A Logic for Anytime Non-Interference

Clément Aubert
School of Computer and Cyber Sciences, Augusta University

Neea Rusch
School of Computer and Cyber Sciences, Augusta University

Abstract

Non-interference is an information flow policy for guaranteeing confidentiality, i.e., that effects of sensitive data are not exposed to lower-level users, even indirectly. When phrased in terms of programming languages, non-interference is studied by attaching security classes to variables, then analyzing the classes to determine if a violation, or a data "leak", can occur. Security type systems are common controls for analyzing and enforcing non-interference. Unfortunately, they require inference algorithms, program-level security specifications, non-standard compilers, and are generally too restrictive or complex for practical implementation. In this paper, we present a program logic $\top^*_{\text{NI}}$ that guarantees the semantic security property of *anytime non-interference*. By anytime, we mean a malicious actor with low-level access cannot infer anything about higher-level values at any point of the program execution. The logic links non-interference violations precisely to the faulty commands and violations cannot be erased by program composition. We draw rich inspiration from complexity-theoretic flow calculi, but obtaining a logic for security analysis required significant adjustments. Finally, we share a prototype to demonstrate $\top^*_{\text{NI}}$ can be implemented as an automatic, annotation-free, static security analyzer to obtain confidentiality guarantees in practice.

2012 ACM Subject Classification Security and privacy → Software and application security; Software and its engineering → Automated static analysis; Theory of computation → Logic and verification

Keywords and phrases Confidentiality, Information Flow, Security Policies, Language-based Security, Automated Security Analysis, Flow Calculus

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Category Regular research

Supplementary Material *Software*: `https://github.com/statycc/tyni`

Acknowledgements We would like to thank the reviewers and participants of the 19th Workshop on Programming Languages and Analysis for Security (PLAS 2024) for their comments.

1 Introduction

Verifying that data is handled securely during computation is challenging because it requires information beyond the program syntax. For example, consider a hash function that computes a checksum of its input. Assume there exists a malicious actor who can observe outputs of the hash function. If all inputs are public data, we can guarantee the function does not expose secrets to the actor. However, if we change the inputs to secret data, like social security numbers, we no longer have the same guarantee for the same function. The hash function then "leaks" information that possibly enables the actor to recover the secret data.

Non-interference [15] is a classical semantic security property that constrains information flow during computation. It is a mechanism to enforce *confidentiality*, i.e., concealment of information or resources from unauthorized parties [22]. Non-interference is an attractive target of study because it offers strong end-to-end guarantees for data protection, it is inherently compositional, and can be enforced with program logics or security type systems [10, 14]. Informally, a program is non-interfering when secret data does not affect calculation of its public outputs [27]. In other words, data can only stay at the same security class or flow

42nd Conference on Very Important Topics (CVIT 2016).
Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

to higher classes. Although a desirable property, non-interference is in general undecidable by Rice's Theorem [26] and constructing a system that completely adheres to non-interference is overly restrictive to capture real-world security requirements [9, 10]. However, unattainability of an ideal construction does not restrict analysis of imperfect systems. Program analysis enables detecting information flow issues, supports informed assessments of vulnerabilities, and identifies potential mitigations.

Our work extends the analysis of non-interference in theoretical directions while yielding practical advantages. In literature, terminology around non-interference is defined somewhat fluidly [27]; admitting multiple different but related formal definitions [24], and generalizing to the informal description provided previously. In this paper, we introduce the notion of *anytime non-interference*. The anytime property is powerful because it accounts for intermediate states of computation. Thus, it is strictly stronger than the classic non-interference that is expressed in terms of inputs and outputs. To lift our theory toward the real world, we provide our main result: a program logic $\top^*_{\text{NI}}$ that enables lightweight automatic static program analysis of anytime non-interference. We demonstrate practicality of $\top^*_{\text{NI}}$ through examples, a prototype implementation, and discussions of how anytime non-interference elegantly supports numerous program analysis applications.

Anytime non-interference tracks potential information flow leaks in every legal program state. A non-interfering program can be interrupted arbitrarily without compromising its non-interference guarantee. Conversely, once a violating operation has occurred, it is impossible to erase it. We consider time as updates of public variable values. In other words, the latency between public variable updates is instant. A change between two secret values, that causes a loop to iterate longer according to a physical clock, has no observable side effect. In general, anytime non-interference models security at the abstraction level of programming languages and excludes lower-level execution details. However, we consider the approach justified because potential information flow issues are often detectable from syntax.

Anytime non-interference is furthermore *termination-insensitive*. Because information signaled through termination can leak secrets indirectly, termination handling is an ongoing design challenge for non-interference systems [7]. Untrusted programs, that pose high security risks, require strong *progress-sensitive* non-interference that considers both termination and I/O interactions [17]. Trusted programs, with predictable run-time behavior, permit weaker security checks and termination-insensitivity. Unfortunately, the binary situation provides no middle ground for programs that mix trusted and untrusted code, e.g., by dynamic code loading. In Sect. 6.2 we discuss how to address this limitation by partitioning computations based on security classes. This hybrid approach relaxes the limitations of termination-insensitivity and monolithic termination handing.

1.1 The Essential Security Terminology Decoded

Our work belongs to the domain of *language-based security* [28, 27], where programming languages principles (semantics, analysis, type systems, rewriting, etc.) are used to strengthen application security. The $\top^*_{\text{NI}}$ logic draws rich inspiration from implicit computational complexity (refer to Sect. 6.3), and has applications in static program analysis; thus our work intersects many related fields. Although we assume prior familiarity with logic and programming languages, we define the relevant security concepts in this section.

Information flow denotes an observable action between two agents A and B. If an action performed by A is observable to B, then there exists an information flow from A to B. The flow is *explicit* if it is directly observable from a single action. The flow is *implicit* when it is not directly observable, but reveals deductively the initial performed action, after a sequence of other actions. To represent information flow, we manipulate the conventional lattice model à la Denning [12].

An *information flow policy* is a statement of what is, and what is not, permissible [22] for a program `C` in terms of flow between its variables; formally defined as follows:

▶ **Definition 1** (Information Flow Policy [29], Class Assignment)**.** *An* information flow policy *is a lattice* $\mathrm{SC} = (SC, <)$ *where* SC *is a partially* $<$*-ordered finite set of* security classes*.*

We write ℓ *for the class assignment that assigns statically and definitely to each variable* `x` *occurring in a program* `C` *its security class* $\ell(\mathtt{x}) \in \mathrm{SC}$*.*

By abuse of notation, we assume that a class assignment always comes with an information flow policy, we write $c \in \mathrm{SC}$, and for any two classes c_1, c_2, we write $c_1 \leqslant c_2$ if $c_1 < c_2$ or $c_1 = c_2$, and $c_1 \perp c_2$ if $c_1 \nleqslant c_2$ and $c_2 \nleqslant c_1$–in this case, we say that c_1 and c_2 are *orthogonal.*

A simple policy has two security classes, e.g., $\mathrm{LH} = (\{l, h\}, \{l < h\})$—for *low* and *high*; but a policy can be arbitrarily complex (refer to Ex. 20, located in Appendix, for a more concrete example). We generally use a Hasse diagram to represent the information flow policy and class assignment in a compact manner, as follows:

$$\begin{array}{llll} SC = \{b, m_1, m_2, t\} & & & \\ b < m_1 & b < m_2 & m_1 < t & m_2 < t \\ \ell(\mathtt{x}) = b & \ell(\mathtt{y}_1) = m_1 & \ell(\mathtt{y}_2) = m_2 & \ell(\mathtt{w}) = \ell(\mathtt{z}) = t \end{array}$$

$$\begin{tikzcd}
& \ell(\mathtt{w}) = \ell(\mathtt{z}) & \\
\ell(\mathtt{y}_1) \arrow[ur] & & \ell(\mathtt{y}_2) \arrow[ul] \\
& \ell(\mathtt{x}) \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

An *information flow control* (IFC) is a mechanism to enforce a policy [22]. Security type systems (presented in Sect. 6.3) are an example of a programming languages based IFCs. They enforce a policy by annotating a program with security types. Then, to be secure, a program must pass a compile-time type check. A sound IFC guarantees to find all policy violations and a precise IFC avoids raising excessive false alarms.

Formal security analysis uses the terms system model, security objective, and an attacker model [6, 8]. Our *system model*, i.e., the system we want to secure, is a sequential imperative program, as specified in Sect. 3.1, with effectful function calls discussed in Sect. 5. Our *security objective*, i.e., the system behaviors that are considered secure, is defined by anytime non-interference (Def. 14). An *adversary* is a malicious actor who poses a threat to the system. An *attacker model* specifies the capabilities and motivations of the adversary. We assume a *program-centric* [17] attacker model, where the adversary can (i) see the program syntax (ii) control public inputs and observe public outputs, and (iii) up to the attacker's security class, access memory registers after updates.

1.2 Contributions

Our contributions are three-fold.

1. The main result is a lightweight syntactic IFC logic $\top^*_{\mathrm{NI}}$ (Sect. 3), with a built-in automatable inference algorithm, that captures the semantic property of anytime non-interference.
2. We introduce the definition of anytime non-interference (Def. 14), and prove its correspondence with $\top^*_{\mathrm{NI}}$, that follows from the fundamental theorem (Thm. 16).
3. We demonstrate the promising practical utility of $\top^*_{\mathrm{NI}}$ (Sect. 6), to include by describing our prototype static analyzer TYNI for analysis of `Java` programs.

