Sound Abstract Nonexploitability Analysis

Francesco Parolini, Antoine Miné

Sound Abstract Nonexploitability Analysis

Francesco Parolini[0000−0002−1077−7812] and Antoine Miné[0000−0002−6375−3179]

Sorbonne Université, CNRS, LIP6, 75005 Paris, France
{francesco.parolini, antoine.mine}@lip6.fr

Abstract. Runtime errors that can be triggered by an attacker are sensibly more dangerous than others, as they not only result in program failure, but can also be exploited and lead to security breaches such as Denial-of-Service attacks or remote code execution. Proving the absence of exploitable runtime errors is challenging, as it involves combining classic techniques for safety with novel security analyses. While numerous approaches to statically detect runtime errors have been proposed, they lack the ability to classify program failures as potentially exploitable or not. In this paper, we bridge the gap between traditional safety properties and security hyperproperties by putting forward a novel definition of *nonexploitability*, which we leverage to propose a sound static analysis by abstract interpretation to prove the absence of exploitable runtime errors. While false alarms can occur, if our analysis determines that a program is *nonexploitable*, then there is a strong mathematical guarantee that it is *impossible* for an attacker to trigger a runtime error. Furthermore, our analysis reduces the noise generated from false positives by classifying each warning as security-critical or not. We implemented the first nonexploitability analyzer for a subset of C, and we evaluated it on a set of 77 real-world programs taken from the GNU Coreutils package that are long up to 4,188 lines of code. Our analysis was able to *prove* that more than 70% of the runtime errors previously reported (3,498 over 4,715) cannot be triggered by an attacker.

Keywords: Security and Privacy · Static Analysis · Abstract Interpretation.

1 Introduction

Program failures that can be triggered by a malicious user are sensibly more dangerous than others, as they can lead to security breaches. Attackers can exploit well-known runtime errors, such as index out-of-bounds and double free, to perform dangerous attacks including Denial-of-Service (DoS) attacks or remote code execution. Numerous companies identified such exploitable vulnerabilities in their systems, including Meta [5], Apple [4], and Google [6]. Microsoft recently published a report showing that consistently over 20 years, around 70% of the security breaches that have been reported in their systems are due to exploitable memory corruption [10]. As it is difficult to identify program errors with manual inspection, static analysis is an invaluable tool to automatically detect them.

While sound static analyzers can report *all* possible runtime errors, including the exploitable ones, they often raise a high number of false positives. If the noise generated by the false alarms is elevated, the report of the analyzer quickly becomes unintelligible, and it is then difficult to identify the true exploitable runtime errors. In order to filter out the warnings that do not concern security issues, it is necessary to combine a traditional analysis for safety properties with a security analysis for hyperproperties [21].

In this paper, we bridge the gap between classic safety and security. We first formalize *nonexploitability* as a hyperproperty, and then we propose an alternative characterization based on *semantically tainted* (i.e. user-controlled) variables. We leverage such a characterization to put forward a sound analysis by abstract interpretation [26] that can prove the absence of exploitable runtime errors. Our analysis has the capability to classify each warning by its threat level (security-related or not), which makes the report of the analyzer more intelligible.

We leverage an underlying abstract value domain to infer numeric invariants, which we pair with a *semantic taint analysis* that tracks the set of user-controlled variables. Combining the two is necessary in order to infer an overapproximation of the exploitable runtime errors. By taking advantage of the semantic information inferred by the abstract numeric domain, our taint analysis achieves enhanced precision compared to traditional methods. Furthermore, our framework can handle programming language features that are essential to analyze real-world programs, such as nondeterminism and runtime user input.

We implemented and evaluated the first analyzer for nonexploitability in the Mopsa [38] static analysis platform. The analysis targets a large subset of C and it is *fully automatic*. We analyzed 77 real-world programs, each up to 4,188 lines long, taken from the GNU Coreutils package, to which we added 13,261 test cases taken from the Juliet test suite developed by NIST [9]. We found that our tool can *prove* that more than 70% of the warnings previously raised by the analyzer (3,498 over 4,715) cannot be triggered by an attacker, while incurring a performance overhead of less than 16%.

In this paper, we claim the following contributions:

- We introduce a novel property, *nonexploitability*, and we give its semantic characterization as a hyperproperty.
- We put forward an alternative characterization of nonexploitability in terms of *semantically tainted* (i.e. user-controlled) variables.
- We introduce a new practical, modular analysis that combines a traditional value analysis with a taint analysis to prove nonexploitabilty.
- We implement our analysis and evaluate it on a large set of real-world C programs.

2 Motivation

Figure 1 represents an exploitable program where a buffer overflow can occur depending on the user's input. A malicious user can take advantage of this type of vulnerability to execute sophisticated, dangerous attacks. There are numerous

