [22] proposes a semantic patch analysis which can also be applied to security properties by using a relational abstract domain to relate pairs of executions; such analyses use an abstraction that also writes as in Definition 7. In the following, we assume a sound dependence analysis is fixed.

Product of symbolic execution and dependence analysis. We now combine dependence analysis and symbolic execution. For most statements, SoundRSE rules defined in Figure 6 introduce no imprecision. The notable exception is the case where the execution bound is reached as in rule SR-APPROX-MANY. Therefore, the principle of the combined analysis is to replace this imprecise rule with another that uses dependence analysis results to strengthen relational stores. First, we introduce two operations to transport information in a sound manner into and from the dependence abstract domain:

Definition 8 (Information translation and dependence abstraction). *The* translation from symbolic to dependence *is a function* $\tau_{s\rightarrow\mathbb{D}}:\overline{\mathbb{M}}_2\rightarrow\mathbb{D}$ *that is sound in the following sense:* $\forall\tilde{\rho}\in\overline{\mathbb{M}}_2$, $\forall(\mu_0,\mu_1,\nu)\in\gamma_{\overline{\mathbb{M}}_2}(\tilde{\rho})$, $(\mu_0,\mu_1)\in\gamma_{\mathbb{D}}\circ\tau_{s\rightarrow\mathbb{D}}(\tilde{\rho})$*. The* extraction of dependence information *is a function* $\lambda_{\mathbb{D}\rightarrow\mathbb{L}}:\mathbb{D}\rightarrow\mathcal{P}(\mathbb{X})$ *that is sound in the following sense:* $\forall d\in\mathbb{D},\forall(\mu_0,\mu_1)\in\gamma_{\mathbb{D}}(d),\mu_0 =_{\lambda_{\mathbb{D}\rightarrow\mathbb{L}}(d)} \mu_1$

Intuitively, $\tau_{s\rightarrow\mathbb{D}}$ should compute a dependence abstract domain element that expresses a property implied by the relational symbolic store it is applied to. In the set-up of Example 8, a straightforward way to achieve that is to map $\tilde{\rho}$ to an element d that maps $\mathbb{L}$ to the set: $\{\mathtt{x}\in\mathbb{X} \mid \mathbf{may}(\mathit{\Pi}_0(\tilde{\rho})(\mathtt{x})=\mathit{\Pi}_1(\tilde{\rho})(\mathtt{x}))\}$

When $\tilde{\rho}(\mathtt{x})=\langle\varepsilon\rangle$, this equality is clearly satisfied; when $\tilde{\rho}(\mathtt{x})=\langle\varepsilon_0\mid\varepsilon_1\rangle$, the equality $\varepsilon_0=\varepsilon_1$ needs to be discharged by an external tool such as an SMT solver. Similarly, the function $\lambda_{\mathbb{D}\rightarrow\mathbb{L}}$ extracts a set of variables which are proved to remain low by the its argument. In the setup of Example 8, this boils down to returning $d(\mathbb{L})$.

We now present the combined analysis. The symbolic execution step SR-APPROX-MANY-DEP is shown in Figure 7 and replaces rule SR-APPROX-MANY (Figure 6). When

$$\textsc{sr-approx-many-dep}\ \frac{\begin{array}{c}\mathfrak{step}(\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c},(\mathbf{c};\ \texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}),w)=(\mathbf{ff},w') \\ d=[\![\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}]\!]^{\sharp}_{\mathbb{D}}(\tau_{s\to\mathbb{D}}(\tilde{\rho})) \qquad \tilde{\rho}''=\mathfrak{modif}_{\mathbb{D}}(\tilde{\rho},\mathbf{c},\lambda_{\mathbb{D}\to\mathbb{L}}(d)) \\ (\mathbf{b},\tilde{\rho}'')\vdash_{\mathrm{sr}}\langle\beta_0,\beta_1\rangle \qquad \pi'\triangleq\pi\wedge\neg\beta_0\wedge\neg\beta_1\end{array}}{(\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c},(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr\times\mathbb{D}}(\texttt{skip},(\tilde{\rho}'',\pi'),w',\mathbf{ff})}$$

Fig. 7. RedSoundRSE: Symbolic execution approximation and product with dependence information.

the execution bound is reached for a loop statement, it performs the dependence analysis of the whole loop from the dependence state derived by applying $\tau_{s\to\mathbb{D}}$ to the relational symbolic store. Then, it applies $\lambda_{\mathbb{D}\to\mathbb{L}}$ to derive the set of variables that are proved to be low by the dependence analysis. Finally, it computes a new relational symbolic store by modifying the variables according to the set of variables determined low:

- if variable $\mathtt{x}$ is low based on the $\lambda_{\mathbb{D}\to\mathbb{L}}$ output, $\mathfrak{modif}_{\mathbb{D}}$ synthesizes one fresh symbolic value $\mathfrak{x}_{\text{new}}$ and maps it to $\langle\mathfrak{x}_{\text{new}}\rangle$;
- if variable $\mathtt{x}$ cannot be proved low, $\mathfrak{modif}_{\mathbb{D}}$ synthesizes two fresh symbolic values $\mathfrak{x}_{\text{new0}}$, $\mathfrak{x}_{\text{new1}}$ and maps $\mathtt{x}$ to $\langle\mathfrak{x}_{\text{new0}}\,|\,\mathfrak{x}_{\text{new1}}\rangle$.