Sect. 5 furthermore discusses in detail how our approach can accommodate different treatments of functions with and without side effects, and Appendix B gathers additional examples to help understanding.

2 High-level Overview

We consider deterministic imperative programs, with conventional operational semantics, and variables of basic data types (integers, strings, etc.). Let our expository program be

```
if (z==1)
  then if (x==1) then y = 1 else y = 0
  else x = y
```

In the program, certain data flows are potentially problematic. The assignment `x = y` is an instances of an explicit flow, since there is a direct flow from variable `y` to `x`. The control expressions `z==1` and `x==1` represent implicit flows. They reveal information over execution paths, and indirectly expose values of the control statement variables. Admissibility of these data flows depends on the security classes of the variables, since non-interference forbids data flowing from higher classes to lower or between orthogonal classes. A sound IFC detects such issues and raises an alarm.

The logic $\top^*_{\mathrm{NI}}$ produces a matrix of coefficients by applying inference rules to programs. In a single derivation, it captures dependencies between all the program variables, for all execution paths and security classes. The matrix is interpreted by matching the *in-variables*, $\mathtt{v}_{\mathrm{in}}$ (rows), with the *out-variables*, $\mathtt{v}_{\mathrm{out}}$ (columns). The coefficients indicate:

- · – *no* (non-interference) *violation*, no dependency from $\mathtt{v}_{\mathrm{in}}$ to $\mathtt{v}_{\mathrm{out}}$,
- 💧 – a *violation*—or “leak”, hence the symbol—, if $\ell(\mathtt{v}_{\mathtt{out}}) < \ell(\mathtt{v}_{\mathtt{in}})$ or $\ell(\mathtt{v}_{\mathtt{out}}) \perp \ell(\mathtt{v}_{\mathtt{in}})$.

The matrix of the expository program, expectedly shows *potential* violations in from `y` to `x` (explicit, in gray here) and `x` to `y`, `z` to `x`, and `z` to `y` (implicit). The matrix gives a summary of potential violations for the program that induced it. Evaluating the matrix determines if the program is non-interfering; or if there exists a class assignment that makes the program non-interfering. The evaluation function is parametric on the policy, enabling evaluation against different policies.

$$\begin{array}{c c} & \begin{array}{ccc} \mathtt{x} & \mathtt{y} & \mathtt{z} \end{array} \\ \begin{array}{c} \mathtt{x} \\ \mathtt{y} \\ \mathtt{z} \end{array} & \begin{pmatrix} \cdot & \text{💧} & \cdot \\ {\color{gray}\text{💧}} & \cdot & \cdot \\ \text{💧} & \text{💧} & \cdot \end{pmatrix} \end{array}$$

3 The Non-interference Logic

3.1 A Simple Imperative While Language

We use a simple imperative `while` language, with semantics similar to `C`. The grammar is given in Fig. 1. The language supports arrays and we let `for` and `do...while` loops be represented using `while` loops. How function calls can be added is discussed in Sect. 5. The language subsumes (up to `letvar` construct) the “core block-structured language” [29], and it maps easily to the core fragment of `C`, Java, and other imperative programming languages.

A variable `x`, `y`, `z`, … represents either an undetermined “primitive” data type, e.g., not a reference variable, or an array, whose indices are given by an expression. We reserve `t` for arrays. An expression is either a variable, a value (e.g., integer literal) or the application to expressions of some operator *op*, which can be e.g., relational (`==`, `<`, etc.) or arithmetic (`+`, `-`, etc.). We let `e` (resp. `C`) range over expressions (resp. commands). We also use compound assignment operators and write e.g., `x++` for `x+=1`. We assume commands to be correct, e.g., with operators correctly applied to expressions, no out-of-bounds errors, etc. A *program* `C` is a sequence of commands, each command being either an *assignment*, a *skip*, a *branching*, a while *loop* or the *composition of two commands*. A program $\mathtt{C}'$ is *a sub-program of* `C`, denoted $\mathtt{C}' \subseteq \mathtt{C}$, if $\mathtt{C}'$ occurs verbatim in `C`. We also define the following sets of variables.

$$\begin{aligned}
&\mathit{var} ::= \mathtt{i} \mid \cdots \mid \mathtt{t} \mid \cdots \mid \mathtt{x}_1 \mid \cdots \mid \mathit{var}[\mathit{exp}] && \text{(Variable)}\\
&\mathit{exp} ::= \mathit{var} \mid \mathit{val} \mid \mathit{op}(\mathit{exp}, \dots, \mathit{exp}) && \text{(Expression)}\\
&\mathit{com} ::= \mathit{var}\ \mathtt{=}\ \mathit{exp} \mid \mathtt{skip} \mid \mathtt{if}\ \mathit{exp}\ \mathtt{then}\ \mathit{com}\ \mathtt{else}\ \mathit{com} \mid \mathtt{while}\ \mathit{exp}\ \mathtt{do}\ \mathit{com} \mid \mathit{com};\mathit{com} && \\
& && \text{(Command)}
\end{aligned}$$

Figure 1 A simple imperative `while` language

C	Out(C)	In(C)	Occ(C) = Out(C) ∪ In(C)
`x = e`	x	Occ(e)	x ∪ Occ(e)
$\mathtt{t[e_1]\ =\ e_2}$	t	$\mathrm{Occ}(\mathtt{e}_1) \cup \mathrm{Occ}(\mathtt{e}_2)$	$\mathtt{t} \cup \mathrm{Occ}(\mathtt{e}_1) \cup \mathrm{Occ}(\mathtt{e}_2)$
`skip`	$\emptyset$	$\emptyset$	$\emptyset$
$\mathtt{if\ e\ then\ C_1\ else\ C_2}$	$\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{In}(\mathtt{C}_1) \cup \mathrm{In}(\mathtt{C}_2)$	$\mathrm{Occ}(\mathtt{e}) \cup \mathrm{Occ}(\mathtt{C}_1) \cup \mathrm{Occ}(\mathtt{C}_2)$
`while e do C`	Out(C)	Occ(e) ∪ In(C)	Occ(e) ∪ Occ(C)
$\mathtt{C_1;C_2}$	$\mathrm{Out}(\mathtt{C}_1) \cup \mathrm{Out}(\mathtt{C}_2)$	$\mathrm{In}(\mathtt{C}_1) \cup \mathrm{In}(\mathtt{C}_2)$	$\mathrm{Occ}(\mathtt{C}_1) \cup \mathrm{Occ}(\mathtt{C}_2)$

Table 1 Definition of Out, In and Occ for commands

▶ **Definition 2** (Occ, Out and In). *We define the* variables occurring in an expression $\mathtt{e}$ *by:*

$$\mathrm{Occ}(\mathtt{x}) = \mathtt{x}\ \mathrm{Occ}(\mathtt{op(e_1,\dots,e}_n\mathtt{)}) = \cup_{i=1}^{n} \mathrm{Occ}(\mathtt{e}_i) \quad \mathrm{Occ}(\mathtt{t[e]}) = \mathtt{t} \cup \mathrm{Occ}(\mathtt{e})\ \mathrm{Occ}(\mathrm{val}) = \emptyset$$

The set Occ(C) *(resp.* Out(C), In(C)*) of variables* occurring *in (resp.* modified *by,* used *by) a program* C *is defined in Table 1. We let* |Occ(C)| *be the cardinal of* Occ(C)*.*

3.2 Security-Flow Matrices for Non-interference Violation

The $\top^*_{\mathrm{NI}}$ logic relies fundamentally on its ability to analyze data-flow dependencies between variables occurring in commands. In this section, we define the principles of this dependency analysis, founded on the theory of *security-flow matrices*, and how it maps to the presented language. This dependency analysis is reminiscent of the one we developed to distribute loops [3]. We assume familiarity with monoids and matrices addition.

A security-flow matrix $\mathbb{M}(\mathtt{C})$ for a command C is a hollow matrix (i.e., a matrix with only · on the diagonal[1]) over a monoid with an implicit choice of a denumeration of Occ(C)[2]

▶ **Definition 3** (Security monoid). *The* security monoid *is* $(\{\cdot, \text{💧}\}, \max)$*, with* $\cdot < \text{💧}$.

This monoid is isomorphic to the two-element Boolean algebra with only the disjunction, with 💧 representing a possible (non-interference) violation that cannot be erased.

▶ **Definition 4** (Security-flow matrix). *Given a program* C*, its* security-flow matrix $\mathbb{M}(\mathtt{C})$ *is a* $|\mathrm{Occ}(\mathtt{C})| \times |\mathrm{Occ}(\mathtt{C})|$ *matrix over the security monoid, whose construction is the object of Sect. 3.3. For* $\mathtt{x}, \mathtt{y} \in \mathrm{Occ}(\mathtt{C})$*, we write* $\mathbb{M}(\mathtt{C})(\mathtt{x}, \mathtt{y})$ *for the coefficient in* $\mathbb{M}(\mathtt{C})$ *at the row corresponding to the* in-variable x *and column corresponding to the* out-variable y*.*

▶ **Definition 5** (Violation). *Given* C*, its security-flow matrix* $\mathbb{M}(\mathtt{C})$ *and a class assignment* ℓ*,* C has a violation *if there exists* x *and* y *such that* $\mathbb{M}(\mathtt{C})(\mathtt{x}, \mathtt{y}) = \text{💧}$ *and either* $\ell(\mathtt{y}) < \ell(\mathtt{x})$ *or* $\ell(\mathtt{y}) \perp \ell(\mathtt{x})$*:*

[1] This choice is clarified after Def. 5.