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void use_input(const char* input) {
5    char dest[10];
6    strcpy(dest, input);
7  }
8
9  void main() {
10   char buff[100];
11   fgets(buff, sizeof(buff), stdin);
12   use_input(buff);
13 }
```

Fig. 1: C program with exploitable buffer overflow

examples of well-known attacks that exploit runtime errors: among them we find the Morris Worm [48], Code Red [19], SQL Slammer [54], and Heartbleed [29]. Many companies detected exploitable runtime errors in their products, for instance Adobe [2], NVIDIA [3], Apple [4], Meta [5], and Google [6]. Microsoft recently published a report that shows that around 70% of the security vulnerabilities that they found in their systems are due to exploitable memory corruption [10].

While techniques such as testing and human inspection by security experts are useful to detect (exploitable) runtime errors, the only option to rule out their existence is through formal methods. In particular, abstract interpretation [26] has been effective in proving the absence of program failures in real-time avionics software [23]. While analyses by abstract interpretation are *sound*, they can often raise false positives. If the noise generated by the false alarms is too high, the analyzer quickly becomes unusable.

Reducing the number of false alarms is usually achieved by employing more precise abstract domains. This paper takes an orthogonal approach to the problem, reducing the number of alarms by reporting the subset of possible runtime errors that can be triggered by an attacker. As these errors are comparatively more dangerous than the others, the report of the analyzer becomes more intelligible, enhancing the usefulness of sound static analysis tools. Girol et al. make the same observation, leveraging the concept of *robust reachability* to identify errors that are relatively more dangerous [31]. A bug is *robustly reachable* if there exists a user input for which the bug is always reached, regardless of the value of the uncontrolled input. The main difference with our concept of exploitability, is that we require the user input to be actually used in triggering the bug, while strong reachability does not (see Section 8 for a detailed comparison).

Taint analysis is a popular technique used in computer security to track the flow of untrusted data within a program, and we leverage this method to prove nonexploitability. While taint analyzers often rely on heuristics to track the flow of unsafe data [13], our approach is *semantic*, namely grounded in a definition based on the formal semantics of programs. While formal methods techniques extensively studied the verification of security properties such as *noninterfer-*

$$
\begin{aligned}
&\texttt{P} \coloneqq \texttt{S} && \text{(Programs)}\\
&\texttt{S} \coloneqq \texttt{skip} \mid \texttt{x} \leftarrow \texttt{input()} \mid \texttt{x} \leftarrow \texttt{rand()} \mid \texttt{x} \leftarrow \texttt{A} && \text{(Statements)}\\
&\quad \mid \texttt{S; S} \mid \texttt{if (B) S else S} \mid \texttt{while (B) S}\\
&\texttt{A} \coloneqq \ n \mid \ \texttt{x} \mid \texttt{A} \diamond \texttt{A} \ (\diamond \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\}) && \text{(Arithmetic Expressions)}\\
&\texttt{B} \coloneqq \texttt{tt} \mid \texttt{ff} \mid \texttt{A < A} \mid \texttt{¬B} \mid \texttt{B} \diamond \texttt{B} \ (\diamond \in \{\texttt{\&\&}, \texttt{||}\}) && \text{(Boolean Expressions)}
\end{aligned}
$$

Fig. 2: Syntax of the WHILE language

ence [22,32,33], the exploitability analysis is, to the best of our knowledge, uninvestigated (see Section 8 for a comparison). This paper bridges the gap between classic safety properties analysis and security hyperproperties [21] in order to rule out the existence of exploitable runtime errors from a software system. Our approach is closely related to [24], which proposes a technique that can prove noninterference using the abstract interpretation framework. Nevertheless, our technique supports features such as nondeterminism and dynamic user input that are not considered in [24]. Furthermore, we leverage the values of the variables to enhance the precision of our analysis, which is an extension proposed but not implemented in [24].

3 Syntax and concrete semantics

In this section, we define a *reachability semantics* that computes the set of reachable program states. As we are interested in program errors, the semantics also collects the set of error states. The finite set of program variables is denoted as $\mathbb{V}$, and in Figure 2 we present the syntax of the WHILE language that we consider.

Expressions are deterministic, and nondeterminism is isolated in the language in specific statements (`rand`, `input`) to simplify the presentation. We define the set of *program memories* as $\mathbb{M} \triangleq \mathbb{V} \to \mathbb{Z}$. The value $\lightning$ represents a runtime error, and $\mathbb{Z}_\lightning \triangleq \mathbb{Z} \cup \{\lightning\}$. The *arithmetic evaluation* $\mathcal{A}[\![\texttt{A}]\!] : \mathbb{M} \to \mathbb{Z}_\lightning$ definition is straightforward: it results in an error if there is a division by zero. For illustration purposes, we consider only runtime failures arising from divisions by zero, but our implementation supports all classic C arithmetic and memory errors. The set $\mathbb{B}$ is $\{\texttt{tt}, \texttt{ff}\}$, and $\mathbb{B}_\lightning \triangleq \mathbb{B} \cup \{\lightning\}$. The *boolean evaluation* $\mathcal{B}[\![\texttt{B}]\!] : \mathbb{M} \to \mathbb{B}_\lightning$ results in a runtime error if the arithmetic evaluation results in a runtime error. The definitions of $\mathcal{A}[\![\texttt{A}]\!]$ and $\mathcal{B}[\![\texttt{B}]\!]$ are standard, and they are reported in Appendix A.

The *program states* are triplets $(m, i, r) \in \mathbb{M} \times \mathbb{Z}^\omega \times \mathbb{Z}^\omega \triangleq \mathbb{S}$. The first element is the program memory, the second is the unbounded sequence of inputs provided by the user, and the third is the unbounded sequence of random numbers. We explicitly represent states in which a runtime error occurred by setting a special *return* variable to 1. All error-free states have the return variable, denoted as `ret`, set to 0. Programs cannot read nor write explicitly to `ret`, as it cannot syn-

tactically appear in statements. Our semantics relies on pairs of initial-reachable states. A set of initial-reachable states is a relation $\mathcal{R} \in \wp(\mathbb{S} \times \mathbb{S}) \triangleq \mathbb{D}$.

In this section, we will define the *reachability semantics* of statements $\mathcal{S}[\![\mathtt{S}]\!] : (\mathbb{D} \times \mathbb{D}) \to (\mathbb{D} \times \mathbb{D})$ by induction. The first element in the input pair is the set of pre-post states that reach the current statement without encountering an error, while the second is the set of pre-post states that previously resulted in an error. $\mathcal{S}[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E})$ outputs both the reachable and the error states after executing $\mathtt{S}$. The set of initial states is $\mathcal{I} \triangleq \{\, ((m,i,r),(m,i,r)) \mid (m,i,r) \in \mathbb{S}, m[\![\mathtt{ret}]\!] = 0 \,\}$. We define the semantics of programs $\mathcal{S}[\![\mathtt{P}]\!] \in \mathbb{D}$ by merging the reachable states at the end of the program with those that resulted in an error. Let $\mathtt{P} \coloneqq \mathtt{S}$.

$$\mathcal{S}[\![\mathtt{P}]\!] \triangleq \text{let } (\mathcal{R}, \mathcal{E}) = \mathcal{S}[\![\mathtt{S}]\!](\mathcal{I}, \varnothing) \text{ in } \mathcal{R} \cup \mathcal{E}$$

We now define by structural induction the reachability semantics of statements. As the definitions for $\mathtt{skip}$ and $\mathtt{S}_1\,;\,\mathtt{S}_2$ are standard, we do not report them. For the input read statement, we update the memory by assigning the first number in the infinite input sequence to the assigned variable, and then we shift the input sequence. The operator $\circ$ denotes the relation composition, while $\mathtt{hd}$ and $\mathtt{tl}$ respectively extract the head and the tail of a sequence.

Input read statement ($\mathtt{S} \coloneqq \mathtt{x} \leftarrow \mathtt{input()}$).

$$\mathcal{S}[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}) \triangleq (\mathcal{R} \circ \{\, ((m,i,r),(m[\mathtt{x} \leftarrow \mathtt{hd}(i)], \mathtt{tl}(i), r) \mid (m,i,r) \in \mathbb{S}) \,\}, \mathcal{E})$$

The random read statement is similar to the input read statement, but uses the infinite sequence of random numbers.

Random read statement ($\mathtt{S} \coloneqq \mathtt{x} \leftarrow \mathtt{rand()}$).

$$\mathcal{S}[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}) \triangleq (\mathcal{R} \circ \{\, ((m,i,r),(m[\mathtt{x} \leftarrow \mathtt{hd}(r)], i, \mathtt{tl}(r)) \mid (m,i,r) \in \mathbb{S}) \,\}, \mathcal{E})$$

Assignments can result in errors, which in our semantics are represented as states where $\mathtt{ret}$ is 1. If a runtime error occurs, the program sets $\mathtt{ret}$ to 1 and adds the state to the second element of the output pair. Error states are collected throughout the execution, and are propagated at the end of the program even in case of non-termination. Note, however, that non-termination is not considered to be an error. We define $\mathtt{ok}[\![\mathtt{A}]\!] : \mathbb{D} \to \mathbb{D}$ and $\mathtt{err}[\![\mathtt{A}]\!] : \mathbb{D} \to \mathbb{D}$ to collect respectively regular and error states in the evaluation of $\mathtt{A}$.

$$\mathtt{ok}[\![\mathtt{A}]\!]\mathcal{R} \triangleq \mathcal{R} \circ \{\, ((m,i,r),(m[\mathtt{x} \leftarrow \mathcal{A}[\![\mathtt{A}]\!]m], i, r)) \mid (m,i,r) \in \mathbb{S}, \mathcal{A}[\![\mathtt{A}]\!]m \neq \lightning \,\}$$
$$\mathtt{err}[\![\mathtt{A}]\!]\mathcal{R} \triangleq \mathcal{R} \circ \{\, ((m,i,r),(m[\mathtt{ret} \leftarrow 1], i, r)) \mid (m,i,r) \in \mathbb{S}, \mathcal{A}[\![\mathtt{A}]\!]m = \lightning \,\}$$

Assignment statement ($\mathtt{S} \coloneqq \mathtt{x} \leftarrow \mathtt{A}$)

$$\mathcal{S}[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}) \triangleq (\mathtt{ok}[\![\mathtt{A}]\!]\mathcal{R}, \mathcal{E} \cup \mathtt{err}[\![\mathtt{A}]\!]\mathcal{R})$$

We define $\texttt{test}[\![\texttt{B}]\!] : \mathbb{D} \to \mathbb{D}$ to filter states according to a boolean condition B: $\texttt{test}[\![\texttt{B}]\!]\mathcal{R} \triangleq \mathcal{R} \circ \{((m,i,r),(m,i,r)) \mid (m,i,r) \in \mathbb{S}, \mathcal{B}[\![\texttt{B}]\!]m = \texttt{tt}\}$. We abuse the notation, and we use $\texttt{err}[\![\texttt{B}]\!]$ for the boolean evaluation of errors.

If statement ($\texttt{S} \coloneqq \texttt{if (B) } \texttt{S}_t \texttt{ else } \texttt{S}_e$)

$$\begin{aligned}\mathcal{S}[\![\texttt{S}]\!](\mathcal{R},\mathcal{E}) \triangleq\ & \text{let } (\mathcal{R}_t,\mathcal{E}_t) = \mathcal{S}[\![\texttt{S}_t]\!](\texttt{test}[\![\texttt{B}]\!]\mathcal{R},\mathcal{E}) \text{ in}\\ & \text{let } (\mathcal{R}_e,\mathcal{E}_e) = \mathcal{S}[\![\texttt{S}_e]\!](\texttt{test}[\![\neg\texttt{B}]\!]\mathcal{R},\mathcal{E}) \text{ in}\\ & (\mathcal{R}_t \cup \mathcal{R}_e, \mathcal{E}_t \cup \mathcal{E}_e \cup \texttt{err}[\![\texttt{B}]\!]\mathcal{R})\end{aligned}$$

The semantics of while statements is a classic fixpoint definition. The operator $\dot\cup$ denotes the point-wise set union on pairs. For the rest of the paper, we denote the point-wise lifting of operator $\diamond$ to tuples as $\dot\diamond$.

While statement ($\texttt{S} \coloneqq \texttt{while (B) } \texttt{S}_b$)

$$\begin{aligned}&\mathcal{S}[\![\texttt{S}]\!](\mathcal{R},\mathcal{E}) \triangleq \text{let } (\mathcal{R}_f,\mathcal{E}_f) = \operatorname{lfp} F \text{ in } (\texttt{test}[\![\neg\texttt{B}]\!]\mathcal{R}_f,\mathcal{E}_f)\\ &\qquad \text{where } F(\mathcal{R}_1,\mathcal{E}_1) \triangleq (\mathcal{R},\mathcal{E}) \mathbin{\dot\cup} \mathcal{S}[\![\texttt{if (B) } \texttt{S}_b \texttt{ else skip}]\!](\mathcal{R}_1,\mathcal{E}_1)\end{aligned}$$

4 Nonexploitability

In this section, we first give a formal definition of *nonexploitability* as a hyperproperty [21]. Then, we put forward an alternative characterization based on semantically tainted (i.e. user-controlled) variables, which we leverage to introduce a sound, effective analysis for nonexploitability. The proofs of the theoretical results are reported in Appendix B.

Nonexploitability formalizes the idea that by modifying only the user input at the beginning of a program, it is not possible to change whether the program results in a runtime error or not. Since we designed our concrete semantics to explicitly represent runtime errors as states with the return variable set to 1, we use program memories to differentiate erroneous states from regular ones.

Definition 1 (Nonexploitability).

$$\begin{aligned}\mathscr{NE} \triangleq \{\, \mathcal{R} \in \mathbb{D} \mid \forall ((m_0,i_0,r_0),(m_1,i_1,r_1)),((m'_0,i'_0,r'_0),(m'_1,i'_1,r'_1)) \in \mathcal{R} :\\ m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 \implies m_1[\texttt{ret}] = m'_1[\texttt{ret}] \,\}\end{aligned}$$

Example 1. Accordingly to our definition, the following program is exploitable: `x ← input(); 1/x`. This is because if we consider two initial states, one in which the first element of the input sequence is zero, and the other in which it is not, we observe that the value of `ret` changes. Conversely, the program `x ← rand(); 1/x` is *nonexploitable*: even if there is a possible division by zero, once we fix the sequence of random numbers, changing the user input does not result in modifying the value of `ret`. If we did not compare pairs of initial states with the *same* sequence of random numbers, the program would be exploitable, even if the user input is never read.

Example 2 (Comparison with robust reachability [31]). A bug is *robustly reachable* if there exists a user input for which the bug is always reached, regardless of the value of the random input. Consider a program that always results in a division by zero: `1/0`. The program is nonexploitable: for any possible user input, the value of `ret` will always be 1. Conversely, the error is robustly reachable, as it is trivially reached for *any* user input. This highlights an important difference between the two concepts: nonexploitability requires the user input to be *effectively used* in triggering program errors, while robust reachability does not.

In what follows, we show that nonexploitability can be expressed in terms of *semantically tainted variables*. Intuitively, a variable is tainted if an attacker can control its value. Taint analysis [39] is a well-known technique in computer security to track the variables that are controlled by external users. However, many existing approaches use heuristics and syntactic formulations of the problem, which may be both imprecise and unsound. In contrast, we rely on a *semantic* approach, which is grounded in the formal semantics of programs. The following hyperproperty captures the set of semantics where the value of a variable `x` depends on the user's input, i.e. `x` is tainted. We compare pairs of executions that in the initial states differ only for the user's input, but then result in different values for `x`. The definition formalizes the intuition that `x` is tainted if, by modifying only the user input, it is possible to change the value of `x`.

Definition 2 (Taint). *Let* $\mathtt{x} \in \mathbb{V}$.

$$\mathscr{T} : \mathbb{V} \to \wp(\mathbb{D})$$
$$\mathscr{T}(\mathtt{x}) \triangleq \{\mathcal{R} \in \mathbb{D} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in \mathcal{R} : m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 : m_1[\mathtt{x}] \neq m'_1[\mathtt{x}]\}$$

We define abstraction and concretization functions for $\wp(\mathbb{V})$.

$$\alpha_t : \wp(\mathbb{D}) \to \wp(\mathbb{V}) \qquad \gamma_t : \wp(\mathbb{V}) \to \wp(\mathbb{D})$$
$$\alpha_t(\mathscr{R}) \triangleq \{\, \mathtt{x} \in \mathbb{V} \mid \mathscr{R} \subseteq \mathscr{T}(\mathtt{x}) \,\} \qquad \gamma_t(\mathcal{T}) \triangleq \bigcap_{\mathtt{x} \in \mathcal{T}} \mathscr{T}(\mathtt{x})$$

As it turns out, there is a Galois connection $(\wp(\mathbb{D}), \subseteq) \underset{\alpha_t}{\overset{\gamma_t}{\leftrightarrows}} (\wp(\mathbb{V}), \supseteq)$ between $\wp(\mathbb{D})$ and $\wp(\mathbb{V})$ defined by α_t and γ_t. The order for the abstract domain $\wp(\mathbb{V})$ is $\supseteq$ because if we consider more relations, we obtain fewer tainted variables common to *all* of these relations. Notice that this is different from observing that larger relations present more tainted variables, which will be discussed later in this section. A variable `x` is tainted in a program `P` if $\mathtt{x} \in \alpha_t(\{\mathcal{S}[\![\mathtt{P}]\!]\})$.

Example 3 (Implicit flows). If statements can generate *implicit flows* [28], namely dependencies that arise from the program control flow. Consider the following: `x ← input(); if (x==0) y ← 1 else y ← 2`. Depending on the user's input, `y` can be either 1 or 2, and accordingly to our semantic characterization of tainted variables, `y` is tainted. Taint analyzers (e.g. [7,8,13,56]) often ignore implicit

flows, considering only *explicit flows* (i.e. when tainting is propagated through assignments only), which is unsound in our framework. In the analysis described in Section 6, we develop an abstraction that does take implicit flows into account.

If the user cannot control the value of `ret`, then they cannot control whether there is a runtime error, i.e. the program is nonexploitable. This is the fundamental observation used in the following alternative characterization of $\mathscr{NE}$.

$$\mathcal{R} \in \mathscr{NE} \iff \texttt{ret} \notin \alpha_t(\{\mathcal{R}\}) \tag{1}$$

Equation (1) is significant because it shows that nonexploitability can be verified with a taint analysis. In contrast to classic taint analyses, simply tracking the set of user-controlled variables is not sufficient, as to infer whether `ret` is tainted we also need to detect runtime errors. In fact, `ret` does not syntactically appear in programs, and its value changes only when program failures occur. To determine when this happens, is it important to consider the *values* of the variables. Without semantic information about the values of the variables, every expression with a division should be considered dangerous in order to be sound, and this would result in an unacceptable loss of precision. In Section 6 we put forward a sound analysis by abstract interpretation that can prove programs to be nonexploitable by combining a classic value analysis with a taint analysis. The former detects program locations that potentially present runtime errors, while the latter determines whether the user can trigger those errors.

Hyperproperties verification is challenging for analyses based on abstract interpretation, because not every hyperproperty is *subset-closed* [21]: by computing an overapproximation $\mathcal{R}_1$ of $\mathcal{R}_0$, the fact that $\mathcal{R}_1$ respects an hyperproperty does not, in the general case, imply that $\mathcal{R}_0$ respects the hyperproperty. To overcome this problem, many works rely on *hypersemantics* [40,41,42,59,14]: the concrete semantics of a program is a set of sets of states, in contrast to a classic set of states. The main disadvantage of hypersemantics is that *hyper-domains* [40,41,42] are incompatible with regular abstract domains: the former abstract hypersemantics, while the latter abstract regular semantics.

In this paper, we rely on the standard abstract interpretation framework. In the rest of this section, we show that an overapproximation of the concrete semantics is sufficient to prove nonexploitability. A significant benefit of using the standard framework is that we can combine a taint analysis with any existing over-approximating value domain, which leads to a modular design. Furthermore, enhancing the precision of the numeric analysis improves the precision of the taint analysis as well. Observe that, as discussed in this section, in our context, it is important to rely on a classic safety analysis (and hence, on regular abstract numeric domains) to identify expressions that potentially present runtime errors.

We observe that larger semantics have more tainted variables. This holds due to the existential quantifier in Definition 2. Let $\mathcal{R}_0, \mathcal{R}_1 \in \mathbb{D}$.

$$\mathcal{R}_0 \subseteq \mathcal{R}_1 \implies \alpha_t(\{\mathcal{R}_0\}) \subseteq \alpha_t(\{\mathcal{R}_1\}) \tag{2}$$

By using this result, we observe that if `ret` is not tainted in $\mathcal{R}_1$, it cannot be tainted in $\mathcal{R}_0$. This implies that if $\mathcal{R}_1$ is nonexploitable, then $\mathcal{R}_0$ is nonexploitable, namely $\mathscr{NE}$ is subset-closed.

Theorem 1 ($\mathscr{NE}$ is subset-closed). *Let* $\mathcal{R}_0, \mathcal{R}_1 \in \mathbb{D}$.

$$(\mathcal{R}_0 \subseteq \mathcal{R}_1 \text{ and } \mathcal{R}_1 \in \mathscr{NE}) \implies \mathcal{R}_0 \in \mathscr{NE}$$

Theorem 1 is significant because it implies that by overapproximating the semantics of a program, we can still prove that it is nonexploitable. This justifies why the standard abstract interpretation framework is sufficient, and allows using the large library of existing abstract value domains. The theorem formalizes the intuition that if it is not possible for an attacker to trigger any runtime error, by further reducing the semantics of the program–and hence the capabilities of the attacker–he is still not able to make the program fail.

5 Taint concrete semantics

In this section, we define the non-computable concrete taint semantics that we overapproximate in Section 6. The semantics associates the reachable states with the set of semantically tainted variables using the abstraction function α_t. As the semantics is not *structural* (i.e. defined by induction on the program syntax), we also develop a structural equivalent definition. This is necessary in order to overapproximate the concrete taint semantics with an inductive and effectively computable abstract semantics.

We first define the reachability taint semantics of statements $\mathcal{S}_t[\![\mathtt{S}]\!] : (\mathbb{D} \times \mathbb{D}) \to (\mathbb{D} \times \mathbb{D} \times \wp(\mathbb{V}))$. This semantics associates each statement with its set of truly tainted variables by relying on α_t.

$$\mathcal{S}_t[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}) \triangleq \text{let } (\mathcal{R}_1, \mathcal{E}_1) = \mathcal{S}[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}) \text{ in } (\mathcal{R}_1, \mathcal{E}_1, \alpha_t(\{\, \mathcal{R}_1 \,\}))$$

We then define the reachability taint semantics for programs $\mathcal{S}_t[\![\mathtt{P}]\!] \in \mathbb{D} \times \wp(\mathbb{V})$. As regular and erroneous states are merged at the end of programs, we use α_t to obtain the tainted variables in the union. Let $\mathtt{P} \coloneqq \mathtt{S}$.

$$\mathcal{S}_t[\![\mathtt{P}]\!] \triangleq \text{let } (\mathcal{R}, \mathcal{E}, \mathcal{T}) = \mathcal{S}_t[\![\mathtt{S}]\!](\mathcal{I}, \varnothing) \text{ in } (\mathcal{R} \cup \mathcal{E}, \mathcal{T} \cup \alpha_t(\{\mathcal{R} \cup \mathcal{E}\}))$$

Observe that only at the end of the program `ret` can become tainted: regular and erroneous states are partitioned in the semantics for statements, so that `ret` is always constant (0 for the normal executions and 1 for the others). The program `P` is then nonexploitable iff `ret` is not tainted in $\mathcal{S}_t[\![\mathtt{P}]\!]$.

Example 4. The statement `x` ← `rand()` can taint `x` if there are two executions in which the sequence of random numbers is out-of-sync due to a user action. This is because in the definition of $\mathscr{T}$ we compare pairs of execution with the *same* sequence of random numbers. Consider the following program:

```
x ← input(); if (x != 0) { y ← rand() }; z ← rand()
```

The user can control whether `z` is assigned to the first or the second number in the random sequence. If we fix as random sequence $1, 2, \ldots$, we can observe that `z` can be either 1 or 2 at the end of the program, depending on the user's input.