Remark 1 (Reduced product property). We stress the fact that the rule SR-APPROX-MANY-DEP may be applied multiple times during the analysis, essentially whenever a loop statement is analyzed, which is generally many times more than the number of loop commands in the program due to abstract iterations. Therefore, our analysis *cannot* be viewed as a fixed sequence of analyses. Such a decomposition (e.g., where dependence analysis is ran first and SE second) would be strictly less precise than our reduced product based approach.

Soundness and refutation properties. Under the assumption that the dependence analysis and translation operations are sound, so is the combined symbolic execution, thus Theorem 4 still holds. Moreover, the refutation property of Theorem 5 also holds.

Example 9 (Combined analysis). We consider program 2(b). As discussed in Section 3, the loop statement may execute unboundedly many times, thus relational symbolic execution applies rule SR-APPROX-MANY-DEP. The initial dependence abstract element computed for the loop by $\tau_{s\to\mathbb{D}}$ maps $\mathbb{L}$ to $\{\mathtt{i},\mathtt{z}\}$ and $\mathbb{H}$ to all variables. The dependence analysis of the loop returns the same element. Thus, the set of low variables returned by $\lambda_{\mathbb{D}\to\mathbb{L}}$ is $\{\mathtt{i},\mathtt{z}\}$, which allows to compute a precise relational symbolic store and to successfully verify the program is noninterferent.

8 Comparison

In this section we compare our analyses among them as well as with the dependency analysis of Assaf et al. [4]. To do so, we implemented prototypes of all the analyses. Our goal is not to evaluate the analyses in large code bases but to assess their differences based on programs that are small but challenging for typical noninteference analysers.

```
if (priv > 0)
  i = 0;
else
  i = 0;
while (i < 10) {
  i += 1;
  priv += 5;
}
```

(a) Secure

```
i = 0; w = 2;
x = 100;
while(i < x) {
  if (x <= 0)
    w = priv;
  i += 2;
  x += 1;
}
```

(b) Secure

```
i = 0;
while (i < 3) {
    y0 = y1;
    y1 = y2;
    y2 = priv;
    i += 1;
}
y1 = 0; y2 = 0;
```

(c) Insecure

```
i = 0;
while (i < 100) {
    if (priv > 0)
        y = 5;
    i += 1;
}
```

(d) Insecure

Fig. 8. Programs illustrating different properties of the analyzer. Variable `priv` is high.

Relational Analysis		$\mathbb{D}$	SoundRSE		RedSoundRSE ($\mathbb{D}$)	
relational analysis input:		None	SoundSE	RedSoundSE	SoundSE	RedSoundSE
Program	**Secure?**					
Fig. 2(a)	Yes	✗ False alarm	✓ Secure	✓ Secure **(I,P)**	✓ Secure	✓ Secure **(I,P)**
Fig. 2(b)	Yes	✓ Secure	✗ False alarm	✓ Secure **(P)**	✓ Secure	✓ Secure **(I,P)**
Fig. 2(d)	Yes	✗ False alarm	✗ False alarm	✓ Secure **(I,P)**	✗ False alarm	✓ Secure **(I,P)**
Fig. 8(a)	Yes	✗ False alarm	✗ False alarm	✗ False alarm	✓ Secure	✓ Secure **(I,P)**
Fig. 8(b)	Yes	✗ False alarm	✗ False alarm	✗ False alarm	✗ False alarm	✓ Secure **(I,P)**
Fig. 2(c)	No	✓ Alarm	✓ Refutation model	✓ Refutation model	✓ Refutation model	✓ Refutation model
Fig. 8(c)	No	✓ Alarm	✓ Refutation model	✓ Refutation model	✓ Refutation model	✓ Refutation model
Fig. 8(d)	No	✓ Alarm	✓ Alarm	✓ Alarm	✓ Alarm	✓ Alarm

Table 2. Evaluation and comparison of analyses combination. $\mathbb{D}$ denotes the dependency analysis of [4]. Symbol ✓ (resp., ✗) denotes a semantically correct (resp., incorrect) analysis outcome, with either a proof of security, a (possibly false) alarm, or a refutation model. For RedSoundSE columns, when the analyses succeed to prove NI, we mark the result with **I** (resp. **P**) to indicate that the intervals (resp. polyhedra) domain is being used.

Implementation. We prototype the analyses proposed in this work as well as the dependency analysis, intervals and convex polyhedra analysis. The prototype is implemented in around 4k lines of OCaml code, using the Apron library [29] for the numerical domains and the Z3 SMT solver [21]. By defining a shared interface for SoundSE and RedSoundSE, the implementation of RedSoundRSE is parameterized by these. An artifact of the implementation has been provided.

Evaluation. We compare the 3 different relational techniques using different single-trace analyses by evaluating them on a set of challenging examples. Our results are shown in Table 2. In the following, we split NI programs from non NI ones. For the latter we look at the refutation capabilities of the analysis.

Comparison of the verification capabilities of different relational analyses. Programs of Fig. 2 were already explained in Section 3 and our prototype confirmed these results, which are summarized in Table 2.

In Program 8(a), the first condition renders dependence analysis useless as it will consider variable `i` high. This program will also fail to be verified by SoundRSE if the iteration bound is lower than 10: in this case, `i` will be assigned a fresh symbolic value and hence be deemed high. In contrast, RedSoundRSE can determine that the value of `i` in the loop does not depend on `priv`.