[2] We will use the order in which the variables occur in the program as their implicit order.

$$\begin{array}{cc} & \begin{array}{ccc} \cdots & \mathtt{y} & \cdots \end{array} \\ \begin{array}{c} \vdots \\ \mathtt{x} \\ \vdots \end{array} & \begin{pmatrix} \ddots & & \iddots \\ & \text{🌢} & \\ \iddots & & \ddots \end{pmatrix} \end{array} \implies \mathtt{C} \textit{ has a violation if } \ell(\mathtt{y}) < \ell(\mathtt{x}) \textit{ or } \ell(\mathtt{y}) \perp \ell(\mathtt{x}).$$

Since $\ell(\mathtt{x}) < \ell(\mathtt{x})$ and $\ell(\mathtt{x}) \perp \ell(\mathtt{x})$ are always false, there is no point keeping track of the values on the diagonal: this is why hollow matrices are enough. This is also confirmed by the intuition: it does not make sense to track data "leaking" from a variable to itself.

How a security-flow matrix is constructed by induction over the command is explained in Sect. 3.3. To avoid resizing matrices whenever additional variables are considered, we identify $\mathbb{M}(\mathtt{C})$ with its embedding in any larger matrix, i.e., we abusively call the security-flow matrix of $\mathtt{C}$ any matrix containing $\mathbb{M}(\mathtt{C})$ (up to rows swapping and columns swapping) and containing $\cdot$ otherwise, implicitly viewing the additional rows and columns as variables not occuring in $\mathtt{C}$. Visually, this means that the following matrices are all viewed as $\mathbb{M}(\mathtt{C})$ with $\mathrm{Occ}(\mathtt{C}) = \{\mathtt{x}, \mathtt{y}\}$ and $\mathbb{M}(\mathtt{C})(\mathtt{x}, \mathtt{y}) = \text{🌢}$:

$$\begin{array}{cc} & \begin{array}{cc} \mathtt{x} & \mathtt{y} \end{array} \\ \begin{array}{c} \mathtt{x} \\ \mathtt{y} \end{array} & \begin{pmatrix} \cdot & \text{🌢} \\ \cdot & \cdot \end{pmatrix} \end{array} \qquad \begin{array}{cc} & \begin{array}{ccc} \mathtt{y} & \mathtt{x} & \mathtt{z} \end{array} \\ \begin{array}{c} \mathtt{y} \\ \mathtt{x} \\ \mathtt{z} \end{array} & \begin{pmatrix} \cdot & \cdot & \cdot \\ \text{🌢} & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix} \end{array} \qquad \begin{array}{cc} & \begin{array}{ccc} \mathtt{w} & \mathtt{x} & \mathtt{y} \end{array} \\ \begin{array}{c} \mathtt{w} \\ \mathtt{x} \\ \mathtt{y} \end{array} & \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \text{🌢} \\ \cdot & \cdot & \cdot \end{pmatrix} \end{array}$$

Continuing this example and using our compact presentation of information flow policy and class assignment as single Hasse diagram, $\mathtt{C}$ would have a violation with the level assignments $\begin{tikzcd} \ell(\mathtt{x}) \\ \ell(\mathtt{y}) \arrow[u] \end{tikzcd}$ and $\begin{tikzcd} & c_2 & \\ \ell(\mathtt{y}) \arrow[ur] & & \ell(\mathtt{x}) \arrow[ul] \\ & c_1 \arrow[ul] \arrow[ur] & \end{tikzcd}$, but would be free of violation with $\ell(\mathtt{x}) = \ell(\mathtt{y})$ or $\begin{tikzcd} \ell(\mathtt{x}) \\ \ell(\mathtt{y}) \arrow[u] \end{tikzcd}$.

3.3 Constructing Security-Flow Matrices

The security-flow matrix of a command is constructed by induction, using the security monoid. Appendix B gathers additional examples with longer discussion.

3.3.1 Base Cases: Assignment and Skip

The security-flow matrix for an assignment $\mathtt{C}$ simply tracks flows from $\mathrm{In}(\mathtt{C})$ to $\mathrm{Out}(\mathtt{C})$:

▶ **Definition 6** (Assignment). *Given an assignment* $\mathtt{C}$*, we define* $\mathbb{M}(\mathtt{C})$ *by:*

$$\mathbb{M}(\mathtt{C})(\mathtt{x}, \mathtt{y}) = \begin{cases} \text{🌢} & \textit{if } \mathtt{x} \in \mathrm{In}(\mathtt{C}),\ \mathtt{y} \in \mathrm{Out}(\mathtt{C}) \textit{ and } \mathtt{x} \neq \mathtt{y} \\ \cdot & \textit{otherwise} \end{cases}$$

We illustrate in Fig. 2 some basic cases: we consider an array a single entity, and that changing one value in it means being able to access it completely. More precisely, `t[i]` on the left-hand side of an assignment is a violation if $\ell(\mathtt{t}) > \ell(\mathtt{i})$ (resp. $\ell(\mathtt{t}) \perp \ell(\mathtt{i})$). Indeed, it implies that a lower-class (resp. orthogonal-class) variable ($\mathtt{i}$) can decide where to write in a higher-class (resp. orthogonal-class) variable ($\mathtt{t}$). However, `t[i]` as an expression (e.g., on the right-hand side of an assignment or in a condition, as discussed in Sect. 3.3.3) is acceptable as long as the variable(s) storing the result of this calculation or dependent on that condition's truth value have class higher or equal to `t` and `i` classes.

C	Out(C), In(C)	$\mathbb{M}(\texttt{C})$	C has violation(s) if …
`w = 3`	$\text{Out}(\texttt{C}) = \{\texttt{w}\}$ $\text{In}(\texttt{C}) = \emptyset$	$\begin{array}{cc} & \texttt{w} \\ \texttt{w} & (\cdot) \end{array}$	(Impossible)
`y = x`	$\text{Out}(\texttt{C}) = \{\texttt{y}\}$ $\text{In}(\texttt{C}) = \{\texttt{x}\}$	$\begin{array}{cc} & \begin{array}{cc} \texttt{y} & \texttt{x} \end{array} \\ \begin{array}{c} \texttt{y} \\ \texttt{x} \end{array} & \begin{pmatrix} \cdot & \cdot \\ \blacklozenge & \cdot \end{pmatrix} \end{array}$	$\ell(\texttt{y}) < \ell(\texttt{x})$ or $\ell(\texttt{y}) \perp \ell(\texttt{x})$.
`w = t[x + 1]`	$\text{Out}(\texttt{C}) = \{\texttt{w}\}$ $\text{In}(\texttt{C}) = \{\texttt{t}, \texttt{x}\}$	$\begin{array}{cc} & \begin{array}{ccc} \texttt{w} & \texttt{t} & \texttt{x} \end{array} \\ \begin{array}{c} \texttt{w} \\ \texttt{t} \\ \texttt{x} \end{array} & \begin{pmatrix} \cdot & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot \end{pmatrix} \end{array}$	$\ell(\texttt{w}) < \ell(\texttt{t})$, $\ell(\texttt{w}) \perp \ell(\texttt{t})$, $\ell(\texttt{w}) < \ell(\texttt{x})$ or $\ell(\texttt{w}) \perp \ell(\texttt{x})$.
`t[i] = u + j`	$\text{Out}(\texttt{C}) = \{\texttt{t}\}$ $\text{In}(\texttt{C}) = \{\texttt{i}, \texttt{u}, \texttt{j}\}$	$\begin{array}{cc} & \begin{array}{cccc} \texttt{t} & \texttt{i} & \texttt{u} & \texttt{j} \end{array} \\ \begin{array}{c} \texttt{t} \\ \texttt{i} \\ \texttt{u} \\ \texttt{j} \end{array} & \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot & \cdot \end{pmatrix} \end{array}$	$\ell(\texttt{t}) < \ell(\texttt{i})$, $\ell(\texttt{t}) \perp \ell(\texttt{i})$, $\ell(\texttt{t}) < \ell(\texttt{u})$, $\ell(\texttt{t}) \perp \ell(\texttt{u})$, $\ell(\texttt{t}) < \ell(\texttt{j})$ or $\ell(\texttt{t}) \perp \ell(\texttt{j})$.