```
1  void main() {
2    if (getchar() == 'a')
3      rand();
4    int z = rand();
5  }
```

Fig. 3: C program that reads pseudo-random numbers

Observe that this behaviour is relevant in scenarios where the attacker has partial knowledge about the uncontrolled random input. For instance, consider the program in Figure 3, where the application first reads a character from standard input. If the character is `a`, the program reads the first pseudo-random number, and then it assigns `z` to `rand()`. As the sequence of random numbers has not been initialized, it does not change and could be predicted across different executions. The user can make the program assign `z` to the first or second number in the sequence, being able to influence the assigned value. Another relevant case is when a program reads a file with unmodifiable but public content. If an attacker can control which bytes are read, then they can influence the execution of the program without even modifying the contents of the file.

The fact that random read statements can potentially taint the assigned variable directly derives from our semantic definition of $\mathscr{T}$. By changing the definition of $\mathscr{T}$, it is possible to choose whether random read statements can taint assigned variables. We make the choice to use random read statements as potential sources of tainted data because, in a context in which an attacker has (partial) knowledge about the unmodifiable pseudo-random input, such statements can be exploited to influence the execution of the program. While it would be possible to support the classic model where the attacker has no knowledge about the random input, this is less interesting in a context where security is considered fundamental.

As we want to overapproximate the concrete taint semantics by induction on the program structure, we give a structural equivalent definition of $\mathcal{S}_t[\![\mathtt{S}]\!]$. The non-computable semantics $\hat{\mathcal{S}}_t[\![\mathtt{S}]\!] : (\mathbb{D} \times \mathbb{D} \times \wp(\mathbb{V})) \to (\mathbb{D} \times \mathbb{D} \times \wp(\mathbb{V}))$ inductively collects the truly tainted variables. The semantics takes as additional input parameter the set of previously tainted variables, which are used to infer the set of tainted variables after the execution of the statement. Due to space constraints, here we present only the definition for assignments. The full definition of the structural taint semantics is reported in Appendix C.

Assignment statement ($\mathtt{S} \coloneqq \mathtt{x} \leftarrow \mathtt{A}$)

$$
\begin{aligned}
&\hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\mathcal{R},\mathcal{E},\mathcal{T}) \triangleq \\
&\quad \text{let } (\mathcal{R}_1,\mathcal{E}_1) = \mathcal{S}[\![\mathtt{S}]\!](\mathcal{R},\mathcal{E}) \text{ in} \\
&\quad \text{let } \mathcal{T}_1 = \{\, \mathtt{y} \in \mathcal{T} \mid \mathtt{y} \neq \mathtt{x} \,\} \cup \\
&\qquad \{\, \mathtt{x} \mid \exists((m_0,i_0,r_0),(m_1,i_1,r_1)),((m_0',i_0',r_0'),(m_1',i_1',r_1')) \in \mathcal{R} : \\
&\qquad\qquad m_0 = m_0', i_0 \neq i_0', r_0 = r_0' : \lightning \neq \mathcal{A}[\![\mathtt{A}]\!]m_1 \neq \mathcal{A}[\![\mathtt{A}]\!]m_1' \neq \lightning \,\} \text{ in} \\
&\quad (\mathcal{R}_1,\mathcal{E}_1,\mathcal{T}_1)
\end{aligned}
$$

We define the concrete inductive taint semantics for program $\mathtt{P} \coloneqq \mathtt{S}$ as follows.

$$\hat{\mathcal{S}}_t[\![\mathtt{P}]\!] \triangleq \text{let } (\mathcal{R},\mathcal{E},\mathcal{T}) = \hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\mathcal{I},\varnothing,\varnothing) \text{ in } (\mathcal{R} \cup \mathcal{E}, \mathcal{T} \cup \alpha_t(\{\, \mathcal{R} \cup \mathcal{E} \,\}))$$

The following result formalizes that $\hat{\mathcal{S}}_t[\![\mathtt{P}]\!]$ and $\mathcal{S}_t[\![\mathtt{P}]\!]$ are equivalent.

Theorem 2 (Correctness of $\hat{\mathcal{S}}_t[\![\mathtt{P}]\!]$). $\hat{\mathcal{S}}_t[\![\mathtt{P}]\!] \doteq \mathcal{S}_t[\![\mathtt{P}]\!]$

6 Taint abstract semantics

In this section, we introduce a computable sound overapproximation of the taint semantics presented in Section 5. This abstraction of the concrete non-computable semantics is parametric in the underlying abstract domain used to overapproxiamate the values of the variables. In contrast to traditional techniques, we leverage numeric invariants to improve the precision of the taint analysis.

Let $\mathbb{D}^\sharp$ be the abstract domain used to overapproximate $\mathbb{D}$, and $\gamma_d : \mathbb{D}^\sharp \to \mathbb{D}$ be the concretization function.[1] The domain $\mathbb{D}^\sharp$ is equipped with partial order $\subseteq_d^\sharp$ and abstract join $\cup_d^\sharp$, while $\bot_d^\sharp$ is the bottom element. We assume $\mathcal{S}_d^\sharp[\![\mathtt{S}]\!] : (\mathbb{D}^\sharp \times \mathbb{D}^\sharp) \to (\mathbb{D}^\sharp \times \mathbb{D}^\sharp)$ given by the numeric domain to be a sound computable abstraction of $\mathcal{S}[\![\mathtt{S}]\!]$: $\forall \mathcal{R}^\sharp, \mathcal{E}^\sharp \in \mathbb{D}^\sharp : \mathcal{S}[\![\mathtt{S}]\!](\dot{\gamma}_d(\mathcal{R}^\sharp,\mathcal{E}^\sharp)) \mathrel{\dot{\subseteq}} \dot{\gamma}_d(\mathcal{S}_d^\sharp[\![\mathtt{S}]\!](\mathcal{R}^\sharp,\mathcal{E}^\sharp))$. The abstract value domain also exposes the abstract functions $\mathtt{test}^\sharp[\![\mathtt{B}]\!]$ and $\mathtt{err}^\sharp[\![\mathtt{B}]\!]$ to overapproximate the concrete ones.

In the rest of the section, we structurally define the *abstract taint semantics* $\mathcal{S}_t^\sharp[\![\mathtt{S}]\!] : (\mathbb{D}^\sharp \times \mathbb{D}^\sharp \times \wp(\mathbb{V})) \to (\mathbb{D}^\sharp \times \mathbb{D}^\sharp \times \wp(\mathbb{V}))$. The semantics collects an overapproximation of the reachable states, the error states, and the tainted variables. The concretization function $\gamma : (\mathbb{D}^\sharp \times \mathbb{D}^\sharp \times \wp(\mathbb{V})) \to (\mathbb{D} \times \mathbb{D} \times \wp(\mathbb{V}))$ is defined as $\gamma(\mathcal{R}^\sharp,\mathcal{E}^\sharp,\mathcal{T}^\sharp) \triangleq (\gamma_d(\mathcal{R}^\sharp),\gamma_d(\mathcal{E}^\sharp),\mathcal{T}^\sharp)$. The soundness criterion states that the abstract semantics exhibits more tainted variables than those in the concrete semantics $\hat{\mathcal{S}}_t[\![\mathtt{S}]\!]$. Let $\mathcal{R}^\sharp, \mathcal{E}^\sharp \in \mathbb{D}^\sharp, \mathcal{T}^\sharp \in \wp(\mathbb{V})$.

$$\hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\gamma(\mathcal{R}^\sharp,\mathcal{E}^\sharp,\mathcal{T}^\sharp)) \mathrel{\dot{\subseteq}} \gamma(\mathcal{S}_t^\sharp[\![\mathtt{S}]\!](\mathcal{R}^\sharp,\mathcal{E}^\sharp,\mathcal{T}^\sharp)) \tag{3}$$

[1] While the concrete semantics is defined as a set of input-output relations to express nonexploitability, in the numeric abstraction it is possible to use numeric domains that abstract sets of states by abstracting only the image of the relations, and then consider each possible state as initial in the concretization.

In our abstract semantics, we taint `ret` every time there is a *possible* runtime error due to user input. This ensures that if `ret` is untainted in $\mathcal{S}_t^\sharp[\![\mathtt{S}]\!]$, it will be untainted at the end of the program, i.e. the program is nonexploitable. Let $\mathtt{P} \coloneqq \mathtt{S}$, and let $\mathcal{I}^\sharp \in \mathbb{D}^\sharp$ be an overapproximation of the set of initial states, namely $\mathcal{I} \subseteq \gamma_d(\mathcal{I}^\sharp)$. Let $(\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathcal{T}^\sharp) = \mathcal{S}_t^\sharp[\![\mathtt{S}]\!](\mathcal{I}^\sharp, \bot_d^\sharp, \varnothing)$.

$$\mathtt{ret} \notin \mathcal{T}^\sharp \implies \mathcal{S}[\![\mathtt{P}]\!] \in \mathscr{NE} \tag{4}$$

In the rest of this section, we define by structural induction $\mathcal{S}_t^\sharp[\![\mathtt{S}]\!]$. The abstract semantics collects an overapproximation of the tainted variables, and specifically taints `ret` whenever a runtime error potentially caused by the user occurs. We will take advantage of the helper function $\mathtt{taint}^\sharp[\![\mathtt{A}]\!] : (\mathbb{D}^\sharp \times \wp(\mathbb{V})) \to \mathbb{B}$ that returns `ff` only if the result of the evaluation of `A` definitely does not depend on tainted variables. Standard value-insensitive taint analyses ignore the values of the variables and simply return `tt` if a tainted variable *syntactically* appears in `A`. This is sound, but imprecise. For instance, consider the program $\mathtt{x} \leftarrow \mathtt{input()}\,;\, \mathtt{y} \leftarrow \mathtt{x}\,;\, \mathtt{z} \leftarrow \mathtt{x\text{-}y}$. The user cannot control the value of `z`, as it is always 0. By using a relational abstract domain such as polyhedra or octagons [43], it is possible to determine that `z` is constant, and therefore that it is not tainted. The actual definition of $\mathtt{taint}^\sharp[\![\mathtt{A}]\!]$ depends on the underlying abstract value domain, presenting numerous opportunities to improve the function's precision. In this section, we will take advantage of numerous helper functions that depend on the value domain, and in Appendix D we show a concrete instantiation of these functions for the interval abstract domain.

The abstract semantics for `skip` and $\mathtt{S}_1\,;\,\mathtt{S}_2$ are standard, and we do not report them. As variables read from user input are the main sources of tainted data, we always taint variables read from input statements.

Input read statement ($\mathtt{S} \coloneqq \mathtt{x} \leftarrow \mathtt{input()}$)

$$\mathcal{S}_t^\sharp[\![\mathtt{S}]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathcal{T}^\sharp) \triangleq \text{let } (\mathcal{R}_1^\sharp, \mathcal{E}_1^\sharp) = \mathcal{S}_d^\sharp[\![\mathtt{S}]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp) \text{ in } (\mathcal{R}_1^\sharp, \mathcal{E}_1^\sharp, \mathcal{T}^\sharp \cup \{\mathtt{x}\})$$

As observed in Section 5 (see Example 4), random read statements can taint the assigned variable in case the user controls the position of the value which is read in the random input sequence. For the abstract semantics, it would be sound to always taint the assigned variable. Nevertheless, this is too coarse, and we propose an abstraction that improves the precision. The idea is to represent the sequence of random numbers as a queue: programs read from it at index `i`, and then increment `i`. In this model, $\mathtt{x} \leftarrow \mathtt{rand()}$ is syntactically substituted with $\mathtt{x} \leftarrow \mathtt{rand[i]}\,;\, \mathtt{i} \leftarrow \mathtt{i+1}$. We assume that the abstract semantics $\mathcal{S}_d^\sharp[\![\mathtt{S}]\!]$ can handle reading from the queue. The special index variable `i` is then handled by the numeric domain as any other variable. We taint the result of $\mathtt{x} \leftarrow \mathtt{rand()}$ only if `i` is tainted: this happens when the user can control which number is read from the random sequence.

Random read statement ($\mathtt{S} \coloneqq \mathtt{x} \leftarrow \mathtt{rand()}$)

$$\begin{aligned}\mathcal{S}^\sharp_t[\![\mathtt{S}]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathcal{T}^\sharp) \triangleq\ & \text{let } (\mathcal{R}^\sharp_1, \mathcal{E}^\sharp_1) = \mathcal{S}^\sharp_d[\![\mathtt{x} \leftarrow \mathtt{rand[i]\,;\ i} \leftarrow \mathtt{i+1}]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp) \text{ in} \\ & \text{let } \mathcal{T}^\sharp_1 = \{\, \mathtt{y} \in \mathcal{T}^\sharp \mid \mathtt{y} \neq \mathtt{x} \,\} \cup \{\, \mathtt{x} \mid \mathtt{taint}^\sharp[\![\mathtt{i}]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \,\} \text{ in} \\ & (\mathcal{R}^\sharp_1, \mathcal{E}^\sharp_1, \mathcal{T}^\sharp_1)\end{aligned}$$

Assignments can present runtime errors, so that we need to taint `ret` in case the user can trigger a program failure. To determine if there is an exploitable runtime error in the evaluation of an expression, we rely on the function $\mathtt{exploit}^\sharp[\![\mathtt{A}]\!] : (\mathbb{D}^\sharp \times \wp(\mathbb{V})) \to \mathbb{B}$. The function returns `tt` if there is a possible runtime error when evaluating `A`, and such an error can be triggered by the user. We assume the existence of a function $\mathtt{zero}^\sharp[\![\mathtt{A}]\!] : \mathbb{D}^\sharp \to \mathbb{B}$, which is provided by the numeric domain and returns `tt` if the evaluation of `A` is possibly zero. Let $\mathtt{x} \in \mathbb{V}$. We define $\mathtt{exploit}^\sharp[\![\mathtt{x}]\!]$ and $\mathtt{exploit}^\sharp[\![n]\!]$ as `ff`, while for binary expressions we need to consider the values of the variables.

$$\mathtt{exploit}^\sharp[\![\mathtt{A}_1 \diamond \mathtt{A}_2]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \triangleq \begin{cases} \mathtt{tt} & \text{if } \diamond = /,\ \mathtt{zero}^\sharp[\![\mathtt{A}_2]\!]\mathcal{R}^\sharp,\ \mathtt{taint}^\sharp[\![\mathtt{A}_2]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \\ \mathtt{tt} & \text{if } \mathtt{exploit}^\sharp[\![\mathtt{A}_1]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \text{ or } \mathtt{exploit}^\sharp[\![\mathtt{A}_2]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \\ \mathtt{ff} & \text{otherwise} \end{cases}$$

Assignment statement ($\mathtt{S} \coloneqq \mathtt{x} \leftarrow \mathtt{A}$)

$$\begin{aligned}&\mathcal{S}^\sharp_t[\![\mathtt{S}]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathcal{T}^\sharp) \triangleq \\ &\quad \text{let } (\mathcal{R}^\sharp_1, \mathcal{E}^\sharp_1) = \mathcal{S}^\sharp_d[\![\mathtt{S}]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp) \text{ in} \\ &\quad \text{let } \mathcal{T}^\sharp_1 = \{\, \mathtt{y} \in \mathcal{T}^\sharp \mid \mathtt{y} \neq \mathtt{x} \,\} \cup \\ &\qquad \{\, \mathtt{x} \mid \mathtt{taint}^\sharp[\![\mathtt{A}]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \,\} \cup \{\, \mathtt{ret} \mid \mathtt{exploit}^\sharp[\![\mathtt{A}]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \,\} \text{ in} \\ &\quad (\mathcal{R}^\sharp_1, \mathcal{E}^\sharp_1, \mathcal{T}^\sharp_1)\end{aligned}$$

As discussed in Section 4 (see Example 3), if statements can generate *implicit flows* [28], namely dependencies that originate from the program control flow. When an attacker can control which branch of an if statement is executed, and in that branch a variable is assigned, then the variable could be tainted.

The set of variables that become tainted as a result of a tainted condition is traditionally overapproximated (when conditions are handled at all) with the variables that *syntactically* appear in the assignments of the branches. This is a coarse overapproximation, and we can improve this result by using the values of the variables. For instance, consider the program `x ← y; if (y < x) { z ← 10}`. The assignment is never executed, and a relational analysis can deduce that `z` is never assigned. The traditional syntactic approach is not sufficient to infer this information. We rely on the function $\mathtt{assigned}^\sharp[\![\mathtt{S}]\!] : \mathbb{D}^\sharp \to \wp(\mathbb{V})$ that returns an overapproximation of the set of variables that are *semantically* assigned when executing `S`. If there is a state in the concretization of the abstract

input $\mathcal{R}^\sharp$ in which a variable $\mathtt{x}$ changes value during the execution of $\mathtt{S}$, then $\mathtt{x} \in \mathtt{assigned}^\sharp[\![\mathtt{S}]\!]\mathcal{R}^\sharp$. Observe that in case an exploitable runtime error occurs, $\mathtt{assigned}^\sharp[\![\mathtt{S}]\!]\mathcal{R}^\sharp$ includes `ret`, which does not syntactically appear in the program. A straightforward implementation can run the regular value analysis and inductively collect the variables that are assigned. While doing this, the function discards unreachable code and assignments that do not modify the state, such as $\mathtt{x} \leftarrow \mathtt{0}$ when $\mathtt{x}$ is already 0, being effectively more precise than a syntactic approach. We define the following function to compute the set of variables that are tainted due to implicit flows.

$$\mathtt{diff}^\sharp[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \triangleq \\ \{\, \mathtt{x} \in \mathtt{assigned}^\sharp[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!]\mathcal{R}^\sharp \mid \mathtt{taint}^\sharp[\![\mathtt{B}]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \,\}$$

Tainted variables can also become untainted due to conditionals. For instance, the variable $\mathtt{x}$ is not tainted inside of the then branch in the following program: $\mathtt{x} \leftarrow \mathtt{input();\ if\ (x==0)\ \{...\}}$. The reason is that $\mathtt{x}$ equals zero when entering the first branch, and constants are by definition not controlled by the user. Classic methods ignore this, and do not filter tainted variables after conditionals. This is sound, but we can again achieve better precision by taking into account the values of the variables. We define the function $\mathtt{refine}^\sharp[\![\mathtt{B}]\!] : (\mathbb{D}^\sharp \times \wp(\mathbb{V})) \to \wp(\mathbb{V})$ as $\mathtt{refine}^\sharp[\![\mathtt{B}]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \triangleq \mathcal{T}^\sharp \backslash \mathtt{const}^\sharp(\mathtt{test}^\sharp[\![\mathtt{B}]\!]\mathcal{R}^\sharp)$, where $\mathtt{const}^\sharp$ returns the set of constant variables in the abstract state in its argument. The function $\mathtt{refine}^\sharp[\![\mathtt{B}]\!]$ filters out the variables that are definitely constant after the execution of the test $\mathtt{B}$, improving the precision of the analysis. We can now give the definition of the abstract semantics for if statements.

If statement $(\mathtt{S} \coloneqq \mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e)$

$$\begin{aligned}
&\mathcal{S}^\sharp_t[\![\mathtt{S}]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathcal{T}^\sharp) \triangleq \\
&\quad \text{let } (\mathcal{R}^\sharp_t, \mathcal{E}^\sharp_t, \mathcal{T}^\sharp_t) = \mathcal{S}^\sharp_t[\![\mathtt{S}_t]\!](\mathtt{test}^\sharp[\![\mathtt{B}]\!]\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathtt{refine}^\sharp[\![\mathtt{B}]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp)) \text{ in} \\
&\quad \text{let } (\mathcal{R}^\sharp_e, \mathcal{E}^\sharp_e, \mathcal{T}^\sharp_e) = \mathcal{S}^\sharp_t[\![\mathtt{S}_e]\!](\mathtt{test}^\sharp[\![\neg\mathtt{B}]\!]\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathtt{refine}^\sharp[\![\neg\mathtt{B}]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp)) \text{ in} \\
&\quad \text{let } \mathcal{T}^\sharp_{te} = \mathtt{diff}^\sharp[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!](\mathcal{R}^\sharp, \mathcal{T}^\sharp) \text{ in} \\
&\quad (\mathcal{R}^\sharp_t \cup^\sharp_d \mathcal{R}^\sharp_e, \mathcal{E}^\sharp_t \cup^\sharp_d \mathcal{E}^\sharp_e \cup^\sharp_d \mathtt{err}^\sharp[\![\mathtt{B}]\!]\mathcal{R}^\sharp, \mathcal{T}^\sharp_t \cup \mathcal{T}^\sharp_e \cup \mathcal{T}^\sharp_{te})
\end{aligned}$$

Example 5. The program in Figure 4a demonstrates various ways in which our analysis differs from other taint analyses. First, we can infer that the program is exploitable: if the user inputs zero, then there is the possibility, depending on the sequence of random numbers, that a runtime error is triggered. The value analysis is important to infer that $\mathtt{x}$ is zero when performing the division, so that we can deduce that there is a division by zero. Second, we can use the semantic information inferred by the numeric domain to deduce that $\mathtt{y}$ is not tainted. Even if $\mathtt{y}$ is assigned inside of a branch that depends on the user's input, the variable $\mathtt{y}$ does not change, as it is still 1 after the execution of the statement. An interval analysis is sufficient to deduce this. Third, we can infer that the