Program 8(b) is more convoluted. The analysis requires both numerical and dependence abstractions in order to prove its NI. The analysis will determine (conservatively) that three variables are modified in the loop: `x`, `i` and `w`. Dependence analysis can determine that variable `i` and `x` are low even if both are modified. However, since `w` depends on `x`, and the exact value of `x` is unknown, it is not possible to determine that `w` is low. By adding a numerical domain, it is easy to track that the value of `x` is always positive, which implies that the `if` statement can never be executed.

Comparison of the refutation capabilities of different relational analyses. Since SoundRSE and RedSoundRSE unroll loops a bounded number of times, there are insecure programs for which a refutation model can be found, and programs where this is not possible. Notice that, to refute a program with a model, it is required that the symbolic execution did not perform any over approximation, i.e. that the precision flag is set to false when the analysis finds the violation. Therefore, the results for insecure programs of SoundRSE are similar to those of the different combinations that rely on symbolic execution, as reflected on Figure 2. For Program 2(c), a valuation can be found by doing one iteration: $\nu(\boldsymbol{i}_0)=\nu(\boldsymbol{i}_1)=1$ and $\nu(\boldsymbol{priv}_0)=0$, $\nu(\boldsymbol{priv}_1)=1$. For Program 8(c), a model can be found if the bound of iterations is set to 4 or higher. The valuation ν just needs to map variable `priv` to two different values: $\nu(\boldsymbol{priv}_0)\neq\nu(\boldsymbol{priv}_1)$. In Program 8(d), for any user-set bound lower than 100 the execution will have to overapproximate, losing refutation capabilities.

Conclusion of the evaluation. We have evaluated and compared our analyses among them and with the state-of-the-art on dependency analyses [4] on a set of 8 challenging examples. Our results show that, in contrast to dependencies [4], analyses inherit the capacity of providing a refutation model up to a bound from symbolic execution. Moreover, RedSoundRSE instantiated with RedSoundSE is capable of soundly verifying all the examples, in contrast to all the other compared analyses, as summarized in Table 2.

Limitations. As RedSoundSE is sound and automatic, it necessarily fails to achieve completeness (by Rice's Theorem [30, 3]). In return, we provide completeness up to a bound. Another more subtle limitation is that the numerical abstraction are applied at the level of the single symbolic execution (RedSoundSE). This means that these abstractions cannot track down relations between executions, but just local constraints.

9 Related work

Hyperproperties Noninterference was first defined by Goguen and Meseguer [26], and also generalized to more powerful attacker models under the property name of declassification. We refer the reader to a survey on declassification policies [37] up to 2005. As discussed in the introduction, noninterference is not a safety property but a safety hyperproperty [13], a.k.a. hypersafety. Several works in the literature have shown that hypersafety verification can be reduced to verification of safety properties [7, 20, 39, 13], however this reduction is not always efficient in practice [39]. In our work, we do not reduce noninterference to verification of safety but rather apply relational analyses. We only show our results using noninterference but the methodology can be easily generalized to more relaxed declassification properties, provided sound abstract domains exist.

Symbolic execution. SE is a static analysis technique that was born in the 70s [9, 31] and that is now deployed in several popular testing tools, such as KLEE [11] and NASA's Symbolic PathFinder [35], to name a few. A primary goal and strength of SE is to find paths leading to counter-examples to generate concrete input values exercising that path. This is of particular importance to security in order to debug and confirm the feasibility of an attack when a vulnerability is detected.

Alatawi et al. [2] use AI to enhance the precision of a dynamic symbolic execution aimed at path coverage. Their approach consists of first doing an analysis of the program with AI to capture indirect dependences in order to enhance path predicates. Furthermore, their analysis does not maintain soundness (nor completeness). Meanwhile, our approach continuously alternates between abstract domains and symbolic execution, keeping soundness and completeness up to a bound. Lastly, Alatawi et al. [2] do not analyze relational properties such as noninterference but just safety properties.

We focus the rest of the related work on static analysis techniques for relational security properties: for a broader discussion on symbolic execution we refer the interested reader to a survey [10] up to 2011 and an illuminating discussion on SE challenges in practice up to 2013 [12].

Relational symbolic execution. In order to apply SE to security properties such as noninterference, Milushev et al. [32] propose a form of relational symbolic execution (RSE) to use KLEE to analyze noninterference by means of a technique called self-composition [7, 20, 39] to reduce a relational property of a program p to a safety property of a transformation of p. More recently, Daniel et al. have optimized RSE to be applicable to binary code to analyze relational properties such as constant time [17] and speculative constant time [18, 19] and discovered violations of these properties in real-world cryptographic libraries. All these approaches are based on pure (relational) SE static techniques and, as such, they are not capable of recovering soundness beyond a fixed bound as in our case. The closest work to RedSoundRSE is RelSym [23] which supports interactive refutation, as well as soundness. In order to recover soundness, Chong et al. [23] propose to use RelSym on manually annotated programs with loop invariants. Precision of refutation is guaranteed only if the invariants are strong enough, which cannot be determined by the tool itself. Precision is not guaranteed in any other cases. In contrast, our invariants are automatically generated via AI and precision of refutation is always guaranteed up to a bound, which is automatically computed by our tool.