Figure 2 Statement Examples, Sets, Representations of their Possible Violation(s).

$$\overset{\texttt{C}_1}{\begin{array}{cc} & \begin{array}{cccc} \texttt{w} & \texttt{x} & \texttt{y} & \texttt{z} \end{array} \\ \begin{array}{c} \texttt{w} \\ \texttt{x} \\ \texttt{y} \\ \texttt{z} \end{array} & \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \blacklozenge \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array}} + \overset{\texttt{C}_2}{\begin{array}{cc} & \begin{array}{cccc} \texttt{w} & \texttt{x} & \texttt{y} & \texttt{z} \end{array} \\ \begin{array}{c} \texttt{w} \\ \texttt{x} \\ \texttt{y} \\ \texttt{z} \end{array} & \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \blacklozenge & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array}} = \overset{\texttt{C}_1\texttt{;}\texttt{C}_2}{\begin{array}{cc} & \begin{array}{cccc} \texttt{w} & \texttt{x} & \texttt{y} & \texttt{z} \end{array} \\ \begin{array}{c} \texttt{w} \\ \texttt{x} \\ \texttt{y} \\ \texttt{z} \end{array} & \begin{pmatrix} \cdot & \cdot & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot & \cdot \\ \cdot & \blacklozenge & \cdot & \blacklozenge \\ \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array}}$$

```
w = w + x;
z = y + 2
```

```
x = y * 2;
z = 0
```

Figure 3 Security-Flow Matrix of Compositions.

▶ **Definition 7** (Skip). *We let* $\mathbb{M}(\texttt{skip})$ *be the matrix with* 0 *rows and columns.*

Identifying $\mathbb{M}(\texttt{skip})$ with its embeddings, it is the empty matrix of any size.

3.3.2 Composition as a Commutative Operation

The security-flow matrix for a composition of commands is an abstraction that allows manipulating a sequence of commands as one command with its own matrix.

▶ **Definition 8** (Composition). *We let* $\mathbb{M}(\texttt{C}_1\texttt{;}\cdots\texttt{;}\texttt{C}_n)$ *be* $\mathbb{M}(\texttt{C}_1) + \cdots + \mathbb{M}(\texttt{C}_n)$.

The composition of commands $\texttt{C}_1$ and $\texttt{C}_2$—themselves already the result of compositions of assignments involving disjoint variables—is illustrated in Fig. 3. Two important observations:

1. Some existing approaches might consider $\texttt{C}_1\texttt{;}\texttt{C}_2$ as free of violation even if $\ell(\texttt{z}) < \ell(\texttt{y})$, since `z = 0` will wipe out the content of `z` and "cancel" the violation introduced by `z = y`. The intuition is that an attacker observing the output (or even all the final values) cannot deduce anything about `z`'s value (and, transitively, about the value of the higher-class `y`) once the computation is over. Our "once a violation, always a violation" approach ignores

the fact that "ultimately", this violation may be hidden–the anytime non-interference guarantee is discussed in Sect. 4.

2. Interestingly, $\mathbb{M}(\texttt{C}_1\texttt{;C}_2) = \mathbb{M}(\texttt{C}_2\texttt{;C}_1)$ since composition is interpreted as a sum of matrices over our *commutative* security monoid. While previous flow-based approaches [2, 3, 19] requires semi-ring because composition was handled *via* product of matrices, the current set-up simplifies the machinery precisely to keep track of past violations.

3.3.3 A Correction for Implicit Flows

To account for implicit flows, branchings and loops require a *correction.* The main idea is that interpreting **if** `e` **then** $\texttt{C}_1$ **else** $\texttt{C}_2$ (resp. **while** `e` **do** `C`) require to record that all the variables modified in $\texttt{C}_1$ and $\texttt{C}_2$ (resp. in `C`) depend on the variables *occurring* in `e` (as opposed to the assignment considering the variables *used* by `C`).

▶ **Definition 9** (Correction). *The* correction $\mathrm{Cr}(\texttt{e})_{\texttt{C}}$ of an expression `e` on a program `C` *is*

$$\mathrm{Cr}(\texttt{e})_{\texttt{C}}(\texttt{x},\texttt{y}) = \begin{cases} \text{🌢} & \textit{if } \texttt{x} \in \mathrm{Occ}(\texttt{e}),\ \texttt{y} \in \mathrm{Out}(\texttt{C}) \textit{ and } \texttt{x} \neq \texttt{y} \\ \cdot & \textit{otherwise} \end{cases}$$

Intuitively, the correction states that if the variable `y` is modified in the body of either branch of the branching or in the body of the loop and `x` occurs in the expression, then there is a violation if $\ell(\texttt{y}) < \ell(\texttt{x})$ or $\ell(\texttt{y}) \perp \ell(\texttt{x})$.

As an example, let us use Fig. 3 to construct $\mathrm{Cr}(\texttt{w > x})_{\texttt{C}_1\texttt{;C}_2}$, e.g., `w > x`'s correction for $\texttt{C}_1\texttt{;C}_2$. Variables `w` and `x`, through the expression `w > x`, control the values of `w`, `x` and `z` since $\texttt{C}_1$ and $\texttt{C}_2$ set those values, and their execution depend on it, giving:

$$\begin{array}{c|cccc} & \texttt{w} & \texttt{x} & \texttt{y} & \texttt{z} \\ \hline \texttt{w} & \cdot & \text{🌢} & \cdot & \text{🌢} \\ \texttt{x} & \text{🌢} & \cdot & \cdot & \text{🌢} \\ \texttt{y} & \cdot & \cdot & \cdot & \cdot \\ \texttt{z} & \cdot & \cdot & \cdot & \cdot \end{array}$$

Observe also that in $\mathrm{Cr}(\texttt{t[i] != x})_{\texttt{C}}$ the variables `t`, `i` and `x` would be marked as controlling the variables occurring in $\mathrm{Out}(\texttt{C})$. However, no constraint would be imposed between the classes of `t`, `i` and `x`, since they would all be required to flow into classes that are higher or equal to theirs.

3.3.4 Conditionals and Loops

Following our previous observation, branchings and loops are interpreted similarly.

▶ **Definition 10** (Branching). *We let* $\mathbb{M}($**if** `e` **then** $\texttt{C}_1$ **else** $\texttt{C}_2)$ *be* $\mathbb{M}(\texttt{C}_1\texttt{;C}_2) + \mathrm{Cr}(\texttt{e})_{\texttt{C}_1\texttt{;C}_2}$.

Adding $\mathrm{Cr}(\texttt{w > x})_{\texttt{C}_1\texttt{;C}_2}$ to $\mathbb{M}(\texttt{C}_1) + \mathbb{M}(\texttt{C}_2)$ from Fig. 3, we obtain:

$$\mathbb{M}\left(\begin{array}{l}\textbf{if}\ \texttt{(w > x)} \\ \quad\textbf{then}\ \ \texttt{w = w + x;} \\ \qquad\qquad\ \texttt{z = y + 2} \\ \quad\textbf{else}\ \ \texttt{x = y * 2;} \\ \qquad\qquad\ \texttt{z = 0}\end{array}\right) = \begin{array}{c|cccc} & \texttt{w} & \texttt{x} & \texttt{y} & \texttt{z} \\ \hline \texttt{w} & \cdot & \text{🌢} & \cdot & \text{🌢} \\ \texttt{x} & \text{🌢} & \cdot & \cdot & \text{🌢} \\ \texttt{y} & \cdot & \text{🌢} & \cdot & \text{🌢} \\ \texttt{z} & \cdot & \cdot & \cdot & \cdot \end{array}.$$

Observe that there is a violation if $\ell(\texttt{w}) < \ell(\texttt{x})$ or $\ell(\texttt{w}) \perp \ell(\texttt{x})$ from the statement `w = w + x`, and that there is a violation if $\ell(\texttt{x}) < \ell(\texttt{w})$ or $\ell(\texttt{x}) \perp \ell(\texttt{w})$. The latter comes from the fact that the value of `w` will decide if `x = y * 2` will execute through the expression. To be free of violations, such a program must be given a class assignment satisfying $\ell(\texttt{w}) = \ell(\texttt{x})$ and the other constraints recorded in the matrix.

▶ **Definition 11** (Loop). *We let* $\mathbb{M}($**while** `e` **do** `C`$)$ *be* $\mathbb{M}(\texttt{C}) + \mathrm{Cr}(\texttt{e})_{\texttt{C}}$.

Since s1 and s2 do not control any other variable, their rows are all ·s. On the other hand, t, i and j control the values of s1, s2 and i, since they determine how many times the body will execute.

$$\mathbb{M}\left(\begin{array}{l}\texttt{while(t[i]!=j)\{}\\ \quad\texttt{s1[i] = j*j;}\\ \quad\texttt{s2[i] = 1/j;}\\ \quad\texttt{i++}\\ \texttt{\}}\end{array}\right) = \begin{array}{c|ccccc} & \texttt{t} & \texttt{i} & \texttt{j} & \texttt{s1} & \texttt{s2} \\ \hline \texttt{t} & \cdot & \text{🌢} & \cdot & \text{🌢} & \text{🌢} \\ \texttt{i} & \cdot & \cdot & \cdot & \text{🌢} & \text{🌢} \\ \texttt{j} & \cdot & \text{🌢} & \cdot & \text{🌢} & \text{🌢} \\ \texttt{s1} & \cdot & \cdot & \cdot & \cdot & \cdot \\ \texttt{s2} & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}.$$

4 Capturing Anytime Non-Interference

In the seminal work of Volpano et al. [29, pg. 173], "[s]oundness [wa]s formulated as a kind of noninterference property. [. . . I]f a variable v has security [class c], then one can change the initial values of any variables whose security levels are not dominated by [c], execute the program, and the *final value* of v will be the same, *provided the program terminates successfully*." (our emphasis). *Anytime non-interference*, defined below and captured by $\top^*_{\text{NI}}$, inspects the values *while the program is being executed*. It allows us to **1.** avoid making assumptions of program termination, or avoid waiting for termination, and **2.** model attackers who are capable of observing updates to variables at class c or lower.