```
x ← input()
y ← 1
if (x==0) {
    z ← rand()
    if (z==0) { 1/x }
    if (z==1) { y ← z }
}
w ← rand()
```

(a)

```
x ← input()
if (x <= 0) { x ← 1 }
while (tt) {
    1 / x
    x ← rand()
}
```

(b)

Fig. 4: Programs that read values from the user and the random queue

variable `w` is tainted: depending on user input, it is assigned either to the first or the second value in the sequence of random numbers.

Further precision improvements can be implemented. For instance, consider the program `if (x < 10) { y ← 0} else { y ← 1}`. If the abstract value domain can determine that before the execution of the statement the value of `x` is less than 10, the statement is semantically equivalent to `y ← 0`. This implies that, even if `x` is tainted, the user cannot control the value of `y`. When the analysis can infer that one of the two branches is never executed, the if statement can be substituted with the other branch, ignoring the implicit flows that are generated by the condition, and improving again the precision.

The abstract semantics for while statements is a classic limit computation that relies on the widening operator ∇ to guarantee convergence in a finite number of iterations. As the number of variables is finite, $\wp(\mathbb{V})$ has finite height, so that the widening operator for $\wp(\mathbb{V})$ is simply the set union. The operator $(\mathcal{R}_1^\sharp, \mathcal{E}_1^\sharp, \mathcal{T}_1^\sharp) \dot\cup^\sharp (\mathcal{R}_2^\sharp, \mathcal{E}_2^\sharp, \mathcal{T}_2^\sharp)$ denotes $(\mathcal{R}_1^\sharp \cup_d^\sharp \mathcal{R}_2^\sharp, \mathcal{E}_1^\sharp \cup_d^\sharp \mathcal{E}_2^\sharp, \mathcal{T}_1^\sharp \cup \mathcal{T}_2^\sharp)$, and the operator $(\mathcal{R}_1^\sharp, \mathcal{E}_1^\sharp, \mathcal{T}_1^\sharp) \dot\nabla (\mathcal{R}_2^\sharp, \mathcal{E}_2^\sharp, \mathcal{T}_2^\sharp)$ denotes $(\mathcal{R}_1^\sharp \nabla \mathcal{R}_2^\sharp, \mathcal{E}_1^\sharp \nabla \mathcal{E}_2^\sharp, \mathcal{T}_1^\sharp \cup \mathcal{T}_2^\sharp)$.

While statement ($\mathtt{S} \coloneqq \mathtt{while\ (B)\ } \mathtt{S}_b$)

$$\mathcal{S}_t^\sharp[\![\mathtt{S}]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathcal{T}^\sharp) \triangleq \text{let } (\mathcal{R}_f^\sharp, \mathcal{E}_f^\sharp, \mathcal{T}_f^\sharp) = \lim F^n(\bot_d^\sharp, \bot_d^\sharp, \varnothing) \text{ in}$$
$$(\mathtt{test}^\sharp[\![\neg \mathtt{B}]\!]\mathcal{R}_f^\sharp, \mathcal{E}_f^\sharp, \mathtt{refine}^\sharp[\![\neg \mathtt{B}]\!](\mathcal{R}_f^\sharp, \mathcal{T}_f^\sharp))$$

where

$$F(\mathcal{R}_1^\sharp, \mathcal{E}_1^\sharp, \mathcal{T}_1^\sharp) \triangleq \text{let } (\mathcal{R}_2^\sharp, \mathcal{E}_2^\sharp, \mathcal{T}_2^\sharp) = \mathcal{S}_t^\sharp[\![\mathtt{if\ (B)\ } \mathtt{S}_b \mathtt{\ else\ skip}]\!](\mathcal{R}_1^\sharp, \mathcal{E}_1^\sharp, \mathcal{T}_1^\sharp) \text{ in}$$
$$(\mathcal{R}_1^\sharp, \mathcal{E}_1^\sharp, \mathcal{T}_1^\sharp) \dot\nabla ((\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathcal{T}^\sharp) \dot\cup^\sharp (\mathcal{R}_2^\sharp, \mathcal{E}_2^\sharp, \mathcal{T}_2^\sharp))$$

Example 6. In principle, it is possible to first run a value analysis, and then use the inferred numeric invariants in a taint analysis to prove nonexploitability.

Nevertheless, as shown by the program in Figure 4b, executing the two together achieves strictly superior precision by leveraging the reduction between the domains. The invariant inferred at statement `1/x` entails that `x` can be zero, so that there is a potential runtime error. Furthermore, a taint analysis infers that `x` is tainted at the same program location. By combining these information, the division by zero is exploitable. However, if we execute the value and the taint analyses together, we can observe that it is never true *at the same time* that `x` is 0 and tainted, so that the program failure cannot be triggered by an attacker. Our framework runs the two analyses together, and is thus able to prove that the program is nonexploitable. The reduction between the two domains can also refine the taint information using the value information, and this can improve subsequent results.

7 Experimental evaluation

Implementation. We propose Mopsa-Nexp, the *first* analyzer dedicated to nonexploitability. We implemented our analysis for a large subset of C in the Mopsa framework [38], which is a modular platform to build static analyzers based on abstract interpretation. Mopsa offers an extensive collection of ready-to-use abstract domains for analyzing C and Python, providing the flexibility to tune the tradeoff between precision and performance. Mopsa is implemented in 120,000 lines of OCaml code, and our exploitability analysis accounts for around 10,000 of them. Thanks to Mopsa's modular design, we were able to use most of the C analysis with minimal modifications.

In our implementation, we maintain taint information at the level of *memory blocks*, i.e. we perform a *field-insensitive* taint analysis. While this can result in a loss of precision, the implementation is simple and efficient. Proposing an enhanced field-sensitive taint analysis for C is out of the scope of this paper, and it is left as future work. As Mopsa performs dynamic expression rewriting to encourage a design based on layered semantics, to retrieve sources of tainted data, during the analysis we have to consider the expressions' rewriting history.

Our analysis can detect a wide variety of runtime errors, including double free, index-out-of-bounds, and null pointer dereference. While the formal presentation in this article, for the sake of simplicity, only supports division-by-zero errors, it was trivial to adapt our analysis to identify different types of failures. The complete list of errors detected by our tool is reported in Appendix E. In the report of the analyzer each warning is classified as possibly exploitable or not, and we infer a sound overapproximation of both the regular runtime errors and the exploitable ones. All the warnings that are not labelled as exploitable are thus *proved* to be nonexploitable. If the analyzer does not report *any* exploitable warning, then this is a proof that the program is nonexploitable.

The functions that read data from the user are part of the C standard library. They include, for instance, `getchar`, `scanf`, and `recv`. Mopsa provides a stub modeling language to specify the behaviour of library functions [49]. We have extended this language to support the fact that some functions generate tainted

data, and then we annotated our stubs for the C standard library to take into account the taint information. This model makes it trivial to update the list of dangerous sources, and the user of the analyzer does not have to annotate the source code to run the exploitability analysis, which is *fully automatic*. The complete list of functions that generate tainted data is reported in Appendix E.

Performance and Precision Evaluation. To assess the usefulness of our tool, we have analyzed real-world C programs from the GNU Coreutils package, which is a collection of command-line utilities. The test suite is composed of 77 programs that are long up to 4,188 lines of code. To them, we added a large set of short C programs taken from the Juliet test suite developed by NIST [9]. These programs contain examples of various runtime errors that can trigger well-known security vulnerabilities. In fact, Juliet is based on the CVE database [1], which enumerates vulnerabilities and focusses on security. The tested runtime errors include double frees, index out-of-bounds, and null pointer dereferences. The test cases are specifically designed to assess the precision of static analysis tools, and use a large set of features from the C standard. For Juliet, we considered 13,261 different test cases that amount to a total of 2,861,980 lines of code. Each test case comes with two versions: one that triggers a runtime failure, and one where the error is fixed. We run our analysis on both versions. An artifact to reproduce our experimental evaluation is available on Zenodo [50].

We compare the performance and number of alarms between Mopsa-Nexp and Mopsa. The analyses are parametric in the underlying abstract numeric domain, and we consider intervals, octagons [43], and polyhedra. Observe that to compare only the number of alarms raised by the two analyzers it is not necessary to run both tools, as Mopsa-Nexp can report all warnings raised by Mopsa. Notice that while the ground truth about the errors provided with Juliet can be used to evaluate the precision of a classic safety analysis, this is not the case for nonexploitability. In fact, the benchmarks categorize the test cases as either dangerous or not, but they do not include any information about whether an attacker can trigger the errors. We ran our experiments on a server with 128GB of RAM, with 48 Intel Xeon CPUs E5-2650 v4 @ 2.20GHz and Ubuntu 18.04.5 LTS. In Table 1 we report the results of our experiments.

For Coreutils, in the case of intervals, our analysis was able to *prove* that 3,498 over 4,715 runtime errors previously reported by the analyzer cannot be triggered by an attacker. For octagons and polyhedra, our analysis proved that respectively 3,464 and 3,458 potential runtime errors over 4,673 and 4,651 are not exploitable. Overall, this results in filtering out 74.13%-74.35% of the warnings. We found similar results for Juliet, where Mopsa-Nexp was able to prove that 71.75%-72.16% of the warnings are not exploitable. For Coreutils, Mopsa-Nexp raises 1,193 to $1,217$ warnings, which are those that can be potentially triggered by an attacker. The user of the analyzer could prioritize those alarms over the regular ones, as they are comparatively more dangerous.

The exploitability analysis incurs a performance overhead ranging from 13.89% to 15.05% for Coreutils and 2.4% to 3.5% for Juliet. During the analysis we con-

Table 1: Evaluation results.

Test suite	Domain	Analyzer	Alarms	Time
Coreutils	Intervals	Mopsa	4,715	1:17:06
		Mopsa-Nexp	1,217	1:28:42
	Octagons	Mopsa	4,673	2:22:29
		Mopsa-Nexp	1,209	2:43:06
	Polyhedra	Mopsa	4,651	2:12:21
		Mopsa-Nexp	1,193	2:30:44
Juliet	Intervals	Mopsa	49,957	11:32:24
		Mopsa-Nexp	13,906	11:48:51
	Octagons	Mopsa	48,256	13:15:29
		Mopsa-Nexp	13,631	13:41:47
	Polyhedra	Mopsa	48,256	12:54:21
		Mopsa-Nexp	13,631	13:21:26

sider expressions' rewriting history to preserve taint information, and this history is sensibly larger in real-world programs, which justifies the performance overhead difference between Coreutils and Juliet. Observe that we found octagons to be less efficient than polyhedra. This is due to the fact that Mopsa relies on the Apron [37] library, which uses a sparse representation for polyhedra, and can be very efficient if the number of variables is low and there are few constraints. As octagons use a dense representation, even if their algorithmic complexity is better, they are slightly slower for our case.

Discussion. We observed that Mopsa-Nexp is able to consistently filter out more than 70% of the warnings raised by the regular analyzer, while imposing low performance overhead. The Juliet test cases show that Mopsa-Nexp can handle almost the whole C specification, while the Coreutils experiments confirm that our analysis is effective even for real-world programs. The significant advantage of being able to classify each warning as security-critical or not outweighs the reasonable performance cost overhead. Observe that the alarms raised by Mopsa-Nexp are a subset of those reported by Mopsa. This implies that the exploitability analysis is, in the worst case, as precise as the regular analysis.

While it would be desirable to determine how many truly exploitable alarms are raised by Mopsa-Nexp, this cannot be done automatically. In fact, there is no ground truth that classifies program errors as nonexploitable or not, so that human inspection is the only option. In future work, we would like to conduct such an inspection.

8 Related work

Secure information flow. In [28] the authors propose the first mechanism to verify the secure flow of information in a program, namely checking that a program cannot cause supposedly nonconfidential results to depend on confidential

input data. Their formulation of the problem is based on the syntax of a program, and does not take into account its semantics. The concept of secure information flow is related to *noninterference* [22,32,33], which is a semantic definition. A program is *noninterferent* if its public output data does not depend on private input data. Checking that a program cannot cause nonconfidential results to depend on confidential input data has been widely studied through type systems [47,46,60,35,55,61,11,53,62,16,52]. Because these works perform only syntactic checks without taking into account semantic information, the results are generally very imprecise. In contrast, our approach tracks the flow of data generated by the user through a semantic taint analysis, which achieves enhanced precision by leveraging an overapproximation of the values of the variables.