Sound static analyses for hyperproperties As discussed in the introduction, many sound verification methods have been proposed for relational security properties. We refer the reader to an excellent survey on this topic [36] up to 2003. After 2003, several sound (semi-) static verification methods of noninterference-like properties have been proposed by means of type systems (e.g. [6, 24]), hybrid types, (e.g. [38]), relational logics (e.g. [1]), model checking (e.g. [27, 5]), and pure AI [4]. We expand on the ones based on AI since they are the closest to our work. Giacobazzi and Mastroeni [25] define abstractions for attacker's views of program secrets and design sound automatic program analyses based on AI for sets of executions (in contrast to relational executions). Assaf et al. [4] are the first to express hyperproperties entirely within the framework of AI by defining a Galois connection that directly approximates the hyperproperty of interest. We utilize the abstract domain of Assaf et al. [4] combined with SE to obtain RedSoundSE. Notice that because the framework of Assaf et al. [4] relies on incomplete abstraction, their analysis is not capable of precise refutation nor provide refutations models. To the best of our knowledge, no previous work has combined abstract domains and SE to achieve soundness.

10 Conclusion

In this work, we propose a series of analyses, summarized in Fig.1, combining SE and AI. Our analyses are sound, precise, and able to synthesize counter-examples up to a given bound. We prototype these analyses as well as several AI domains and a dependency analysis to verify noninterference. Our results, summarized in Table 2, show that on a set of challenging examples for noninterference, our analysis performs better than the dependency analysis and is able to preciselyblank and soundly conclude on whether programs are noninterferent or not and provide refutation models up to a bound. Given these encouraging results, we plan to generalize the target security property and make the analyses scale to other languages as future work.

Acknowledgements The authors thank the anonymous reviewers for their comments, helpful for improving the paper. This project was funded by INRIA Challenge SPAI and by the VeriAMOS ANR Project. This research was partially supported by the ANR17- CE25-0014-01 CISC project We would also like to thank Josselin Giet and Adam Khayam for their observations.

References

1. A. Aguirre, G. Barthe, M. Gaboardi, D. Garg, and P.-Y. Strub. A relational logic for higher-order programs. *Proc. ACM Program. Lang.*, 1(ICFP), aug 2017.
2. E. Alatawi, H. Søndergaard, and T. Miller. Leveraging abstract interpretation for efficient dynamic symbolic execution. In G. Rosu, M. D. Penta, and T. N. Nguyen, editors, *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 619–624. IEEE Computer Society, 2017.
3. A. Asperti and C. Armentano. A page in number theory. *J. Formaliz. Reason.*, 1(1):1–23, 2008.
4. M. Assaf, D. A. Naumann, J. Signoles, É. Totel, and F. Tronel. Hypercollecting semantics and its application to static analysis of information flow. In *Symposium on Principles of Programming Languages (POPL)*, pages 874–887. ACM, jan 2017.
5. M. Backes, B. Köpf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy (S&P 2009), 17-20 May 2009, Oakland, California, USA*, pages 141–153, 2009.

6. A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *2008 IEEE Symposium on Security and Privacy (S&P 2008), 18-21 May 2008, Oakland, California, USA*. IEEE Computer Society, 2008.
7. G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of the IEEE Computer Security Foundations Workshop (CSF)*, volume 17, pages 100–114, 2004.
8. N. Bielova and T. Rezk. A taxonomy of information flow monitors. In *Principles of Security and Trust - 5th International Conference, POST 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, Lecture Notes in Computer Science, pages 46–67. Springer, 2016.
9. R. S. Boyer, B. Elspas, and K. N. Levitt. SELECT - a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the International Conference on Reliable Software 1975, Los Angeles, California, USA, April 21-23, 1975*, pages 234–245. ACM, 1975.
10. C. Cadar, P. Godefroid, S. Khurshid, C. S. Pasareanu, K. Sen, N. Tillmann, and W. Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, pages 1066–1071. ACM, 2011.
11. C. Cadar and M. Nowack. KLEE symbolic execution engine in 2019. *International Journal of Software Tools Technol. Transf.*, 2021.
12. C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, pages 82–90, 2013.
13. M. R. Clarkson and F. B. Schneider. Hyperproperties. In *Proceedings of the IEEE Computer Security Foundations Symposium (CSF)*, pages 51–65. IEEE, 2008.
14. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.
15. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symposium on Principles of Programming Languages (POPL)*. ACM, 1979.
16. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Symposium on Principles of Programming Languages (POPL)*, pages 84–97. ACM, 1978.
17. L. Daniel, S. Bardin, and T. Rezk. Binsec/rel: Efficient relational symbolic execution for constant-time at binary-level. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1021–1038, 2020.
18. L. Daniel, S. Bardin, and T. Rezk. Hunting the haunter - efficient relational symbolic execution for spectre with haunted relse. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.
19. L. Daniel, S. Bardin, and T. Rezk. Reflections on the experimental evaluation of a binary-level symbolic analyzer for spectre. In *Post-proceedings of the LASER@NDSS 2021*. The Internet Society, 2022.
20. Á. Darvas, R. Hähnle, and D. Sands. A theorem proving approach to analysis of secure information flow. In D. Hutter and M. Ullmann, editors, *Security in Pervasive Computing, Second International Conference, SPC 2005, Boppard, Germany, April 6-8, 2005, Proceedings*, Lecture Notes in Computer Science, pages 193–209. Springer, 2005.
21. L. de Moura and N. Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