First, we need to define a notion of *timed* execution, which captures the idea that an external observer can see updates on variables below a particular security class in "real time".

▶ **Definition 12** (Timed command execution)**.** *Given*

1. *a program* $\texttt{C}$ *with variables* $\texttt{x}_1, \dots, \texttt{x}_n$*,*
2. *a class assignment* $\ell : \text{Occ}(\texttt{C}) \to \text{SC}$*,*
3. *a security class* $c \in \text{SC}$*,*
4. *a* time (counter) $t \in \mathbb{N}$*,*
5. *and a value list* $\vec{v} = v_1, \dots, v_n$*,*

we write

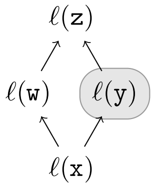
- $\texttt{C}[\vec{v}]_0$ *for the program* $\texttt{C}$ *where the variable* $\texttt{x}_i$ *was assigned* v_i*,*[3] *for* $1 \leqslant i \leqslant n$*,*
- $\texttt{C}[\vec{v} \to \vec{v'}]_t$ *if, while executing the commands in* $\texttt{C}[\vec{v}]_0$*,* $\texttt{x}_i$ *contains the value* v'_i*, for* $1 \leqslant i \leqslant n$ *after variables at class* c *or lower have been updated* t *times.*

If, after t *updates, variables at level* c *or lower stop being updated, then we let, for all* $t' > t$*,* $\texttt{C}[\vec{v} \to \vec{v'}]_t = \texttt{C}[\vec{v} \to \vec{v'}]_{t'}$*.*

Deciding whether variables will be updated after t may be difficult in all generality, but simple checks as e.g., testing for membership in $\text{Out}(\texttt{C}')$ for $\texttt{C}' \subseteq \texttt{C}$ the subprogram of $\texttt{C}$ that remains to be executed can give in some cases a rapid answer.

We give below a program along with a class assignment (where the security class c is grayed out) and two tables containing the value held by memory locations at time counter t. The initial value lists are $\vec{v_1} = \texttt{1,2,3,4}$ and $\vec{v_2} = \texttt{5,2,3,4}$. Observe that t is incremented only when values held by variables at or below c (with the grayed out background) are updated.

```
if  (w > x)
  then   w = w + x;
         z = y + 2
  else   x = y * 2;
         z = 0
```



t	w	x	y	z
0	1	2	3	4
1	1	6	3	4
1	1	6	3	0

t	w	x	y	z
0	5	2	3	4
0	7	2	3	4
0	7	2	3	5

Hence we have $\texttt{C}[\vec{v_1} \to \vec{v'_1}]_1$ and $\texttt{C}[\vec{v_2} \to \vec{v'_2}]_0 = \texttt{C}[\vec{v_2} \to \vec{v'_2}]_1$ for $\vec{v'_1} = \texttt{1,6,3,0}$ and $\vec{v'_2} = \texttt{7,2,3,5}$.

[3] Since arrays have a fixed size, we assume, for simplicity, that a variable $\texttt{x}_i$ representing an array of size s is given a value $v_i = v_i^1, \dots, v_i^s$.

▶ **Definition 13** (Up-to c equivalence)**.** *Given* $\mathtt{C}$*, a class assignment* $\ell : \mathrm{Occ}(\mathtt{C}) \to \mathrm{SC}$ *and* $c \in \mathrm{SC}$*, two values lists* $\vec{v}$ *and* $\vec{w}$ *are* up-to c equivalent*, written* $\vec{v} \overset{c}{\sim} \vec{w}$ *if* $\ell(\mathtt{x}_i) \leqslant c \implies v_i = w_i$*.*

Intuitively, two value lists are up-to c equivalent if they agree on the values of the variables of class c or lower: re-using the example above, we have $\vec{v_1} \overset{\ell(\mathtt{y})}{\sim} \vec{v_2}$ but $\vec{v_1} \overset{\ell(\mathtt{w})}{\not\sim} \vec{v_2}$.

We can now formally state the anytime non-interference property:

▶ **Definition 14** (Anytime non-interference)**.** *A program* $\mathtt{C}$ *is* anytime non-interfering for $\ell : \mathrm{Occ}(\mathtt{C}) \to \mathrm{SC}$ *if for all security class* $c \in \mathrm{SC}$*, for all time* $t \in \mathbb{N}$ *and all* $\vec{v}$ *and* $\vec{w}$*,*

$$\vec{v} \overset{c}{\sim} \vec{w}, \mathtt{C}[\vec{v} \to \vec{v'}]_t, \mathtt{C}[\vec{w} \to \vec{w'}]_t \implies \vec{v'} \overset{c}{\sim} \vec{w'}.$$

Note that we can conclude that the program above is *not* anytime non-interfering for the given class assignment, since $\vec{v_1} \overset{\ell(\mathtt{y})}{\sim} \vec{v_2}$ but $\vec{v_1'} \overset{\ell(\mathtt{y})}{\not\sim} \vec{v_2'}$: we had already noted, using $\top^*_{\mathrm{NI}}$, that $\ell(\mathtt{w}) = \ell(\mathtt{x})$ was required for this program to be anytime non-interfering. Indeed, this property has a natural equivalent in $\top^*_{\mathrm{NI}}$, and can be established using it:

▶ **Definition 15** (Non-interfering class assignment)**.** *Given* $\mathbb{M}(\mathtt{C})$*, a class assignment* ℓ *is* anytime non-interfering for $\mathtt{C}$ *iff* $\mathtt{C}$ *has no violation (Def. 5).*

Note that the trivial class assignment i that assigns to all values the same security class c_i is always non-interfering, since in that case $i(\mathtt{y}) < i(\mathtt{x})$ and $i(\mathtt{y}) \perp i(\mathtt{x})$ are always false. Conversely, any program is anytime non-interfering for i, since value lists are up-to c_i equivalent if and only if they are equal.

▶ **Theorem 16** (Correspondance)**.** *A program* $\mathtt{C}$ *is anytime non-interfering for* ℓ *(Def. 14) if and only if* ℓ *is anytime non-interfering for* $\mathtt{C}$ *(Def. 15).*

The proof is detailed in Appendix A, it leverages the idea that only assignments and corrections can introduce 🌢 in security-flow matrices. This mirror the idea that only assignments, loops and branchings can trigger the update of an element in a value list, hence connecting the two definitions of anytime non-interference. An important assumption is that expressions are falsifiable, e.g., that if $\mathtt{x}_i \in \mathrm{Occ}(\mathtt{e})$, then there exists at least one value for $\mathtt{v}_i$ that will make $\mathtt{e}$ evaluate to `false`, and at least one value that will make it evaluate to `true`.

5 Interpreting Function Calls in an Anytime Non-Interfering Context

We detail below how function calls can be integrated into our analysis. The main challenge is to nail down the correct interpretation of anytime non-interference for functions that may have side effects or, conversely, that may return a value with a lower class than its inputs. We start, as a warm-up, by discussing how to add to $\top^*_{\mathrm{NI}}$ pure functions, then functions with side effects, before finally discussing the meaning of anytime non-interference for functions.

5.1 Warm-Up: Pure Functions

First, let us discuss how *pure* functions can be integrated into $\top^*_{\mathrm{NI}}$. The first steps are to add $\mathit{fun}(\mathit{exp}, \ldots, \mathit{exp})$ to the expressions, let $\mathtt{f}$ and $\mathtt{g}$ range over function, and to let $\mathrm{Occ}(\mathtt{f(e_1,\ldots,e_n)}) = \mathtt{f}_{\vec{\mathtt{e}}}$ for $\mathtt{f}_{\vec{\mathtt{e}}}$ a freshly introduced variable unique to $\mathtt{f}$, $\mathtt{e}_1$, …, $\mathtt{e}_n$.[4] Let us illustrate those first steps by interpreting two programs involving function calls. Simply using the definition of $\mathrm{Occ}(\mathtt{f(e_1,\ldots,e_n)})$, and without changing Definitions 6 or 10, we have:

[4] This point is clarified at the end of this subsection.