Noninterference and nonexploitabilty are closely related: nonexploitability can be seen as a type of noninterference where the only public output variable is `ret`. Nevertheless, we do not rely on the static partitioning of variables into public and private, as our definition supports dynamic user input reads. Our framework can be used to prove noninterference: it is sufficient to read all private input variables at the beginning of the program, and then verify that the public output variables are not tainted. On the contrary, traditional methods to prove noninterference cannot prove nonexploitability, as they do not take the values of the variables into account.

Hyperproperties verification. Clarkson and Schneider [21] put forward the framework of *hyperproperties*, namely program properties that relate different sets of executions. Hyperproperties are able to express security policies, such as secure information flow. K-hypersafety properties [21] can be verified with traditional techniques for safety properties on the k-times self-composed system [18,57], even though this can be computationally expensive [12]. HyperLTL and HyperCTL/CTL* [30,20] define extensions of temporal logic able to quantify over multiple traces to address the verification of hyperproperties.

Noninterference verification by abstract interpretation. Cousot [24] put forward a semantic definition of dependencies in the abstract interpretation framework. He proposes a sound analysis of dependencies, capable of proving noninterference. Similarly to us, he does not rely on hypersemantics, using standard abstract interpretation techniques. Nevertheless, the abstract dependency semantics is not structural (i.e. defined by induction on the program syntax), as it does not take the values of variables into account. The author proposes leveraging the values of the variables to give a structural definition of the semantics, and this paper attempts to implement such an extension. Since his definition of the dependency semantics does not take into account the values of the variables, it is not possible to define an analysis that leverages numeric abstract domains to enhance the precision of the dependency analysis. Another significant difference is that the dependencies are relative to the *initial values* of variables. Our analysis computes *dynamic tainting*, which is the dependency of a variable from

any input statement (including those within conditionals and loops), so that it generalizes the dependency analysis from the beginning of the program.

There are numerous papers that use an alternative version of the abstract interpretation framework based on *hypersemantics* [14,40,41,42,59], where the concrete domain is a set of sets of states, rather than a set of states. This is to overcome the difficulties related to the fact that not every hyperproperty is subset-closed, and classic overapproximation techniques seem to fail. However, as argued in this paper and [24], this is not the case for standard noninterference and nonexploitability. Relying on the classic abstract interpretation framework allows using the large library of existing abstract domains, and leveraging the semantic information inferred by such domains is not only essential for nonexploitabilty, but also enhances the precision of the taint analysis. Another approach to noninterference verification is introduced in [58], where the authors combine abstract interpretation with symbolic execution to define a sound analysis.

Infer [7], Pysa [8], and Julia [56] are static analyzers based on abstract interpretation that support taint analysis. All these tools do not detect *implicit flows*, being effectively unsound in our framework. The analyzers can track taint information, but they cannot classify runtime errors as exploitable or not. To the best of our knowledge, Mopsa-Nexp is the first analyzer to have such capability.

Errors classification. In [31] the authors put forward the concept of *robust reachability*. A runtime error is robustly reachable if a controlled input can make it so the bug is reached whatever the value of uncontrolled input. The authors use symbolic execution and bounded model checking techniques to find robustly reachable bugs. Similarly to this paper, [31] classifies runtime failures by their dangerousness and filters out less interesting alarms that do not concern security issues. Nevertheless, the concept of robustly reachable runtime error is different from nonexploitability: a bug is considered robustly reachable even if it is triggered *for all* possible user input, while such an error is not exploitable according to our formal definition of exploitability. In fact, we require the user input to be actually involved in triggering an error to consider a program exploitable.

Other techniques relying on probability theory to differentiate classes of bugs have been proposed. They include *probabilistic model checking* [15,34], *probabilistic abstract interpretation* [27,44,45,51], *quantitative robust reachability* [17], and *quantitative information flow analysis* [36]. An interesting extension of this paper would be to use ideas from these approaches to put forward a *quantitative* exploitability analysis to classify more finely the level of threat caused by alarms.

9 Conclusions

In this paper, we introduced the novel definition of nonexploitability, which we leveraged to put forward a sound analysis by abstract interpretation. The framework supports constructs that are essential to analyze real-world programs, such as nondeterminism and dynamic user input reads. Our analysis performs a semantic taint analysis that achieves superior precision through a modular reduc-

tion with existing numeric abstract domains. The theoretical framework bridges the gap between traditional safety properties and security hyperproperties, and our analysis can rule out the existence of exploitable runtime errors in programs.

We implemented our analysis in the Mopsa-Nexp tool, the first analyzer dedicated to nonexploitability. The tool is fully automatic, and to assess its effectiveness, we evaluated it on a large set of real-world C programs. The analyzer can consistently *prove* that more than 70% of the previously raised warnings cannot be triggered by an attacker, all while incurring less than 16% performance overhead. While usually the number of false positives is lowered by increasing the precision of the abstract domains, we take an orthogonal approach by reporting only the alarms that can be triggered by an attacker. By leveraging the fundamental observation that security-related warnings are more dangerous than the others, our technique dramatically reduces the noise generated by false alarms, enhancing the usefulness of the analyzer.

In future work, we would like to extend our analysis to prove the absence of other classes of exploitable bugs. A promising path forward is to leverage probability theory to perform a *quantitative* exploitability analysis capable of further reducing the number of alarms. Another interesting extension of this paper is to adapt our framework to rule out the existence of exploitable liveness errors, such as exploitable deadlocks in multithreaded programs.

Acknowledgments We would like to thank the anonymous reviewers for their comments. This work was supported by the SECURVAL project. The SECUREVAL project was funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference ANR-22-PECY-0005.

References

1. Common vulnerabilities and exposures (CVE) database, https://cve.mitre.org/, accessed: 2023-08-30
2. CVE-2016-7869. Available from NIST, CVE-ID CVE-2016-7869., https://nvd.nist.gov/vuln/detail/CVE-2016-7869, accessed: 2023-08-30
3. CVE-2019-5699. Available from NIST, CVE-ID CVE-2019-5699., https://nvd.nist.gov/vuln/detail/CVE-2019-5699, accessed: 2023-08-30
4. CVE-2019-8745. Available from NIST, CVE-ID CVE-2019-8745., https://nvd.nist.gov/vuln/detail/CVE-2019-8745, accessed: 2023-08-30
5. CVE-2022-36934. Available from NIST, CVE-ID CVE-2022-36934., https://nvd.nist.gov/vuln/detail/CVE-2022-36934, accessed: 2023-08-30
6. CVE-2022-4135. Available from NIST, CVE-ID CVE-2022-4135., https://nvd.nist.gov/vuln/detail/CVE-2022-4135, accessed: 2023-08-30
7. The Infer static analyzer, https://fbinfer.com/
8. The Pysa static analyzer, https://engineering.fb.com/2020/08/07/security/pysa/
9. Juliet C/C++ test suite (2017), https://samate.nist.gov/SARD/test-suites/112, accessed: 2023-08-30
10. Microsoft: A proactive approach to more secure code (2019), https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/, accessed: 2023-08-30

11. Agat, J.: Transforming out timing leaks. In: Principles of Programming Languages, POPL. pp. 40–53. ACM (2000). https://doi.org/10.1145/325694.325702
12. Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. In: Conference on Programming Language Design and Implementation, PLDI. pp. 362–375. ACM (2017). https://doi.org/10.1145/3062341.3062378
13. Arzt, S., Rasthofer, S., Fritz, C., Bodden, E., Bartel, A., Klein, J., Traon, Y.L., Octeau, D., McDaniel, P.D.: Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In: Programming Language Design and Implementation, PLDI. pp. 259–269. ACM (2014). https://doi.org/10.1145/2594291.2594299
14. Assaf, M., Naumann, D.A., Signoles, J., Totel, E., Tronel, F.: Hypercollecting semantics and its application to static analysis of information flow. In: Principles of Programming Languages, POPL (2017). https://doi.org/10.1145/3009837.3009889
15. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Verifying continuous time markov chains. In: Computer Aided Verification, CAV. Lecture Notes in Computer Science, vol. 1102, pp. 269–276. Springer (1996). https://doi.org/10.1007/3-540-61474-5_75
16. Banerjee, A., Naumann, D.A.: Secure information flow and pointer confinement in a java-like language. In: Computer Security Foundations Workshop CSFW. p. 253. IEEE Computer Society (2002). https://doi.org/10.1109/CSFW.2002.1021820
17. Bardin, S., Girol, G.: A quantitative flavour of robust reachability. CoRR **abs/2212.05244** (2022). https://doi.org/10.48550/arXiv.2212.05244
18. Barthe, G., D'Argenio, P.R., Rezk, T.: Secure information flow by self-composition. Mathematical Structures in Computer Science **21**(6), 1207–1252 (2011). https://doi.org/10.1017/S0960129511000193
19. Berghel, H.: The code red worm. Commun. ACM **44**(12), 15–19 (2001). https://doi.org/10.1145/501317.501328
20. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.: Temporal logics for hyperproperties. In: Principles of Security and Trust, POST. Lecture Notes in Computer Science, vol. 8414, pp. 265–284. Springer (2014). https://doi.org/10.1007/978-3-642-54792-8_15
21. Clarkson, M.R., Schneider, F.B.: Hyperproperties. 21st IEEE Computer Security Foundations Symposium pp. 51–65 (2008)
22. Cohen, E.S.: Information transmission in computational systems. In: Symposium on Operating System Principles, SOSP. pp. 133–139. ACM (1977). https://doi.org/10.1145/800214.806556
23. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The Astrée analyzer. In: European Symposium on Programming ESOP. Lecture Notes in Computer Science (LNCS), vol. 3444, pp. 21–30. Springer (2005). https://doi.org/10.1007/978-3-540-31987-0_3, http://www-apr.lip6.fr/~mine/publi/esop05_astree.pdf
24. Cousot, P.: Abstract semantic dependency. In: Static Analysis Symposium, SAS. vol. 11822, pp. 389–410. Springer (2019). https://doi.org/10.1007/978-3-030-32304-2_19
25. Cousot, P.: Principles of Abstract Interpretation. The MIT Press (2022), https://mitpress.mit.edu/9780262044905/principles-of-abstract-interpretation
26. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Principles of Programming Languages, POPL (1977)

27. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: European Symposium on Programming, ESOP. Lecture Notes in Computer Science, vol. 7211, pp. 169–193. Springer (2012). https://doi.org/10.1007/978-3-642-28869-2_9
28. Denning, D.E., Denning, P.J.: Certification of programs for secure information flow. Commun. ACM **20**(7), 504–513 (1977). https://doi.org/10.1145/359636.359712
29. Durumeric, Z., Kasten, J., Adrian, D., Halderman, J.A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., Paxson, V.: The matter of heartbleed. In: Internet Measurement Conference, IMC. pp. 475–488. ACM (2014). https://doi.org/10.1145/2663716.2663755