22. D. Delmas and A. Miné. Analysis of software patches using numerical abstract interpretation. In B. E. Chang, editor, *Static Analysis Symposium (SAS)*, volume 11822 of *LNCS*, pages 225–246. Springer, 2019.
23. G. P. Farina, S. Chong, and M. Gaboardi. Relational symbolic execution. In E. Komendantskaya, editor, *Proceedings of the 21st International Symposium on Principles and Practice of Programming Languages, PPDP 2019, Porto, Portugal, October 7-9, 2019*, pages 10:1–10:14. ACM, 2019.
24. C. Fournet, J. Planul, and T. Rezk. Information-flow types for homomorphic encryptions. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 351–360, 2011.
25. R. Giacobazzi and I. Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004*, pages 186–197. ACM, 2004.
26. J. A. Goguen and J. Meseguer. Security policies and security models. In *IEEE Symposium on Security and Privacy, Oakland*, pages 11–20. IEEE Computer Society, 1982.
27. M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *19th IEEE Computer Security Foundations Workshop (CSFW'06)*, 2006.
28. S. Hunt and D. Sands. On flow-sensitive security types. In *Symposium on Principles of Programming Languages (POPL)*, pages 79–90. ACM, 2006.
29. B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In A. Bouajjani and O. 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.
30. H. R. Jr. *Theory of recursive functions and effective computability (Reprint from 1967)*. MIT Press, 1987.
31. J. C. King. Symbolic execution and program testing. *Commununications of the ACM*, 19(7):385–394, 1976.
32. D. Milushev, W. Beck, and D. Clarke. Noninterference via symbolic execution. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE 2012, Stockholm, Sweden, June 13-16, 2012. Proceedings*, Lecture Notes in Computer Science, pages 152–168. Springer, 2012.
33. M. Ngo, N. Bielova, C. Flanagan, T. Rezk, A. Russo, and T. Schmitz. A better facet of dynamic information flow control. In P. Champin, F. Gandon, M. Lalmas, and P. G. Ipeirotis, editors, *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*, pages 731–739, 2018.
34. H. Palikareva, T. Kuchta, and C. Cadar. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pages 1181–1192. ACM, 2016.
35. C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In B. G. Ryder and A. Zeller, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2008, Seattle, WA, USA, July 20-24, 2008*, pages 15–26. ACM, 2008.
36. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal Sel. Areas Commun.*, 21(1):5–19, 2003.

37. A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *18th IEEE Computer Security Foundations Workshop, (CSFW-18 2005), 20-22 June 2005, Aix-en-Provence, France*, pages 255–269. IEEE Computer Society, 2005.
38. J. F. Santos, T. P. Jensen, T. Rezk, and A. Schmitt. Hybrid typing of secure information flow in a javascript-like language. In *Trustworthy Global Computing - 10th International Symposium, TGC 2015, Madrid, Spain, August 31 - September 1, 2015 Revised Selected Papers*, Lecture Notes in Computer Science, pages 63–78. Springer, 2015.
39. T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *Static Analysis, 12th International Symposium, SAS 2005, London, UK, September 7-9, 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005.

A Trace of program 2(d) with **RedSoundSE** using intervals

This section aims to show the execution of one symbolic trace of program 2(d). Initial precise store κ will capture the initial low-equality of variables `i` and `y`. The abstract state is a. Changes to the product store are marked in red.

$$\kappa = \begin{cases} \rho = [\mathtt{i} \to \langle i_0 \rangle, \mathtt{y} \to \langle y_0 \rangle, \mathtt{priv} \to \langle \boldsymbol{priv}_0 \rangle] \\ \pi = \mathbf{tt} \\ a_l = [\ ] \end{cases}$$

In line 3, since `priv` is unconstrained, the semantics can choose either path. Let us assume that our trace follows rule S-IF-T. Then, by line 5 the state is as follows.

$$\kappa = \begin{cases} \rho = [\mathtt{i} \to \langle i_0 \rangle, \mathtt{y} \to \langle y_0 \rangle, \mathtt{priv} \to \langle 0 \rangle] \\ \pi = \boldsymbol{priv}_0 < 0 \\ a = [\mathtt{priv} = 0] \end{cases}$$

Since this loop has an unbounded amount of iterations, we know that an over approximation will happen. Let us assume that the iteration bound is 1 (meaning that the semantics will execute the loop once at most before over approximating), and that $\mathtt{i} < 10$. By executing one full iteration the following symbolic state is reached.

$$\kappa = \begin{cases} \rho = [\mathtt{i} \to \langle i_0 + 1 \rangle, \mathtt{y} \to \langle y_0 \rangle, \mathtt{priv} \to \langle 2 \rangle] \\ \pi = i_0 < 10 \wedge \boldsymbol{priv}_0 < 0 \\ a = [\mathtt{priv} = 0; \mathtt{i} < 11] \end{cases}$$

Since now the limit of iterations is reached, next step is over approximating the loop. For the example we will next show the state just before the reduction. Notice that the new constraints in π are the result of negating the guard.

$$\kappa = \begin{cases} \rho = [\mathtt{i} \to \langle i_1 \rangle, \mathtt{y} \to \langle y_0 \rangle, \mathtt{priv} \to \langle \boldsymbol{priv}_1 \rangle] \\ \pi = i_1 \geq 10 \wedge i_0 < 10 \wedge \boldsymbol{priv}_0 < 0 \\ a = [\mathtt{priv} \geq 2; \mathtt{i} = 10] \end{cases}$$

Because variables `i` and `priv` were modified, new symbolic values are assigned. This generates a big inaccuracy, but abstract states can compensate. By reducing we add the constraints of a to π.

$$\kappa = \begin{cases} \rho = [\mathtt{i} \to \langle i_1 \rangle, \mathtt{y} \to \langle y_0 \rangle, \mathtt{priv} \to \langle \boldsymbol{priv}_1 \rangle] \\ \pi = \boldsymbol{priv}_1 \geq 2 \wedge i_1 = 10 \wedge i_1 \geq 10 \wedge i_0 < 10 \wedge \boldsymbol{priv}_0 < 0 \\ a = [\mathtt{priv} \geq 2; \mathtt{i} = 10] \end{cases}$$

Thanks to the reduction, we get information allowing for the low equality of `i` but also we get information about `priv` being positive. Finally, the last `if` statement will not be executed.