`C`	Out(`C`), In(`C`)	$\mathbb{M}(\texttt{C})$
`x = f(y)`	$\text{Out}(\texttt{C}) = \{\texttt{x}\}$ $\text{In}(\texttt{C}) = \{\texttt{f}_\texttt{y}\}$	$\begin{array}{c@{}c} & \begin{array}{ccc} \texttt{x} & \texttt{y} & \texttt{f}_\texttt{y} \end{array} \\ \begin{array}{c} \texttt{x} \\ \texttt{y} \\ \texttt{f}_\texttt{y} \end{array} & \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \bullet & \cdot & \cdot \end{pmatrix} \end{array}$

`if (g(x, y) > x)`
`  then y = z`
`  else skip`
	$\text{Out}(\texttt{C}) = \{\texttt{y}\}$
	$\text{In}(\texttt{C}) = \{\texttt{g}_{\texttt{x},\texttt{y}}, \texttt{x}, \texttt{z}\}$
		$\begin{array}{c@{}c} & \begin{array}{cccc} \texttt{x} & \texttt{y} & \texttt{z} & \texttt{g}_{\texttt{x},\texttt{y}} \end{array} \\ \begin{array}{c} \texttt{x} \\ \texttt{y} \\ \texttt{z} \\ \texttt{g}_{\texttt{x},\texttt{y}} \end{array} & \begin{pmatrix} \cdot & \bullet & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \bullet & \cdot & \cdot \\ \cdot & \bullet & \cdot & \cdot \end{pmatrix} \end{array}$

It may seem surprising that `y` does not occur in the In(`C`) sets, considering that its value may affect the output of the function call, hence controlling indirectly the out-variables. This design choice *lets the class assignment handle this decision.* The core idea is that the level assignment $\ell : \text{Occ}(\texttt{C}) \to \text{SC}$ now additionally needs to assign a class (or a collection of constraints) to each $\texttt{f}_{\vec{\texttt{e}}}$ variable. Multiple design choices exist, e.g.,

- $\ell(\texttt{f}_{\vec{\texttt{e}}})$ can be a constant class c, reflecting the fact that all function outputs should be assigned the same security class regardless of the classes assigned to its inputs,
- $\ell(\texttt{f}_{\vec{\texttt{e}}})$ can be a function of $\ell(\texttt{x})$ for $\texttt{x} \in \text{Occ}(\texttt{e}_1) \cup \cdots \cup \text{Occ}(\texttt{e}_n)$ such as the supremum, the infimum (written max and min), the first projection π_1, etc.
- $\ell(\texttt{f}_{\vec{\texttt{e}}})$ could otherwise depends on the particular structure of $\vec{\texttt{e}}$, e.g., be the supremum if a variable whose class is above a particular threshold occurs, a constant otherwise, etc.

This adds "external" constraints to our definition of violation: *in addition* of having to provide a level assignment meeting Def. 5's condition, one has to check that the constraints given on the classes of the $\texttt{f}_{\vec{\texttt{e}}}$ variables are met. As an additional benefit, this allows to handle functions with 0 parameters, since the flow from the argument(s, or lack thereof) to the function's output need not to be tracked in the security-flow matrix.

Going back to our first example above, if we consider that `f`'s output class must be strictly higher than its input class, then we have the additional requirement that $\ell(\texttt{f}_\texttt{x}) > \ell(\texttt{x})$ for each variable `x` such that $\texttt{f}_\texttt{x}$ occurs in $\mathbb{M}(\texttt{C})$. Hence, our first program `C` would be free of violations with

$$\begin{tikzcd} \ell(\texttt{x}) \\ \ell(\texttt{f}_\texttt{y}) \arrow[u] \\ \ell(\texttt{y}) \arrow[u] \end{tikzcd}$$

and

$$\begin{tikzcd} \ell(\texttt{f}_\texttt{y}) = \ell(\texttt{x}) \\ \ell(\texttt{y}) \arrow[u] \end{tikzcd}$$

, but it would have a violation with

$$\begin{tikzcd} & \ell(x) & \\ \ell(\texttt{y}) \arrow[ur] & & \ell(\texttt{f}_\texttt{y}) \arrow[ul] \\ & c \arrow[ul] \arrow[ur] & \end{tikzcd}$$

, even if this latter class assignment would have met the requirements of Def. 5.

The rest of the interpretation is the same, even $\mathbb{M}(\texttt{C})$ remains a $|\text{Occ}(\texttt{C})| \times |\text{Occ}(\texttt{C})|$ matrix, since $\texttt{f}_{\vec{\texttt{e}}}$ variables are defined as occurring in `C`. The only tedious aspect is to handle the introduction of $\texttt{f}_{\vec{\texttt{e}}}$ variables elegantly. One would want e.g.,

$$\text{Occ}(\texttt{f(x + y)}) = \text{Occ}(\texttt{f(x - y)}) \quad \text{and} \quad \text{Occ}(\texttt{f(x + 3)}) = \text{Occ}(\texttt{f(x - 5)})$$

but relying on *the set* $\text{Occ}(\texttt{e}_1) \cup \cdots \cup \text{Occ}(\texttt{e}_n)$ would not be correct, as one would want e.g.,

$$\text{Occ}(\texttt{f(x, y)}) \neq \text{Occ}(\texttt{f(y, x)}) \quad \text{and} \quad \text{Occ}(\texttt{f(x, x, y)}) \neq \text{Occ}(\texttt{f(x, y, y)}).$$

A more precise definition of function type signature, capable of handling repetition and swapping in the argument list, would be required but presents no challenge.

C	$\mathbb{M}^{e}(\mathtt{C}) = \cdots$	Out(C), In(C)	$\mathbb{M}^{e}(\mathtt{C})$
`g(x + y)`	$\mathbb{M}(\mathtt{g}^{\mathtt{out}}_{\mathtt{x,y}}\mathtt{= x + y})$ $+\mathbb{M}(\mathtt{skip})$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{g}^{\mathtt{out}}_{\mathtt{x,y}}\}$ $\mathrm{In}(\mathtt{C}) = \{\mathtt{x}, \mathtt{y}\}$	$\begin{array}{c} \\ \mathtt{x} \\ \mathtt{y} \\ \mathtt{g}^{\mathtt{in}}_{\mathtt{x,y}} \end{array}\begin{array}{c} \begin{array}{ccc} \mathtt{x} & \mathtt{y} & \mathtt{g}^{\mathtt{out}}_{\mathtt{x,y}} \end{array} \\ \begin{pmatrix} \cdot & \cdot & \blacklozenge \\ \cdot & \cdot & \blacklozenge \\ \cdot & \cdot & \cdot \end{pmatrix} \end{array}$
`x = f(y)`	$\mathbb{M}(\mathtt{f}^{\mathtt{out}}_{\mathtt{y}}\mathtt{= y})$ $+\mathbb{M}(\mathtt{x = f(y)})$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{x}, \mathtt{f}^{\mathtt{out}}_{\mathtt{y}}\}$ $\mathrm{In}(\mathtt{C}) = \{\mathtt{y}, \mathtt{f}^{\mathtt{in}}_{\mathtt{y}}\}$	$\begin{array}{c} \\ \mathtt{x} \\ \mathtt{y} \\ \mathtt{f}^{\mathtt{in}}_{\mathtt{y}} \end{array}\begin{array}{c} \begin{array}{ccc} \mathtt{x} & \mathtt{y} & \mathtt{f}^{\mathtt{out}}_{\mathtt{y}} \end{array} \\ \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \blacklozenge \\ \blacklozenge & \cdot & \cdot \end{pmatrix} \end{array}$

`if(g()==1)`
`  then x = 0`
`  else skip`
	$\mathrm{Cr}(\mathtt{g()==1})_{\mathtt{x=0;skip}}$
	$+\mathbb{M}(\mathtt{x = 0})$
	$+\mathbb{M}(\mathtt{g}^{\mathtt{out}}_{\emptyset}\mathtt{=}) + \mathbb{M}(\mathtt{skip})$
		$\mathrm{Out}(\mathtt{C}) = \{\mathtt{x}, \mathtt{g}^{\mathtt{out}}_{\emptyset}\}$
		$\mathrm{In}(\mathtt{C}) = \emptyset$
			$\begin{array}{c} \\ \mathtt{x} \\ \mathtt{g}^{\mathtt{in}}_{\emptyset} \end{array}\begin{array}{c} \begin{array}{cc} \mathtt{x} & \mathtt{g}^{\mathtt{out}}_{\emptyset} \end{array} \\ \begin{pmatrix} \cdot & \cdot \\ \blacklozenge & \blacklozenge \end{pmatrix} \end{array}$

Figure 4 Statement Examples, Interpretation and Sets – Involving Effects

5.2 Completing the Picture: Functions With Side Effects

Our development so far assumes that the only way a function can leak information is through its return value. Considering functions with side effects (such as `print`, `read`, accessing a non-local variable, passing argument by reference, etc.) increases $\top^{*}_{\mathrm{NI}}$'s expressivity, but requires to discuss more precisely what is meant by anytime non-interfering function calls.

We first focus on how effects can be integrated into $\top^{*}_{\mathrm{NI}}$. Interestingly, the column $\mathtt{f}_{\vec{\mathtt{e}}}$ always remains empty, since the variale $\mathtt{f}_{\vec{\mathtt{e}}}$ will never be in an Out set (it never "receives" a value). Hence, we can use its out-variable to store information about its possible side-effects. To this end, we now "split" $\mathtt{f}_{\vec{\mathtt{e}}}$ into $\mathtt{f}^{\mathtt{in}}_{\vec{\mathtt{e}}}$ (on the row) and $\mathtt{f}^{\mathtt{out}}_{\vec{\mathtt{e}}}$ (on the column) for each "signature" $\mathtt{f(e}_1\mathtt{,...,e}_n\mathtt{)}$. This convention is illustrated in Ex. 22 with another example of *pure* functions.