30. Finkbeiner, B., Rabe, M.N., Sánchez, C.: Algorithms for model checking hyperltl and hyperctl ^*. In: Computer Aided Verification, CAV. Lecture Notes in Computer Science, vol. 9206, pp. 30–48. Springer (2015). https://doi.org/10.1007/978-3-319-21690-4_3
31. Girol, G., Farinier, B., Bardin, S.: Not all bugs are created equal, but robust reachability can tell the difference. In: Computer Aided Verification, CAV. vol. 12759, pp. 669–693. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_32
32. Goguen, J.A., Meseguer, J.: Security policies and security models. In: Security and Privacy. pp. 11–20. IEEE Computer Society (1982). https://doi.org/10.1109/SP.1982.10014
33. Goguen, J.A., Meseguer, J.: Unwinding and inference control. In: Security and Privacy. pp. 75–87. IEEE Computer Society (1984). https://doi.org/10.1109/SP.1984.10019
34. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects Comput. **6**(5), 512–535 (1994). https://doi.org/10.1007/BF01211866
35. Heintze, N., Riecke, J.G.: The slam calculus: Programming with secrecy and integrity. In: Principles of Programming Languages, POPL. pp. 365–377. ACM (1998). https://doi.org/10.1145/268946.268976
36. Heusser, J., Malacaria, P.: Quantifying information leaks in software. In: Annual Computer Security Applications Conference, ACSAC. pp. 261–269. ACM (2010). https://doi.org/10.1145/1920261.1920300
37. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Computer Aided Verification, CAV. Lecture Notes in Computer Science (LNCS), vol. 5643, pp. 661–667. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_52, http://www-apr.lip6.fr/~mine/publi/article-mine-jeannet-cav09.pdf
38. Journault, M., Miné, A., Monat, R., Ouadjaout, A.: Combinations of reusable abstract domains for a multilingual static analyzer. In: Proc. of the 11th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE19). Lecture Notes in Computer Science (LNCS), vol. 12031, pp. 1–18. Springer (2019). https://doi.org/10.1007/978-3-030-41600-3_1, http://www-apr.lip6.fr/~mine/publi/article-mine-al-vstte19.pdf
39. Li, L., Bissyandé, T.F., Papadakis, M., Rasthofer, S., Bartel, A., Octeau, D., Klein, J., Traon, Y.L.: Static analysis of android apps: a systematic literature review. Inf. Softw. Technol. **88**, 67–95 (2017). https://doi.org/10.1016/j.infsof.2017.04.001
40. Mastroeni, I., Pasqua, M.: Hyperhierarchy of semantics - A formal framework for hyperproperties verification. In: Static Analysis Symposium, SAS. vol. 10422, pp. 232–252 (2017). https://doi.org/10.1007/978-3-319-66706-5_12
41. Mastroeni, I., Pasqua, M.: Verifying bounded subset-closed hyperproperties. In: Static Analysis Symposium, SAS. vol. 11002, pp. 263–283 (2018). https://doi.org/10.1007/978-3-319-99725-4_17

42. Mastroeni, I., Pasqua, M.: Statically analyzing information flows: an abstract interpretation-based hyperanalysis for non-interference. In: Symposium on Applied Computing, SAC. pp. 2215–2223 (2019). https://doi.org/10.1145/3297280.3297498
43. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation (HOSC) **19**(1), 31–100 (2006). https://doi.org/10.1007/s10990-006-8609-1, http://www-apr.lip6.fr/~mine/publi/article-mine-HOSC06.pdf
44. Monniaux, D.: Abstract interpretation of probabilistic semantics. In: Static Analysis Symposium, SAS. Lecture Notes in Computer Science, vol. 1824, pp. 322–339. Springer (2000). https://doi.org/10.1007/978-3-540-45099-3_17
45. Monniaux, D.: An abstract analysis of the probabilistic termination of programs. In: Static Analysis Symposium, SAS. Lecture Notes in Computer Science, vol. 2126, pp. 111–126. Springer (2001). https://doi.org/10.1007/3-540-47764-0_7
46. Myers, A.C., Liskov, B.: A decentralized model for information flow control. In: Symposium on Operating System Principles, SOSP. pp. 129–142. ACM (1997). https://doi.org/10.1145/268998.266669
47. Ørbæk, P., Palsberg, J.: Trust in the lambda-calculus. J. Funct. Program. **7**(6), 557–591 (1997). https://doi.org/10.1017/s0956796897002906
48. Orman, H.K.: The morris worm: A fifteen-year perspective. IEEE Secur. Priv. **1**(5), 35–43 (2003). https://doi.org/10.1109/MSECP.2003.1236233
49. Ouadjaout, A., Miné, A.: A library modeling language for the static analysis of C programs. In: Static Analysis Symposium, SAS. Lecture Notes in Computer Science (LNCS), vol. 12389, pp. 223–246. Springer (2020). https://doi.org/10.1007/978-3-030-65474-0_11, http://www-apr.lip6.fr/~mine/publi/ouadjaout-al-sas20.pdf
50. Parolini, F., Miné, A.: Sound Abstract Nonexploitability Analysis Artifact (Sep 2023). https://doi.org/10.5281/zenodo.8334112
51. Pierro, A.D., Wiklicky, H.: Probabilistic abstract interpretation: From trace semantics to dtmc's and linear regression. In: Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays. Lecture Notes in Computer Science, vol. 9560, pp. 111–139. Springer (2016). https://doi.org/10.1007/978-3-319-27810-0_6
52. Pottier, F., Simonet, V.: Information flow inference for ML. ACM Trans. Program. Lang. Syst. **25**(1), 117–158 (2003). https://doi.org/10.1145/596980.596983
53. Sabelfeld, A., Sands, D.: Probabilistic noninterference for multi-threaded programs. In: Computer Security Foundations Workshop, CSFW. pp. 200–214. IEEE Computer Society (2000). https://doi.org/10.1109/CSFW.2000.856937
54. Schultz, E., Mellander, J., Peterson, D.: The MS-SQL slammer worm. Network Security **2003**(3), 10–14 (2003). https://doi.org/https://doi.org/10.1016/S1353-4858(03)00310-6
55. Smith, G., Volpano, D.M.: Secure information flow in a multi-threaded imperative language. In: Principles of Programming Languages, POPL. pp. 355–364. ACM (1998). https://doi.org/10.1145/268946.268975
56. Spoto, F., Burato, E., Ernst, M.D., Ferrara, P., Lovato, A., Macedonio, D., Spiridon, C.: Static identification of injection attacks in java. ACM Trans. Program. Lang. Syst. **41**(3), 18:1–18:58 (2019). https://doi.org/10.1145/3332371
57. Terauchi, T., Aiken, A.: Secure information flow as a safety problem. In: Static Analysis Symposium, SAS. Lecture Notes in Computer Science, vol. 3672, pp. 352–367 (2005). https://doi.org/10.1007/11547662_24
58. Tiraboschi, I., Rezk, T., Rival, X.: Sound symbolic execution via abstract interpretation and its application to security. In: Verification, Model Checking, and

Abstract Interpretation, VMCAI. Lecture Notes in Computer Science, vol. 13881, pp. 267–295. Springer (2023). https://doi.org/10.1007/978-3-031-24950-1_13

59. Urban, C., Müller, P.: An abstract interpretation framework for input data usage. In: European Symposium on Programming, ESOP. vol. 10801, pp. 683–710 (2018). https://doi.org/10.1007/978-3-319-89884-1_24
60. Volpano, D.M., Irvine, C.E., Smith, G.: A sound type system for secure flow analysis. Journal of Computer Security **4**(2/3), 167–188 (1996). https://doi.org/10.3233/JCS-1996-42-304
61. Volpano, D.M., Smith, G.: Probabilistic noninterference in a concurrent language. J. Comput. Secur. **7**(1) (1999). https://doi.org/10.3233/jcs-1999-72-305, https://doi.org/10.3233/jcs-1999-72-305
62. Zdancewic, S., Myers, A.C.: Secure information flow and CPS. In: European Symposium on Programming, ESOP. Lecture Notes in Computer Science, vol. 2028, pp. 46–61. Springer (2001). https://doi.org/10.1007/3-540-45309-1_4

A Expression Semantics

$$\begin{aligned}
\mathcal{A}[\![\mathtt{A}]\!] &: \mathbb{M} \to \mathbb{Z}_{\lightning} \\
\mathcal{A}[\![n]\!]m &\triangleq n \\
\mathcal{A}[\![\mathtt{x}]\!]m &\triangleq m[\mathtt{x}] \\
\mathcal{A}[\![\mathtt{A}_1 \diamond \mathtt{A}_2]\!]m &\triangleq \begin{cases} \lightning & \text{if } \mathcal{A}[\![\mathtt{A}_1]\!]m = \lightning, \text{ or } \mathcal{A}[\![\mathtt{A}_2]\!]m = \lightning, \\ & \text{or } \diamond = / \text{ and } \mathcal{A}[\![\mathtt{A}_2]\!]m = 0 \\ \mathcal{A}[\![\mathtt{A}_1]\!]m \diamond \mathcal{A}[\![\mathtt{A}_2]\!]m & \text{otherwise} \end{cases}
\end{aligned}$$

$$\begin{aligned}
\mathcal{B}[\![\mathtt{B}]\!] &: \mathbb{M} \to \mathbb{B}_{\lightning} \\
\mathcal{B}[\![\mathtt{tt}]\!]m &\triangleq \mathtt{tt} \\
\mathcal{B}[\![\mathtt{ff}]\!]m &\triangleq \mathtt{ff} \\
\mathcal{B}[\![\mathtt{A}_1 < \mathtt{A}_2]\!]m &\triangleq \begin{cases} \lightning & \text{if } \mathcal{A}[\![\mathtt{A}_1]\!]m = \lightning \text{ or } \mathcal{A}[\![\mathtt{A}_2]\!]m = \lightning \\ \mathcal{A}[\![\mathtt{A}_1]\!]m < \mathcal{A}[\![\mathtt{A}_2]\!]m & \text{otherwise} \end{cases} \\
\mathcal{B}[\![\mathtt{B}_1 \diamond \mathtt{B}_2]\!]m &\triangleq \begin{cases} \lightning & \text{if } \mathcal{B}[\![\mathtt{B}_1]\!]m = \lightning \text{ or } \mathcal{B}[\![\mathtt{B}_2]\!]m = \lightning \\ \mathcal{B}[\![\mathtt{B}_1]\!]m \diamond \mathcal{B}[\![\mathtt{B}_2]\!]m & \text{otherwise} \end{cases} \\
\mathcal{B}[\![\neg\mathtt{B}_1]\!]m &\triangleq \begin{cases} \lightning & \text{if } \mathcal{B}[\![\mathtt{B}_1]\!]m = \lightning \\ \neg\mathcal{B}[\![\mathtt{B}_1]\!]m & \text{otherwise} \end{cases}
\end{aligned}$$

B Proofs

Proof *($(\wp(\mathbb{D}), \subseteq) \underset{\alpha_t}{\overset{\gamma_t}{\leftrightarrows}} (\wp(\mathbb{V}), \supseteq)$).* Let $\mathscr{R} \in \wp(\mathbb{D})$ and $\mathcal{T} \in \wp(\mathbb{V})$.

$$\begin{aligned}
\alpha_t(\mathscr{R}) \supseteq \mathcal{T} &\iff \mathcal{T} \subseteq \alpha_t(\mathscr{R}) \\
&\iff \mathcal{T} \subseteq \{\, \mathtt{x} \in \mathbb{V} \mid \mathscr{R} \subseteq \mathscr{T}(\mathtt{x}) \,\} \\
&\iff \forall \mathtt{x} \in \mathcal{T} : \mathscr{R} \subseteq \mathscr{T}(\mathtt{x}) \\
&\iff \forall \mathtt{x} \in \mathcal{T} : \forall \mathcal{R} \in \mathscr{R} : \mathcal{R} \in \mathscr{T}(\mathtt{x}) \\
&\iff \forall \mathcal{R} \in \mathscr{R} : \forall \mathtt{x} \in \mathcal{T} : \mathcal{R} \in \mathscr{T}(\mathtt{x}) \\
&\iff \mathscr{R} \subseteq \{\, \mathcal{R} \mid \forall \mathtt{x} \in \mathcal{T} : \mathcal{R} \in \mathscr{T}(\mathtt{x}) \,\} \\
&\iff \mathscr{R} \subseteq \bigcap_{\mathtt{x} \in \mathcal{T}} \mathscr{T}(\mathtt{x}) \\
&\iff \mathscr{R} \subseteq \gamma_t(\mathcal{T})
\end{aligned}$$

Before proving Equation (1), we give an alternative characterization of $\mathscr{NE}$.

Theorem 3.

$$\mathscr{NE} = \{\, \mathcal{R} \in \mathbb{D} \mid \mathtt{ret} \notin \alpha_t(\{\, \mathcal{R} \,\}) \,\}$$

Proof.

$$\begin{aligned}
&\mathscr{NE} \\
&= \{\mathcal{R} \in \mathbb{D} \mid \forall ((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in \mathcal{R} : \\
&\qquad\qquad m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 \implies m_1[\mathtt{ret}] = m'_1[\mathtt{ret}]\} \\
&\qquad\qquad \text{(definition of } \mathscr{NE}) \\
&= \{\mathcal{R} \in \mathbb{D} \mid \nexists ((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in \mathcal{R} : \\
&\qquad\qquad m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 : m_1[\mathtt{ret}] \neq m'_1[\mathtt{ret}]\} \qquad \text{(negation of } \forall) \\
&= \{\mathcal{R} \in \mathbb{D} \mid \mathtt{ret} \in \overline{\alpha_t(\{\mathcal{R}\})}\} \qquad \text{(definition of } \alpha_t) \\
&= \{\mathcal{R} \in \mathbb{D} \mid \mathtt{ret} \notin \alpha_t(\{\mathcal{R}\})\} \qquad (\alpha_t \text{ defines a partition over } \mathbb{V})
\end{aligned}$$

Proof (Equation (1)). Follows immediately from Theorem 3.

Proof (Equation (2)).

$$
\begin{aligned}
&\alpha_t(\{\mathcal{R}_0\}) \\
&= \{\, \mathtt{x} \in \mathbb{V} \mid \{\mathcal{R}_0\} \subseteq \mathscr{T}(\mathtt{x}) \,\} \\
&= \{\, \mathtt{x} \in \mathbb{V} \mid \mathcal{R}_0 \in \mathscr{T}(\mathtt{x}) \,\} \\
&= \{\, \mathtt{x} \in \mathbb{V} \mid \mathcal{R}_0 \in \{\, \mathcal{R} \in \mathbb{D} \mid \\
&\qquad\qquad \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in \mathcal{R} : \\
&\qquad\qquad\quad m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 : m_1[\mathtt{x}] \neq m'_1[\mathtt{x}] \,\}\,\} \qquad \text{(definition of } \mathscr{T}(\mathtt{x})) \\
&= \{\, \mathtt{x} \in \mathbb{V} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in \mathcal{R}_0 : \\
&\qquad\qquad\quad m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 : m_1[\mathtt{x}] \neq m'_1[\mathtt{x}] \,\} \\
&\subseteq \{\, \mathtt{x} \in \mathbb{V} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in \mathcal{R}_1 : \\
&\qquad\qquad\quad m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 : m_1[\mathtt{x}] \neq m'_1[\mathtt{x}] \,\} \qquad (\mathcal{R}_0 \subseteq \mathcal{R}_1) \\
&= \alpha_t(\{\mathcal{R}_1\})
\end{aligned}
$$

Proof (Theorem 1).

$$
\begin{aligned}
\mathcal{R}_1 \in \mathscr{NE} &\iff \mathtt{ret} \notin \{\, \mathtt{x} \mid \mathcal{R}_1 \in \mathscr{T}(\mathtt{x}) \,\} && \text{(Equation (1))} \\
&\implies \mathtt{ret} \notin \{\, \mathtt{x} \mid \mathcal{R}_0 \in \mathscr{T}(\mathtt{x}) \,\} && (\mathcal{R}_0 \subseteq \mathcal{R}_1 \text{ and Equation (2)}) \\
&\iff \mathcal{R}_0 \in \mathscr{NE} && \text{(Equation (1))}
\end{aligned}
$$

Before proving Theorem 2, we have to prove the following result. Theorem 4 states that if the argument $\mathcal{T}$ of the concrete structural semantics is the set of truly tainted variables in $\mathcal{R}$, then $\hat{\mathcal{S}}_t[\![\mathtt{S}]\!]$ exactly corresponds to $\mathcal{S}_t[\![\mathtt{S}]\!]$.