B SE step relation

This section shows the full set of rules of SE, the standard not-sound symbolic execution.

$$\text{S-ASSIGN}\ \frac{(\mathbf{e},\rho)\vdash_{\mathrm{s}}\varepsilon}{(\mathtt{x}:=\mathbf{e},(\rho,\pi))\rightharpoonup_s(\mathtt{skip},(\rho[\mathtt{x}\rightsquigarrow\langle\varepsilon\rangle],\pi))}$$

$$\text{S-SEQ-EXIT}\ \frac{}{(\mathtt{skip};\ \mathbf{c}_1,\kappa)\rightharpoonup_s(\mathbf{c}_1,\kappa)}\qquad \text{S-SEQ}\ \frac{(\mathbf{c}_0,\kappa)\rightharpoonup_s(\mathbf{c}_0',\kappa')}{(\mathbf{c}_0;\ \mathbf{c}_1,\kappa)\rightharpoonup_s(\mathbf{c}_0';\ \mathbf{c}_1,\kappa')}$$

$$\text{S-IF-T}\ \frac{(\mathbf{b},\rho)\vdash_{\mathrm{s}}\beta \qquad \pi'\triangleq\pi\wedge\beta \qquad \mathbf{may}(\pi')}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1,(\rho,\pi))\rightharpoonup_s(\mathbf{c}_0,(\rho,\pi))}$$

$$\text{S-IF-F}\ \frac{(\mathbf{b},\rho)\vdash_{\mathrm{s}}\beta \qquad \pi'\triangleq\pi\wedge\neg\beta \qquad \mathbf{may}(\pi')}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1,(\rho,\pi))\rightharpoonup_s(\mathbf{c}_1,(\rho,\pi))}$$

$$\text{S-LOOP-T}\ \frac{(\mathbf{b},\rho)\vdash_{\mathrm{s}}\beta \qquad \pi'\triangleq\pi\wedge\beta \qquad \mathbf{may}(\pi')}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c},(\rho,\pi))\rightharpoonup_s(\mathbf{c};\ \mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c},(\rho,\pi))}$$

$$\text{S-LOOP-F}\ \frac{(\mathbf{b},\rho)\vdash_{\mathrm{s}}\beta \qquad \pi'\triangleq\pi\wedge\neg\beta \qquad \mathbf{may}(\pi')}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c},(\rho,\pi))\rightharpoonup_s(\mathtt{skip},(\rho,\pi))}$$

C SoundSE step relation

This section shows the full set of rules of SoundSE by using SE, in Appendix B.

$$\text{S-NEXT}\ \frac{(\mathbf{c},\kappa)\rightharpoonup_s(\mathbf{c}',\kappa') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w)=(\mathbf{tt},w')}{(\mathbf{c},\kappa,w,b)\rightharpoonup_s(\mathbf{c}',\kappa',w',b)}$$

$$\text{S-APPROX-MANY}\ \frac{(\mathbf{c},\kappa)\rightharpoonup_s(\mathbf{c}',\kappa') \qquad \mathfrak{step}(\mathbf{c},\mathbf{c}',w)=(\mathbf{ff},w') \qquad \rho''=\mathfrak{modif}(\rho,\mathbf{c})}{(\mathbf{c},(\rho,\pi),w,b)\rightharpoonup_s(\mathtt{skip},(\rho'',\pi),w',\mathbf{ff})}$$

D Abstract step relation

This section shows the full set of rules of the abstract analysis used in RedSoundSE.

$$\text{A-ASSIGN}\ \frac{a' \triangleq \mathfrak{assign}_{\mathbf{x},\mathbf{e}}(a)}{(\mathtt{x} := \mathbf{e}, a) \rightharpoonup_{\mathbb{A}} (\mathtt{skip}, a')} \qquad \text{A-IF-T}\ \frac{a' \triangleq \mathfrak{guard}_{\mathbf{b}}(a) \qquad a' \neq \bot}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}_0, a')}$$

$$\text{A-IF-F}\ \frac{a' \triangleq \mathfrak{guard}_{\neg\mathbf{b}}(a) \qquad a' \neq \bot}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}_0, a')}$$

$$\text{A-SEQ-EXIT}\ \frac{}{(\mathtt{skip};\ \mathbf{c}_1, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}_1, a)} \qquad \text{A-SEQ}\ \frac{(\mathbf{c}_0, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}_0', a')}{(\mathbf{c}_0;\ \mathbf{c}_1, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}_0';\ \mathbf{c}_1, a')}$$

$$\text{A-LOOP-F}\ \frac{a' \triangleq \mathfrak{guard}_{\neg\mathbf{b}}(a) \qquad a' \neq \bot}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}_0, a) \rightharpoonup_{\mathbb{A}} (\mathtt{skip}, a')}$$

$$\text{A-LOOP-T}\ \frac{a' \triangleq \mathfrak{guard}_{\mathbf{b}}(a) \qquad a' \neq \bot}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}_0, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c};\ \mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}_0, a')}$$