To account for effects, the expression $\mathit{fun}(\mathit{exp}, \ldots, \mathit{exp})$ should now be treated as a *command* and *an expression*. We then edit the definition of the variables occuring in $\mathtt{f(e}_1\mathtt{,...,e}_n\mathtt{)}$ (henceforth simply denoted $\mathtt{f(}\vec{\mathtt{e}}\mathtt{)}$) and in $\vec{\mathtt{e}}$ to get a more complete picture:

$$\mathrm{Occ}(\mathtt{f(}\vec{\mathtt{e}}\mathtt{)}) = \mathtt{f}^{\mathtt{in}}_{\vec{\mathtt{e}}} \qquad\qquad \mathrm{Occ}(\vec{\mathtt{e}}) = \begin{cases} \mathrm{Occ}(\mathtt{e}_1) \cup \cdots \cup \mathrm{Occ}(\mathtt{e}_n) & \text{if } n > 0 \\ \mathtt{f}^{\mathtt{in}}_{\emptyset} & \text{otherwise} \end{cases}$$

with the "otherwise" case above handling functions with 0 parameters. We then let

$$\mathbb{M}(\mathtt{f(}\vec{\mathtt{e}}\mathtt{)}) = \mathbb{M}(\mathtt{skip}) \qquad \mathbb{M}^{e}(\mathtt{C}) = \sum_{\mathtt{f(}\vec{\mathtt{e}}\mathtt{)} \subseteq \mathrm{Occ}(\mathtt{C})} \mathbb{M}(\mathtt{f}^{\mathtt{out}}_{\vec{\mathtt{e}}}\mathtt{=}\ \vec{\mathtt{e}}) + \mathbb{M}(\mathtt{C})$$

and study $\mathbb{M}^{e}(\mathtt{C})$ moving forward, where the above definition interprets all function calls as assignments and then interpret the rest of the program as before, skipping the commands calling a function without using their return value. Fig. 4 gathers examples of programs involving effectul functions. The last example follows our definitions, but may be hard to unpack: the critical point is to see that $\mathtt{g}^{\mathtt{in}}_{\emptyset}$ controlling the values of `x` and $\mathtt{g}^{\mathtt{out}}_{\emptyset}$ reflects the fact that `g` "on its own" (i.e., without any input) will decide of its output and hence if `x = 0` will execute.

This interpretation entails the following two principles:

- An effectful function is completely transparent: the first example of Fig. 4 requires $\ell(\mathtt{g}^{\mathtt{out}}_{\mathtt{x,y}}) \geqslant \max(\ell(\mathtt{y}), \ell(\mathtt{x}))$, e.g., as if `g` is revealing all the data it is processing.

- A function can nevertheless have $\ell(\mathtt{f}^{\mathtt{in}}_{\vec{\mathtt{e}}})$ be less than or orthogonal to the level of its arguments. This means that a function can have a return value that is independent of its arguments, e.g., a `success` or `failure` code in displaying the arguments at the screen.

Those principles can be both desirable and are not incompatible. Indeed, a program

```
success = print(secret);
  if(success==0) then x++
  else x--
```

with the class assignment

$$\begin{tikzcd}
\ell(\mathtt{secret}) = \ell(\mathtt{print}^{\mathtt{out}}_{\mathtt{secret}}) \\
\ell(\mathtt{x}) \arrow[u] \\
\ell(\mathtt{success}) = \ell(\mathtt{print}^{\mathtt{in}}_{\mathtt{secret}}) \arrow[u]
\end{tikzcd}$$

should be considered as anytime non-interfering: a user having access to `secret`'s class can see its value be displayed on the screen, but an attacker having access to at most `x`'s class cannot infer `secret`'s value, despite being able to access the return value of `print(secret)`—which is constantly set to the lowest class available. Three challenges remain:

- The intuitive reading of our security-flow matrices is lost. For example, since $\mathtt{y} \notin \mathrm{In}(\mathtt{x = f(y)})$, $\mathbb{M}^{\mathrm{e}}(\mathtt{x = f(y)})(\mathtt{x}, \mathtt{y}) \neq$ 🌢, and `y` is not recorded as impacting the value of `x`. This design choice is a *feature*, as it does not "force" `y` to control `x`'s value when processed through `f`: this allows finer constraints on the level of $\mathtt{f}^{\mathtt{in}}_{\mathtt{y}}$.
- It rigidly assumes that functions with side effects will reveal all their data at all times. This conservative approach is also a feature, but could be tuned by refining how constraints for $\mathtt{f}^{\mathtt{out}}_{\vec{\mathtt{e}}}$ class assignments are recorded.
- Developing a definition of anytime non-interference for programs with side effects (Def. 14) will requires to develop a notion of external observer and of contextual equivalences, to correctly account for side effects and multiple communication channels.

We believe that those issues can be addressed by developing a richer theory that incorporates *external knowledge on functions*, but reserve it for futur work.

6 Practical Applications and Comparison

6.1 Implementing the Anytime Non-interference Logic

We have created a prototype static analyzer TYNI that implements the $\top^*_{\mathrm{NI}}$ logic. The way matrices are composed in $\top^*_{\mathrm{NI}}$ is a key feature in making the analysis straightforward and efficient in practice. Because composition order is irrelevant (Ex. 18), it suffices to represent matrices as hash maps where composition is a union of maps.

The TYNI analyzer accepts as input a program file written in `Java`. It first translates the program into a parse tree (without optimizations), then analyzes the tree based on the rules of $\top^*_{\mathrm{NI}}$. The analysis is recursive over the methods of a `Java` class. Obtaining a sound result requires a `Java` method fully expressible in the $\top^*_{\mathrm{NI}}$ grammar (Fig. 1). Commands that are not covered are highlighted by TYNI and a partial result is given. This handling assists the continued development of $\top^*_{\mathrm{NI}}$, that already handles all the examples from Appendix B.

The outlined engineering choices have multiple advantages. `Java` is frequently used to implement taint analyzers—an instance of non-interference fixed to two security classes—thus preparing TYNI for a similar use case. Since `Java` compiles into bytecode, a kind of intermediate stack language, it enables program analysis at multiple language representations. Although compiler optimizations could reduce the rate of false alarms, e.g., by eliminating dead-code, it would artificially inflate the analysis precision and thus we prefer our strategy. Currently, TYNI produces security-flow matrices for input programs. The security flow matrices serve as basis for the extended applications, including the directions presented next.

Preservation of anytime non-interference. The $\top^*_{\text{NI}}$ logic does not require much language structure; in particular, it assumes no language-specific syntactic features. It is possible to map its grammar to numerous language representations, including intermediate representations and bytecode. Comparing security-flow matrices of the same program at different representations enables analyzing preservation of security properties and detecting compilation issues.

Security class inference. When security classes of variables are known partially, it is possible to infer them for all variables. The inference requires a security flow-matrix, an information flow policy, and the known class assignments. The inference is then framed as a satisfiability problem. If a satisfactory assignment exists, it provides the security classes for all variables. This application is similar to type inference, but requires no program as input. Further, the same security flow-matrix can be easily evaluated against different information flow policies.

Taint analysis. Taint analysis detects information flow issues between a high source and a low sink. Aside a program, the sources and sinks are necessary, and analyzers commonly assume them as inputs. The analyzers then compete on precision along various axes: path-coverage, syntax-coverage, context-sensitivity, false alarm rate, etc. Taint analysis can be formulated with security flow-matrices by analyzing source to sink connectivity.

6.2 Circumventing Termination-insensitivity via Distribution

Several real-world programs are non-terminating by design: web servers, embedded systems, and cyber-physical systems are among the examples. While the programs can terminate, the termination events are infrequent and uncharacteristic of standard behavior. Security analyses that can handle absence of termination are necessary to support such programs.

Anytime non-interference is termination-insensitive and compatible with the study of non-terminating programs. However, termination-insensitivity is too weak to guard against untrusted code, e.g., execution of the `eval` command. To offer an alleviation strategy, we present an approach to distribute security-sensitive computation. This way, computations that require elevated security checks are handled separately from trusted code.

The idea is to pair $\top^*_{\text{NI}}$ with a *distribution analysis* [3] that detects disjoint program fragments. That program fragments are disjoint means there exists no exchange of variable data between program fragments. The judgement of disjointness is derived via a sound data-flow analysis that guarantees the property. It is then permissible to execute the program fragments in separate execution contexts. Although the distribution analysis naturally fits parallel computations, it is not restricted to this use case. We conjecture it offers broader utility here in ensuring program security.

To combine the two analysis, we first analyze a program with $\top^*_{\text{NI}}$ to identify its information flow constraints. Then, we use the distribution analysis to identify the program's distribution potential. Merging the results, the disjoint program fragments are assigned appropriate security classes. The fragments are then allocated to different execution contexts, where each fragment can have a different security class designation. This way, a program must not adhere to a monolithic security strategy, but can have a finer-grained strategy based on its content. Due to paper scope, we reserve a detailed treatment for an extended version.

6.3 Overview of Alternative and Related Approaches

In language-based security, non-interference is commonly achieved through security types systems. Type theoretic non-interference provides strong end-to-end confidentiality guarantees in a static and scalable way. Initiating from the seminal work of Volpano et al. [29], security type systems have been extended to consider non-interference under numerous paradigms, including concurrency [30, 13, 14], formal reasoning [24, 14], compilation [5], etc. A major challenge among the security type systems is *declassification*, a kind of security downgrading operation [10]. A *downgrading* mechanisms permits elevating the security judgement around control-flow constructs then lowering it afterward. In other words, information is allowed to flow contrary to the policy [10]. The mechanism is necessary to increase the expressive power of security type systems; however, downgrading is generally not safe [13] and eliminates the strong compositional guarantees of non-interference [10]. In practice, security type systems are challenging to use because they modify the programming language. A program must be annotated with security types and compiled with non-standard tools that can enforce the types [20]. There is also a stark contrast in expressiveness of theoretical and practical systems; e.g., [18] categorically excludes implicit flows.