Theorem 4 (Correctness of $\hat{\mathcal{S}}_t[\![\mathtt{S}]\!]$). *Let $\mathcal{R}, \mathcal{E} \in \mathbb{D}$ and $\mathcal{T} \in \wp(\mathbb{V})$.*

$$\mathcal{T} = \alpha_t(\{\, \mathcal{R} \,\}) \implies \mathcal{S}_t[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}) \doteq \hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}, \mathcal{T})$$

Proof (Theorem 4). By structural induction. For the definition of the inductive concrete taint semantics, see Appendix C. Some cases are trivial, and we do not report them. The correctness of $\mathtt{x} \leftarrow \mathtt{input()}$ follows from the fact that the variable $\mathtt{x}$ is the only one that assumes a different value, and hence can become tainted after the execution of the statement. For this reason, $\{\, \mathtt{y} \in \mathcal{T} \mid \mathtt{y} \neq \mathtt{x} \,\}$ exactly corresponds the set of tainted variables after the execution of the statement using the hypothesis $\mathcal{T} = \alpha_t(\{\, \mathcal{R} \,\})$. Using the definition of α_t, we can observe that $\mathtt{x}$ is tainted if and only if the first element in the sequence of input can be controlled by the user, namely there are two executions whose initial states differ only in the user input and end in different first values for the input sequence. Random read statements are analogous.

Assignments are similar. Again, the only variable that can assume a different value is $\mathtt{x}$, so that it is the only one that can become possibly tainted. For this reason, $\{\, \mathtt{y} \in \mathcal{T} \mid \mathtt{y} \neq \mathtt{x} \,\}$ exactly corresponds to the set of other tainted variables after the execution of the statement using the hypothesis $\mathcal{T} = \alpha_t(\{\, \mathcal{R} \,\})$. Using the definition of α_t, we observe that $\mathtt{x}$ is tainted iff there are two executions that differ in the initial states only in the input sequence, and result in different (non-error) values for the arithmetic evaluation in the input memory. This exactly corresponds to our definition.

By using α_t, we can observe that the tainted variables after the execution of if statements are: 1) those tainted in the then branch; 2) those tainted in the else

branch; 3) those that assume different values in the two branches, in case which one of the two branches is executed depends on the user input. The first two cases are simple, and follow by inductive hypothesis. The third is covered by the definition of $\texttt{diff}[\![\texttt{if (B) S}_t \texttt{ else S}_e]\!]$, as it considers all possible pairs of execution that in the initial state differ only in the user input, explore different branches, and then result in different values for some variables. For while statements it is sufficient to observe that $\lambda(\mathcal{R}_1, \mathcal{E}_1, \mathcal{T}_1) \,.\, (\mathcal{R}, \mathcal{E}, \mathcal{T}) \dot\cup \hat{\mathcal{S}}_t[\![\texttt{if (B) S}_b \texttt{ else skip}]\!](\mathcal{R}_1, \mathcal{E}_1, \mathcal{T}_1)$ is monotonic, and by using the correctness for if statements, it is possible to prove the correctness of while statements.

Proof (Theorem 2). We observe that there are no tainted variables in the set of initial states $\mathcal{I}$. Then, the theorem follows immediately from Theorem 4.

Proof (Equation (3)*).* By structural induction. The soundness of the abstract taint semantics is proved with respect the structural concrete taint semantics, which is reported in Appendix C. Some cases are trivial, and we do not report them. For input read statements, it is sound to always taint the variable `x`, which is the only variable that possibly assumes a different value.

For random read statements, the only variable that can assume a different value is `x`, so that `x` is the only variable that can become tainted after the execution of $\texttt{x} \leftarrow \texttt{rand()}$. By considering the definition of $\hat{\mathcal{S}}_t[\![\texttt{x} \leftarrow \texttt{rand()}]\!]$, we observe that this happens only if the user can control which number in the sequence of random numbers is read. The soundness then follows by inductive hypothesis observing that if the user can control the value of the index variable `i`, then `i` is tainted. If `i` is tainted, then we taint `x`.

Similarly to the previous cases, `x` is the only variable that can become tainted after the execution of assignment $\texttt{x} \leftarrow \texttt{A}$. Then, the soundness of the abstract semantics immediately follows from the soundness of $\texttt{taint}^\sharp[\![\texttt{A}]\!]$, which returns `ff` only if the result of the arithmetic evaluation is *definitely* not influenced by user input.

For if statements, the fact that the tainted variables computed in the branches are an overapproximation of the truly tainted ones follows by inductive hypothesis. Then, we observe that $\texttt{diff}^\sharp[\![\texttt{if (B) S}_t \texttt{ else S}_e]\!]$ is a sound overapproximation of $\texttt{diff}[\![\texttt{if (B) S}_t \texttt{ else S}_e]\!]$, namely $\forall \mathcal{R}^\sharp, \mathcal{E}^\sharp : \forall \mathcal{R} \in \gamma_d(\mathcal{R}^\sharp), \mathcal{E} \in \gamma_d(\mathcal{E}^\sharp) : \texttt{diff}[\![\texttt{if (B) S}_t \texttt{ else S}_e]\!](\mathcal{R}, \mathcal{E}) \subseteq \texttt{diff}^\sharp[\![\texttt{if (B) S}_t \texttt{ else S}_e]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp)$. If a variable `x` is in $\texttt{diff}[\![\texttt{if (B) S}_t \texttt{ else S}_e]\!]$, then the user can control the outcome of $\mathcal{B}[\![\texttt{B}]\!]$, and `x` is definitely assigned in at least one of the branches. We can conclude by using the soundness of $\texttt{assigned}^\sharp[\![\texttt{S}]\!]$ and $\texttt{taint}^\sharp[\![\texttt{B}]\!]$. Since while statements are defined in terms of if statements, the soundness of the former follows from the soundness of the latter.

The following result is useful to observe the connection between the $\mathcal{S}_t[\![\texttt{S}]\!]$ and $\mathcal{S}_t^\sharp[\![\texttt{S}]\!]$.

$$\alpha_t(\{\gamma_d(\mathcal{R}^\sharp)\}) \subseteq \mathcal{T}^\sharp \implies \mathcal{S}_t[\![\texttt{S}]\!](\gamma_d(\mathcal{R}^\sharp), \gamma_d(\mathcal{E}^\sharp)) \dot\subseteq \gamma(\mathcal{S}_t^\sharp[\![\texttt{S}]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathcal{T}^\sharp)) \tag{5}$$

Proof (Equation (5)*).*

$$\begin{aligned}
\mathcal{S}_t[\![\texttt{S}]\!](\gamma_d(\mathcal{R}^\sharp), \gamma_d(\mathcal{E}^\sharp)) &\doteq \hat{\mathcal{S}}_t[\![\texttt{S}]\!](\gamma_d(\mathcal{R}^\sharp), \gamma_d(\mathcal{E}^\sharp), \alpha_t(\{\gamma_d(\mathcal{R}^\sharp)\})) && \text{(Theorem 4)} \\
&\dot\subseteq \hat{\mathcal{S}}_t[\![\texttt{S}]\!](\gamma_d(\mathcal{R}^\sharp), \gamma_d(\mathcal{E}^\sharp), \mathcal{T}^\sharp) \\
&\qquad (\alpha_t(\{\gamma_d(\mathcal{R}^\sharp)\}) \subseteq \mathcal{T}^\sharp \text{ and monotonicity of } \hat{\mathcal{S}}_t[\![\texttt{S}]\!]) \\
&\dot\subseteq \gamma(\mathcal{S}_t^\sharp[\![\texttt{S}]\!](\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathcal{T}^\sharp)) && \text{(Equation (3))}
\end{aligned}$$

In order to prove Equation (4), we need to prove two intermediate results. We first define the abstract taint semantics for programs. Then, we define the abstract taint semantics for programs $\mathcal{S}_t^\sharp[\![\mathtt{P}]\!]$, where $\mathtt{P} \coloneqq \mathtt{S}$, as follows.

$$\mathcal{S}_t^\sharp[\![\mathtt{P}]\!] \triangleq \text{let } (\mathcal{R}^\sharp, \mathcal{E}^\sharp, \mathcal{T}^\sharp) = \mathcal{S}_t^\sharp[\![\mathtt{S}]\!](\mathcal{I}^\sharp, \bot_d^\sharp, \varnothing) \text{ in}$$
$$(\mathcal{R}^\sharp \cup_d^\sharp \mathcal{E}^\sharp, (\mathbb{V} \setminus \{\, \mathtt{ret} \,\}) \cup \mathcal{T}^\sharp)$$

At the end of the program we taint all regular program variables but `ret`. This is a sound overapproximation of the variables that are tainted due to runtime errors controlled by the user. For instance, consider the following program: `x ← input(); y ← 1; z ← 1/x; y ← 2`. Depending on the user input, the variable `y` can be either 1 or 2 at the end of the program when we merge errors states with the others. As these dependencies are not useful to prove nonexploitability, overapproximating them with all the non-return variables does not interfere with our objective.

The only variable that can be added from $\mathcal{T}^\sharp$ is then `ret`, which will be collected during the analysis. In fact, while in the concrete semantics `ret` can result tainted only at the end of the program, in our abstract semantics we taint `ret` every time there is a *possible* runtime error due to user input. This guarantees that if `ret` is not tainted in $\mathcal{S}_t^\sharp[\![\mathtt{S}]\!]$, it will not be tainted at the end of the program. Let $(\mathcal{R}^\sharp, \mathcal{T}^\sharp) = \mathcal{S}_t^\sharp[\![\mathtt{P}]\!]$. The abstract semantics for programs is *sound* with respect to the following.

$$\mathcal{S}_t[\![\mathtt{P}]\!] \dot{\subseteq} (\gamma_d(\mathcal{R}^\sharp), \mathcal{T}^\sharp) \tag{6}$$

Proof (Equation (6)). The soundness for the values follows from the soundness of the abstract numeric domain. For the tainted variables, tainting all program variables but `ret` overapproximates the set of truly tainted variables excluding `ret`. The fundamental observation to prove that if `ret` is tainted in the concrete semantics, then it is tainted in the abstract semantics is that if `ret` is tainted in the concrete, then there is *at least* one statement in which a runtime error occurs, and such error can be triggered by the user. Since the abstract semantics taints `ret` every time there is a *possible* runtime error that could be triggered by the user input, if `ret` is tainted in the concrete semantics, it will definitely be tainted in the abstract semantics.

Since $\mathcal{S}_t^\sharp[\![\mathtt{P}]\!]$ finds more tainted variables than the concrete semantics $\mathcal{S}_t[\![\mathtt{P}]\!]$, if $\mathcal{S}_t^\sharp[\![\mathtt{P}]\!] = (\mathcal{R}^\sharp, \mathcal{T}^\sharp)$ determines that `ret` is not tainted, `ret` is *definitely* not tainted in $\mathcal{S}_t[\![\mathtt{P}]\!]$. In other words, there is a proof that `P` is nonexploitable.

$$\mathtt{ret} \notin \mathcal{T}^\sharp \implies \mathcal{S}[\![\mathtt{P}]\!] \in \mathscr{NE} \tag{7}$$

Proof (Equation (7)). Let $(\mathcal{R}, \mathcal{T}) = \mathcal{S}_t[\![\mathtt{P}]\!]$.

$$\begin{aligned}
\mathtt{ret} \notin \mathcal{T}^\sharp &\implies \mathtt{ret} \notin \mathcal{T} && \text{(Equation (6))} \\
&\iff \mathtt{ret} \notin \alpha_t(\{\mathcal{S}[\![\mathtt{P}]\!]\}) && \text{(Definition of } \mathcal{S}_t[\![\mathtt{P}]\!]\text{)} \\
&\iff \mathcal{S}[\![\mathtt{P}]\!] \in \mathscr{NE} && \text{(Equation (1))}
\end{aligned}$$

Proof (Equation (4)). Follows immediately from Equation (7)

C Structural taint semantics

In this section, we give an equivalent structural definition of the concrete reachability taint semantics $\mathcal{S}_t[\![\mathtt{S}]\!]$.

Skip statement ($\mathtt{S} \coloneqq \mathtt{skip}$)

$$\hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}, \mathcal{T}) \triangleq (\mathcal{R}, \mathcal{E}, \mathcal{T})$$

Input read statement ($\mathtt{S} \coloneqq \mathtt{x} \leftarrow \mathtt{input()}$)

$$\begin{array}{l}\hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}, \mathcal{T}) \triangleq \\ \quad \text{let } (\mathcal{R}_1, \mathcal{E}_1) = \mathcal{S}[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}) \text{ in} \\ \quad \text{let } \mathcal{T}_1 = \{\, \mathtt{y} \in \mathcal{T} \mid \mathtt{y} \neq \mathtt{x} \,\} \cup \\ \quad\quad \{\, \mathtt{x} \mid \exists ((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in \mathcal{R} : \\ \quad\quad\quad m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 : \mathtt{hd}(i_1) \neq \mathtt{hd}(i'_1) \,\} \text{ in} \\ \quad (\mathcal{R}_1, \mathcal{E}_1, \mathcal{T}_1)\end{array}$$

Random read statement ($\mathtt{S} \coloneqq \mathtt{x} \leftarrow \mathtt{rand()}$)

$$\begin{array}{l}\hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}, \mathcal{T}) \triangleq \\ \quad \text{let } (\mathcal{R}_1, \mathcal{E}_1) = \mathcal{S}[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}) \text{ in} \\ \quad \text{let } \mathcal{T}_1 = \{\, \mathtt{y} \in \mathcal{T} \mid \mathtt{y} \neq \mathtt{x} \,\} \cup \\ \quad\quad \{\, \mathtt{x} \mid \exists ((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in \mathcal{R} : \\ \quad\quad\quad m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 : \mathtt{hd}(r_1) \neq \mathtt{hd}(r'_1) \,\} \text{ in} \\ \quad (\mathcal{R}_1, \mathcal{E}_1, \mathcal{T}_1)\end{array}$$