E RedSoundSE step relation

RedSoundSE is defined by rules of Appendix B, Appendix C and Appendix D.

$$\text{S-A-NEXT}\ \frac{\begin{array}{c}(\mathbf{c}, \kappa, w, b) \rightharpoonup_s (\mathbf{c}', \kappa', w', b) \\ \mathfrak{step}(\mathbf{c}, \mathbf{c}', w) = (\mathbf{tt}, w') \qquad (\mathbf{c}, a) \rightharpoonup_{\mathbb{A}} (\mathbf{c}', a') \qquad (\kappa'', a'') \triangleq \mathfrak{reduction}(\kappa', a')\end{array}}{(\mathbf{c}, \kappa, a, w, b) \rightharpoonup_{s \times \mathbb{A}} (\mathbf{c}', \kappa'', a'', w', b)}$$

$$\text{S-A-APPROX-MANY}\ \frac{\begin{array}{c}(\mathbf{c}, \kappa) \rightharpoonup_s (\mathbf{c}', \kappa') \qquad \mathfrak{step}(\mathbf{c}, \mathbf{c}', w) = (\mathbf{ff}, w') \\ \kappa'' = \mathfrak{modif}(\kappa, \mathbf{c}) \qquad a' = [\![\mathbf{c}]\!]^{\sharp}_{\mathbb{A}}(a) \qquad (\kappa''', a''') \triangleq \mathfrak{reduction}(\kappa'', a')\end{array}}{(\mathbf{c}, (\kappa, a), w, b) \rightharpoonup_{s \times \mathbb{A}} (\mathtt{skip}, (\kappa''', a'''), w', \mathbf{ff})}$$

F RSE and SoundRSE step relations

This section shows the full set of rules for SoundRSE. RSE is a subset of SoundRSE, by removing rule SR-APPROX-MANY, and removing the counter and boolean flag.

$$\text{SR-ASSIGN}\ \frac{(\mathbf{e},\tilde{\rho})\vdash_{\mathrm{sr}}\tilde{\varepsilon}}{(\mathtt{x}:=\mathbf{e},(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathtt{skip},(\tilde{\rho}[\mathtt{x}\rightsquigarrow\langle\tilde{\varepsilon}\rangle],\pi),w,b)}$$

$$\text{SR-SEQ-EXIT}\ \frac{}{(\mathtt{skip};\ \mathbf{c}_1,\tilde{\kappa})\rightharpoonup_{sr}(\mathbf{c}_1,\tilde{\kappa})}\qquad\text{SR-SEQ}\ \frac{(\mathbf{c}_0,\tilde{\kappa})\rightharpoonup_{sr}(\mathbf{c}_0',\tilde{\kappa}')}{(\mathbf{c}_0;\ \mathbf{c}_1,\tilde{\kappa})\rightharpoonup_{sr}(\mathbf{c}_0';\ \mathbf{c}_1,\tilde{\kappa}')}$$

$$\text{SR-IF-TT}\ \frac{(\mathbf{b},\tilde{\rho})\vdash_{\mathrm{sr}}\tilde{\beta}\qquad\pi'=\pi\wedge\mathit{\Pi}_0(\tilde{\beta})\wedge\mathit{\Pi}_1(\tilde{\beta})\qquad\mathbf{may}(\pi')}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathbf{c}_0,(\tilde{\rho},\pi'),w,b)}$$

$$\text{SR-IF-TF}\ \frac{(\mathbf{b},\tilde{\rho})\vdash_{\mathrm{sr}}\tilde{\beta}\qquad\pi'=\pi\wedge\mathit{\Pi}_0(\tilde{\beta})\wedge\neg\mathit{\Pi}_1(\tilde{\beta})\qquad\mathbf{may}(\pi')}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathbf{c}_0\bowtie\mathbf{c}_1,(\tilde{\rho},\pi'),w,b)}$$

$$\text{SR-IF-FT}\ \frac{(\mathbf{b},\tilde{\rho})\vdash_{\mathrm{sr}}\tilde{\beta}\qquad\pi'=\pi\wedge\neg\mathit{\Pi}_0(\tilde{\beta})\wedge\mathit{\Pi}_1(\tilde{\beta})\qquad\mathbf{may}(\pi')}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathbf{c}_1\bowtie\mathbf{c}_0,(\tilde{\rho},\pi'),w,b)}$$

$$\text{SR-IF-FF}\ \frac{(\mathbf{b},\tilde{\rho})\vdash_{\mathrm{sr}}\tilde{\beta}\qquad\pi'=\pi\wedge\neg\mathit{\Pi}_0(\tilde{\beta})\wedge\neg\mathit{\Pi}_1(\tilde{\beta})\qquad\mathbf{may}(\pi')}{(\mathtt{if}\ \mathbf{b}\ \mathtt{then}\ \mathbf{c}_0\ \mathtt{else}\ \mathbf{c}_1,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathbf{c}_0,(\tilde{\rho},\pi'),w,b)}$$

$$\text{SR-LOOP-TT}\ \frac{\begin{array}{c}\mathfrak{step}(\mathbf{c},\mathbf{c}',w)=(\mathbf{tt},w')\\(\mathbf{b},\tilde{\rho})\vdash_{\mathrm{sr}}\tilde{\beta}\qquad\pi'=\pi\wedge\mathit{\Pi}_0(\tilde{\beta})\wedge\mathit{\Pi}_1(\tilde{\beta})\qquad\mathbf{may}(\pi')\end{array}}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}_0,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathbf{c}_0;\ \mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}_0,(\tilde{\rho},\pi),w',b)}$$

$$\text{SR-LOOP-TF}\ \frac{\begin{array}{c}\mathfrak{step}(\mathbf{c},\mathbf{c}',w)=(\mathbf{tt},w')\\(\mathbf{b},\tilde{\rho})\vdash_{\mathrm{sr}}\tilde{\beta}\qquad\pi'=\pi\wedge\mathit{\Pi}_0(\tilde{\beta})\wedge\neg\mathit{\Pi}_1(\tilde{\beta})\qquad\mathbf{may}(\pi')\end{array}}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}_0,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}((\mathbf{c}_0;\ \mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}_0)\bowtie\mathtt{skip},(\tilde{\rho},\pi'),w',b)}$$

$$\text{SR-LOOP-FT}\ \frac{\begin{array}{c}\mathfrak{step}(\mathbf{c},\mathbf{c}',w)=(\mathbf{tt},w')\\(\mathbf{b},\tilde{\rho})\vdash_{\mathrm{sr}}\tilde{\beta}\qquad\pi'=\pi\wedge\neg\mathit{\Pi}_0(\tilde{\beta})\wedge\mathit{\Pi}_1(\tilde{\beta})\qquad\mathbf{may}(\pi')\end{array}}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}_0,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathtt{skip}\bowtie(\mathbf{c}_0;\ \mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}_0),(\tilde{\rho},\pi'),w',b)}$$

$$\text{SR-LOOP-FF}\ \frac{\begin{array}{c}\mathfrak{step}(\mathbf{c},\mathbf{c}',w)=(\mathbf{tt},w')\\(\mathbf{b},\tilde{\rho})\vdash_{\mathrm{sr}}\tilde{\beta}\qquad\pi'=\pi\wedge\neg\mathit{\Pi}_0(\tilde{\beta})\wedge\neg\mathit{\Pi}_1(\tilde{\beta})\qquad\mathbf{may}(\pi')\end{array}}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}_0,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathtt{skip},(\tilde{\rho},\pi'),w',b)}$$

$$\text{SR-EXIT}\ \frac{}{(\mathtt{skip}\bowtie\mathtt{skip},(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathtt{skip},(\tilde{\rho},\pi),w,b)}$$

$$\text{SR-COMP-R}\ \frac{(\mathbf{c}_1,(\mathit{\Pi}_1(\tilde{\rho}),\pi),w,b)\rightharpoonup_{s}(\mathbf{c}_1',(\rho_1',\pi'),w',b')}{(\mathtt{skip}\bowtie\mathbf{c}_1,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathtt{skip}\bowtie\mathbf{c}_1',(\llparenthesis\mathit{\Pi}_0(\tilde{\rho})\,|\,\rho_1'\rrparenthesis,\pi'),w',b')}$$

$$\text{SR-COMP-L}\ \frac{(\mathbf{c}_0,(\mathit{\Pi}_0(\tilde{\rho}),\pi),w,b)\rightharpoonup_{s}(\mathbf{c}_0',(\rho_0',\pi'),w',b')}{(\mathbf{c}_0\bowtie\mathbf{c}_1,(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathbf{c}_0'\bowtie\mathbf{c}_1,(\llparenthesis\rho_0'\,|\,\mathit{\Pi}_1(\tilde{\rho})\rrparenthesis,\pi'),w',b')}$$

$$\text{SR-APPROX-MANY}\ \frac{\begin{array}{c}\mathfrak{step}(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c},(\mathbf{c};\ \mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c}),w)=(\mathbf{ff},w')\\\tilde{\rho}''=\mathfrak{modif}(\tilde{\rho},\mathbf{c})\qquad(\mathbf{b},\tilde{\rho}'')\vdash_{\mathrm{sr}}\langle\beta_0,\beta_1\rangle\qquad\pi'\triangleq\pi\wedge\neg\beta_0\wedge\neg\beta_1\end{array}}{(\mathtt{while}\ \mathbf{b}\ \mathtt{do}\ \mathbf{c},(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr}(\mathtt{skip},(\tilde{\rho}'',\pi'),w',\mathbf{ff})}$$

G RedSoundRSE step relation

RedSoundRSE is defined by rules of Appendix F plus rule SR-APPROX-MANY-DEP.

$$\textsc{sr-approx-many-dep}\ \frac{\begin{array}{c}\mathfrak{step}(\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c},(\mathbf{c};\ \texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}),w)=(\mathbf{ff},w')\\ d=[\![\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c}]\!]^{\sharp}_{\mathbb{D}}(\tau_{s\to\mathbb{D}}(\tilde{\rho})) \qquad \tilde{\rho}''=\mathfrak{modif}_{\mathbb{D}}(\tilde{\rho},\mathbf{c},\lambda_{\mathbb{D}\to\mathbb{L}}(d))\\ (\mathbf{b},\tilde{\rho}'')\vdash_{\mathrm{sr}}\langle\beta_0,\beta_1\rangle \qquad \pi'\triangleq\pi\wedge\neg\beta_0\wedge\neg\beta_1\end{array}}{(\texttt{while}\ \mathbf{b}\ \texttt{do}\ \mathbf{c},(\tilde{\rho},\pi),w,b)\rightharpoonup_{sr\times\mathbb{D}}(\texttt{skip},(\tilde{\rho}'',\pi'),w',\mathbf{ff})}$$