Observe that the scenario where $\mathtt{x}$ becomes tainted after the execution of $\mathtt{x} \leftarrow \mathtt{rand()}$ can only occur if there exist two executions that only differ in the input sequence, resulting in one of the two reading from the random sequence at least one additional time. This happens when the user can control how many times there is a read from the sequence of random numbers.

Statement composition ($\mathtt{S} \coloneqq \mathtt{S}_1 \,;\, \mathtt{S}_2$)

$$\hat{\mathcal{S}}_t[\![\mathtt{S}_1 \,;\, \mathtt{S}_2]\!](\mathcal{R}, \mathcal{E}, \mathcal{T}) \triangleq \hat{\mathcal{S}}_t[\![\mathtt{S}_2]\!](\hat{\mathcal{S}}_t[\![\mathtt{S}_1]\!](\mathcal{R}, \mathcal{E}, \mathcal{T}))$$

As discussed in the paper, if statements can generate *implicit flows* [28]. We define a helper function to compute the set of variables that are tainted due to implicit flows. This function considers pairs of executions that initially differ only by the user input. If two executions follow different branches and yield different values for a variable, such a variable is tainted. This happens when the evaluation of the boolean condition depends

on the user input.

$$
\begin{aligned}
&\mathtt{diff}[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!] : (\mathbb{D} \times \mathbb{D}) \to \wp(\mathbb{V}) \\
&\mathtt{diff}[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!](\mathcal{R}, \mathcal{E}) \triangleq \\
&\quad \{\, \mathtt{x} \in \mathbb{V} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in \mathcal{R} : \\
&\qquad \text{let } (\mathcal{R}_t, \mathcal{E}_t) = \mathcal{S}[\![\mathtt{S}_t]\!](\mathtt{test}[\![\mathtt{B}]\!]\{((m_0, i_0, r_0), (m_1, i_1, r_1))\}, \mathcal{E}) \text{ in} \\
&\qquad \text{let } (\mathcal{R}_e, \mathcal{E}_e) = \mathcal{S}[\![\mathtt{S}_e]\!](\mathtt{test}[\![\neg\mathtt{B}]\!]\{((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1))\}, \mathcal{E}) \text{ in} \\
&\qquad \exists(m_2, i_2, r_2) : ((m_0, i_0, r_0), (m_2, i_2, r_2)) \in \mathcal{R}_t : \\
&\qquad \exists(m'_2, i'_2, r'_2) : ((m'_0, i'_0, r'_0), (m'_2, i'_2, r'_2)) \in \mathcal{R}_e : \\
&\qquad m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 : \mathcal{B}[\![\mathtt{B}]\!]m_1 = \mathtt{tt}, \mathcal{B}[\![\mathtt{B}]\!]m'_1 = \mathtt{ff} : \\
&\qquad m_2[\mathtt{x}] \neq m'_2[\mathtt{x}] \,\}
\end{aligned}
$$

If statement ($\mathtt{S} \coloneqq \mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e$)

$$
\begin{aligned}
&\hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}, \mathcal{T}) \triangleq \\
&\quad \text{let } (\mathcal{R}_t, \mathcal{E}_t, \mathcal{T}_t) = \hat{\mathcal{S}}_t[\![\mathtt{S}_t]\!](\mathtt{test}[\![\mathtt{B}]\!]\mathcal{R}, \mathcal{E}, \alpha_t(\{\mathtt{test}[\![\mathtt{B}]\!]\mathcal{R}\})) \text{ in} \\
&\quad \text{let } (\mathcal{R}_e, \mathcal{E}_e, \mathcal{T}_e) = \hat{\mathcal{S}}_t[\![\mathtt{S}_e]\!](\mathtt{test}[\![\neg\mathtt{B}]\!]\mathcal{R}, \mathcal{E}, \alpha_t(\{\mathtt{test}[\![\neg\mathtt{B}]\!]\mathcal{R}\})) \text{ in} \\
&\quad \text{let } \mathcal{T}_{te} = \mathtt{diff}[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!](\mathcal{R}, \mathcal{E}) \text{ in} \\
&\quad (\mathcal{R}_t \cup \mathcal{R}_e, \mathcal{E}_t \cup \mathcal{E}_e \cup \mathtt{err}[\![\mathtt{B}]\!]\mathcal{R}, \mathcal{T}_t \cup \mathcal{T}_e \cup \mathcal{T}_{te})
\end{aligned}
$$

Observe that since the boolean evaluation is filtering $\mathcal{R}$, $\alpha_t(\{\mathtt{test}[\![\mathtt{B}]\!]\mathcal{R}\})$ is always a subset of $\alpha_t(\{\mathcal{R}\})$. This follows immediately from the monotonicity of α_t (see Equation (2)), which implies that $\mathcal{T}$ is a sound overapproximation of $\alpha_t(\{\mathtt{test}[\![\mathtt{B}]\!]\mathcal{R}\})$. The taint semantics for while statements is expressed as a least fixpoint.

While statement ($\mathtt{S} \coloneqq \mathtt{while\ (B)\ S}_b$)

$$
\begin{aligned}
&\hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\mathcal{R}, \mathcal{E}, \mathcal{T}) \triangleq \text{let } (\mathcal{R}_f, \mathcal{E}_f, \mathcal{T}_f) = \mathrm{lfp}\, F \text{ in} \\
&\qquad\qquad (\mathtt{test}[\![\neg\mathtt{B}]\!]\mathcal{R}_f, \mathcal{E}_f, \alpha_t(\{\mathtt{test}[\![\neg\mathtt{B}]\!]\mathcal{R}_f\})) \\
&\quad \text{where } F(\mathcal{R}_1, \mathcal{E}_1, \mathcal{T}_1) \triangleq (\mathcal{R}, \mathcal{E}, \mathcal{T}) \dot{\cup} \hat{\mathcal{S}}_t[\![\mathtt{if\ (B)\ S}_b\ \mathtt{else\ skip}]\!](\mathcal{R}_1, \mathcal{E}_1, \mathcal{T}_1)
\end{aligned}
$$

D Interval analysis helper functions

In this section, we define the helper functions used in the analysis proposed in Section 6. An *interval* is an element in $\mathbb{I} \triangleq \{[l, h] \mid l \in \mathbb{Z} \cup \{-\infty\}, h \in \mathbb{Z} \cup \{+\infty\}, l \leqslant h\} \cup \{\bot_i\}$. An interval abstract map is an element $\mathcal{R}_i^\sharp \in \mathbb{D}_i^\sharp \triangleq (\mathbb{V} \to \mathbb{I}) \cup \{\bot_i^\sharp\}$. We denote the interval abstract semantics of statements as $\mathcal{S}_i^\sharp[\![\mathtt{S}]\!] : \mathbb{D}_i^\sharp \to \mathbb{D}_i^\sharp$. To simplify the presentation, we ignore the abstract semantics for errors. The concretization function γ_i and the arithmetic abstract evaluation $\mathcal{A}_i^\sharp[\![\mathtt{A}]\!] : \mathbb{D}_i^\sharp \to \mathbb{I}$ are standard. The domain is assumed to be equipped with standard functions $\subseteq_i^\sharp$, $\cup_i^\sharp$, $\cap_i^\sharp$, and $\mathtt{test}_i^\sharp[\![\mathtt{A}]\!]$ [25]. We first define the function $\mathtt{isconst}_i^\sharp : \mathbb{I} \to \mathbb{B}$ which returns $\mathtt{tt}$ if the interval is a singleton.

$$
\mathtt{isconst}_i^\sharp(\bot_i) \triangleq \mathtt{tt} \qquad\qquad \mathtt{isconst}_i^\sharp([l, h]) \triangleq l = h
$$

The initial map $\mathcal{I}_i^\sharp$ is simply $\lambda \mathtt{x} \,.\, [-\infty, +\infty]$. We can then proceed to define $\mathtt{taint}_i^\sharp[\![\mathtt{A}]\!] : (\mathbb{D}_i^\sharp \times \wp(\mathbb{V})) \to \mathbb{B}$. The extension to boolean expressions is trivial.

$$
\begin{aligned}
\mathtt{taint}_i^\sharp[\![\mathtt{A}]\!](\bot_i^\sharp, \mathcal{T}^\sharp) &\triangleq \mathtt{ff} \\
\mathtt{taint}_i^\sharp[\![n]\!](\mathcal{R}_i^\sharp, \mathcal{T}^\sharp) &\triangleq \mathtt{ff} \\
\mathtt{taint}_i^\sharp[\![\mathtt{x}]\!](\mathcal{R}_i^\sharp, \mathcal{T}^\sharp) &\triangleq \begin{cases} \mathtt{ff} & \text{if } \mathtt{isconst}_i^\sharp(\mathcal{R}_i^\sharp[\mathtt{x}]) \\ \mathtt{tt} & \text{if } \mathtt{x} \in \mathcal{T}^\sharp \\ \mathtt{ff} & \text{otherwise} \end{cases} \\
\mathtt{taint}_i^\sharp[\![\mathtt{A}_1 \diamond \mathtt{A}_2]\!](\mathcal{R}_i^\sharp, \mathcal{T}^\sharp) &\triangleq \begin{cases} \mathtt{ff} & \text{if } \mathtt{isconst}_i^\sharp(\mathcal{A}_i^\sharp[\![\mathtt{A}_1 \diamond \mathtt{A}_2]\!]\mathcal{R}_i^\sharp) \\ \mathtt{tt} & \text{if } \mathtt{taint}_i^\sharp[\![\mathtt{A}_1]\!](\mathcal{R}_i^\sharp, \mathcal{T}^\sharp) \text{ or } \mathtt{taint}_i^\sharp[\![\mathtt{A}_2]\!](\mathcal{R}_i^\sharp, \mathcal{T}^\sharp) \\ \mathtt{ff} & \text{otherwise} \end{cases}
\end{aligned}
$$

The function $\mathtt{zero}_i^\sharp[\![\mathtt{A}]\!] : \mathbb{D}_i^\sharp \to \mathbb{B}$ is defined as follows.

$$
\mathtt{zero}_i^\sharp[\![\mathtt{A}]\!](\mathcal{R}_i^\sharp) \triangleq [0,0] \subseteq_i^\sharp \mathcal{A}_i^\sharp[\![\mathtt{A}]\!]\mathcal{R}_i^\sharp
$$

The evaluation of an element in the random sequence `rand[i]` is simply $[-\infty, +\infty]$. We define a helper function $\mathtt{haserror}_i^\sharp[\![\mathtt{A}]\!] : \mathbb{D}_i^\sharp \to \mathbb{B}$ to determine whether the evaluation of `A` possibly results in an error. The extension to boolean expressions is trivial.

$$
\begin{aligned}
\mathtt{haserror}_i^\sharp[\![\mathtt{A}]\!]\bot_i^\sharp &\triangleq \mathtt{ff} \\
\mathtt{haserror}_i^\sharp[\![n]\!]\mathcal{R}_i^\sharp &\triangleq \mathtt{ff} \\
\mathtt{haserror}_i^\sharp[\![\mathtt{x}]\!]\mathcal{R}_i^\sharp &\triangleq \mathtt{ff} \\
\mathtt{haserror}_i^\sharp[\![\mathtt{A}_1 \diamond \mathtt{A}_2]\!]\mathcal{R}_i^\sharp &\triangleq \begin{cases} \mathtt{tt} & \text{if } \diamond = \mathtt{/},\ \mathtt{zero}_i^\sharp[\![\mathtt{A}_2]\!](\mathcal{R}_i^\sharp) \\ \mathtt{tt} & \text{if } \mathtt{haserror}_i^\sharp[\![\mathtt{A}_1]\!]\mathcal{R}_i^\sharp \text{ or } \mathtt{haserror}_i^\sharp[\![\mathtt{A}_2]\!]\mathcal{R}_i^\sharp \\ \mathtt{ff} & \text{otherwise} \end{cases}
\end{aligned}
$$

The function $\mathtt{assigned}^\sharp[\![\mathtt{S}]\!]$ must be sound with respect to the following:

$$
\begin{aligned}
\{\, \mathtt{x} \mid\ & \exists((m_0, i_0, r_0), (m_1, i_1, r_1)) \in \gamma_t(\mathcal{R}^\sharp) : \exists(m_2, i_2, r_2) : \\
& ((m_0, i_0, r_0), (m_2, i_2, r_2)) \in \mathcal{S}[\![\mathtt{S}]\!](\{((m_0, i_0, r_0), (m_1, i_1, r_1))\}, \varnothing) : \\
& m_1[\mathtt{x}] \neq m_2[\mathtt{x}] \,\} \subseteq \mathtt{assigned}^\sharp[\![\mathtt{S}]\!]\mathcal{R}^\sharp
\end{aligned}
$$

For intervals, we define the function $\texttt{assigned}_i^\sharp[\![\texttt{S}]\!] : \mathbb{D}_i^\sharp \to \wp(\mathbb{V})$ as follows.

$$
\begin{aligned}
\texttt{assigned}_i^\sharp[\![\texttt{S}]\!]\bot_i^\sharp &\triangleq \varnothing \\
\texttt{assigned}_i^\sharp[\![\texttt{skip}]\!]\mathcal{R}_i^\sharp &\triangleq \varnothing \\
\texttt{assigned}_i^\sharp[\![\texttt{S}_1\,\texttt{;}\,\texttt{S}_2]\!]\mathcal{R}_i^\sharp &\triangleq \texttt{assigned}_i^\sharp[\![\texttt{S}_1]\!]\mathcal{R}_i^\sharp \cup \texttt{assigned}_i^\sharp[\![\texttt{S}_2]\!](\mathcal{S}_i^\sharp[\![\texttt{S}_1]\!]\mathcal{R}_i^\sharp) \\
\texttt{assigned}_i^\sharp[\![\texttt{x} \leftarrow \texttt{input()}]\!]\mathcal{R}_i^\sharp &\triangleq \{\texttt{x}\} \\
\texttt{assigned}_i^\sharp[\![\texttt{x} \leftarrow \texttt{rand()}]\!]\mathcal{R}_i^\sharp &\triangleq \{\texttt{x}, \texttt{i}\} \\
\texttt{assigned}_i^\sharp[\![\texttt{x} \leftarrow \texttt{A}]\!]\mathcal{R}_i^\sharp &\triangleq \{\,\texttt{ret} \mid \texttt{haserror}_i^\sharp[\![\texttt{A}]\!]\mathcal{R}_i^\sharp\,\} \cup \\
&\qquad \{\,\texttt{x} \mid \neg(\texttt{isconst}_i^\sharp(\mathcal{R}_i^\sharp[\texttt{x}]) \text{ and } \mathcal{R}_i^\sharp[\texttt{x}] = \mathcal{A}_i^\sharp[\![\texttt{A}]\!]\mathcal{R}_i^\sharp)\,\} \\
\texttt{assigned}_i^\sharp[\![\texttt{if (B) S}_t\,\texttt{else S}_e]\!]\mathcal{R}_i^\sharp &\triangleq \{\,\texttt{ret} \mid \texttt{haserror}_i^\sharp[\![\texttt{B}]\!]\mathcal{R}_i^\sharp\,\} \cup \\
&\qquad \texttt{assigned}_i^\sharp[\![\texttt{S}_t]\!](\texttt{test}_i^\sharp[\![\texttt{B}]\!]\mathcal{R}_i^\sharp) \cup \\
&\qquad \texttt{assigned}_i^\sharp[\![\texttt{S}_e]\!](\texttt{test}_i^\sharp[\![\neg\texttt{B}]\!]\mathcal{R}_i^\sharp) \\
\texttt{assigned}_i^\sharp[\![\texttt{while (B) S}_b]\!]\mathcal{R}_i^\sharp &\triangleq \text{let } (\mathcal{R}_f^\sharp, X_f) = \lim F^n(\bot_i^\sharp, \varnothing) \text{ in } X_f \\
\text{where } F(\mathcal{R}_1^\sharp, X_1) &\triangleq \text{let } \mathcal{R}_2^\sharp = \mathcal{R}_1^\sharp \,\nabla_i\, (\mathcal{R}_i^\sharp \cup_i^\sharp \mathcal{S}_i^\sharp[\![\texttt{if (B) S}_b\,\texttt{else skip}]\!]\mathcal{R}_1^\sharp) \text{ in} \\
&\qquad \text{let } X_2 = X_1 \cup \texttt{assigned}_i^\sharp[\![\texttt{if (B) S}_b\,\texttt{else skip}]\!]\mathcal{R}_1^\sharp \text{ in} \\
&\qquad (\mathcal{R}_2^\sharp, X_2)
\end{aligned}
$$

The function $\texttt{const}_i^\sharp : \mathbb{D}_i^\sharp \to \wp(\mathbb{V})$ is defined as follows.

$$\texttt{const}_i^\sharp(\mathcal{R}_i^\sharp) \triangleq \{\,\texttt{x} \mid \texttt{isconst}_i^\sharp(\mathcal{R}_i^\sharp[\texttt{x}])\,\}$$

E Analysis Sources and Runtime Errors

In this section, we report the list of runtime errors detected by our analysis, and the list of sources of tainted data. Our analysis can detect the following runtime errors:

- Null pointer dereference
- Invalid pointer dereference
- Index-out-of-bounds
- Dangling pointer dereference
- Use after free
- Modification of read-only memory
- Division by zero
- Integer overflow
- Invalid bit shift operation
- Invalid pointer comparison
- Invalid pointer subtraction
- Double free
- Insufficient number of variadic arguments
- Insufficient number of format arguments
- Invalid format argument type
- Floating point division by zero
- Floating point invalid operation
- Floating point overflow

– Violation of a stub language requirement. See [49] for more information about the stub modeling language.

The following functions in the C standard library generate tainted data:

– `getchar`
– `getchar_unlocked`
– `fgetc`
– `fgetc_unlocked`
– `getc`
– `getc_unlocked`
– `getw`
– `fgets`
– `fgets_unlocked`
– `gets`
– `getline`
– `getdelim`
– `fread`
– `fread_unlocked`
– `recv`
– `recvfrom`
– `scanf`
– `fscanf`
– `sscanf`