The Dependency Core Calculus (DCC) [1] is conceptually related to $\top^*_{\text{NI}}$. The DCC is an extension of lambda calculus, framed around the notion of data dependency, of which non-interference is an instance. Though similarly rooted in dependency analysis, $\top^*_{\text{NI}}$ originates from works of implicit computational complexity (ICC). It is a refinement of [23, 3], but $\top^*_{\text{NI}}$ required significant adjustment, particularly around matrix composition and functions.

ICC studies machine-free characterizations of complexity classes by introducing *restrictions* in programming languages that in turn guarantee semantic properties [11]. A critical idea is that ICC techniques can benefit from, and offer support in, other analytic domains. The use of ICC techniques is such extended ways is an emerging research topic. Previously, a non-interference type system provided a foundation for a series of complexity-theoretic results [21, 16]. In the opposite direction, an ICC system was applied to cryptographic proofs [4]. Although $\top^*_{\text{NI}}$ has transformed from its origins to not enforce complexity bounds, it reinforces the bidirectional connection between ICC and language-based security.

7 Conclusion: Strengths, Limitations and Future Directions

Anytime non-interference detects violations at any program point, enforcing a finer-grained security policy than classic non-interference that is defined in terms of inputs and outputs. We have presented $\top^*_{\text{NI}}$, a sound and compositional program logic, that captures the semantic security property of anytime non-interference in imperative programs. The logic assigns security flow matrices to commands where the matrices represent the program's potentially interfering information flows. The logic is lightweight and does not require program annotations, specialty compilers, and adds no run-time overhead. Beside the compelling theory, $\top^*_{\text{NI}}$ can be implemented to obtain automated security analysis in practice. We have constructed a prototype static analyzer TYNI to analyze `Java` programs. By extension, TYNI can support a range of applications, e.g., security class inference, taint analysis, and preservation analysis.

Although the utility of $\top^*_{\text{NI}}$ is encouraging, the development is still mainly theoretical. Our immediate priority is enriching the syntax with effectful functions and object oriented constructs. For additional strength, we hope to mechanize the theory. On the practical side, the prototype analyzer has room for enhancements. It already computes security flow matrices, but an extension to the applications requires additional engineering steps. With the current syntax coverage, experimental comparisons are still out of scope. In the meantime, $\top^*_{\text{NI}}$ provides an promising avenue for security analysis and future enhancements.

References

1 Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 147–160. ACM, 1999. doi:10.1145/292540.292555.

2 Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. mwp-analysis improvement and implementation: Realizing implicit computational complexity. In *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *LIPIcs*, pages 26:1–26:23. Schloss Dagstuhl, 2022. doi:10.4230/LIPIcs.FSCD.2022.26.

3 Clément Aubert, Thomas Rubiano, Neea Rusch, and Thomas Seiller. Distributing and parallelizing non-canonical loops. In *Verification, Model Checking, and Abstract Interpretation*, volume 13881 of *LNCS*, pages 1–24. Springer, 2023. doi:10.1007/978-3-031-24950-1_1.

4 Patrick Baillot, Gilles Barthe, and Ugo Dal Lago. Implicit computational complexity of subrecursive definitions and applications to cryptographic proofs. *J. Autom. Reason*, 63(4):813–855, 2019. doi:10.1007/s10817-019-09530-2.

5 Gilles Barthe, Amitabh Basu, and Tamara Rezk. Security types preserving compilation. In *Verification, Model Checking, and Abstract Interpretation*, volume 2937 of *LNCS*, pages 2–15. Springer, 2004. doi:10.1007/978-3-540-24622-0_2.

6 Jason Bau and John C. Mitchell. Security modeling and analysis. *IEEE Security & Privacy Magazine*, 9(3):18–25, 2011. doi:10.1109/msp.2011.2.

7 Johan Bay and Aslan Askarov. Reconciling progress-insensitive noninterference and declassification. In *2020 IEEE 33rd Computer Security Foundations Symposium (CSF)*, pages 95–106. IEEE, 2020. doi:10.1109/csf49147.2020.00015.

8 Marton Bognar, Jo Van Bulck, and Frank Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1638–1655. IEEE, 2022. doi:10.1109/sp46214.2022.9833735.

9 Annalisa Bossi, Damiano Macedonio, Carla Piazza, and Sabina Rossi. Information flow in secure contexts. *JCS*, 13(3):391–422, 2005. doi:10.3233/jcs-2005-13303.

10 Ethan Cecchetti, Andrew C. Myers, and Owen Arden. Nonmalleable information flow control. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS'17, pages 1875–1891. ACM, 2017. doi:10.1145/3133956.3134054.

11 Ugo Dal Lago. A short introduction to implicit computational complexity. In Nick Bezhanishvili and Valentin Goranko, editors, *ESSLLI*, volume 7388 of *LNCS*, pages 89–109. Springer, 2011. doi:10.1007/978-3-642-31485-8_3.

12 Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, 1976. doi:10.1145/360051.360056.

13 Farzaneh Derakhshan, Stephanie Balzer, and Yue Yao. Regrading Policies for Flexible Information Flow Control in Session-Typed Concurrency. In *38th European Conference on Object-Oriented Programming (ECOOP 2024)*, volume 313 of *LIPIcs*, pages 11:1–11:29. Schloss Dagstuhl, 2024. doi:10.4230/LIPIcs.ECOOP.2024.11.

14 Dan Frumin, Robbert Krebbers, and Lars Birkedal. Compositional non-interference for fine-grained concurrent programs. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1433. IEEE, 2021. doi:10.1109/sp40001.2021.00003.

15 Joseph A Goguen and José Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–20. IEEE, 1982. doi:10.1109/SP.1982.10014.

16 Emmanuel Hainry and Romain Péchoux. A general noninterference policy for polynomial time. *Proc. ACM Program. Lang.*, 7(POPL):806–832, 2023. doi:10.1145/3571221.

17 Daniel Hedin and Andrei Sabelfeld. A perspective on information-flow control. In *Software safety and security*, pages 319–347. IOS Press, 2012. doi:10.3233/978-1-61499-028-4-319.

18 Wei Huang, Yao Dong, and Ana Milanova. Type-based taint analysis for java web applications. In *Fundamental Approaches to Software Engineering*, volume 8411 of *LNCS*, pages 140–154. Springer, 2014. doi:10.1007/978-3-642-54804-8_10.

19 Neil D. Jones and Lars Kristiansen. A flow calculus of *mwp*-bounds for complexity analysis. *ACM Trans. Comput. Log.*, 10(4):28:1–28:41, 2009. doi:10.1145/1555746.1555752.

20 Ada Lamba, Max Taylor, Vincent Beardsley, Jacob Bambeck, Michael D. Bond, and Zhiqiang Lin. Cocoon: Static information flow control in rust. *Proc. ACM Program. Lang.*, 8(OOPSLA1):166–193, 2024. doi:10.1145/3649817.

21 Jean-Yves Marion. A type system for complexity flow analysis. In *2011 IEEE 26th Annual Symposium on Logic in Computer Science*, pages 123–132. IEEE, 2011. doi:10.1109/LICS.2011.41.

22 Bishop Matt. *Computer security: art and science.* Addison-Wesley Professional, 2019.

23 Jean-Yves Moyen, Thomas Rubiano, and Thomas Seiller. Loop quasi-invariant chunk detection. In *Automated Technology for Verification and Analysis*, volume 10482 of *LNCS*. Springer, 2017. doi:10.1007/978-3-319-68167-2_7.

24 Luke Nelson, James Bornholt, Arvind Krishnamurthy, Emina Torlak, and Xi Wang. Noninterference specifications for secure systems. *ACM SIGOPS Oper. Syst. Rev.*, 54(1):31–39, 2020. doi:10.1145/3421473.3421478.

25 Louis-Noel Pouchet and Tomofumi Yuki. PolyBench/C 4.2. Accessed: 22 February 2025. URL: https://sourceforge.net/projects/polybench/files/.

26 Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Trans. Am. Math. Soc.*, 74(2):358–366, 1953. doi:10.1090/S0002-9947-1953-0053041-6.

27 Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE J. Sel. Areas Commun.*, 21(1):5–19, 2003. doi:10.1109/JSAC.2002.806121.

28 Fred B. Schneider, Greg Morrisett, and Robert Harper. *A language-based approach to security*, volume 2000 of *LNCS*, pages 86–101. Springer, 2001. doi:10.1007/3-540-44577-3_6.

29 Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *JCS*, 4(2/3):167–188, 1996. doi:10.3233/JCS-1996-42-304.

30 Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. In *Proceedings. 11th IEEE Computer Security Foundations Workshop (Cat. No.98TB100238)*, CSFW-98, pages 34–43. IEEE Comput. Soc. doi:10.1109/csfw.1998.683153.

A Proof of Thm. 16

A first useful observation is that if $\mathtt{C}' \subseteq \mathtt{C}$, then $\mathbb{M}(\mathtt{C}')$ is included in $\mathbb{M}(\mathtt{C})$, in the sense that $\mathbb{M}(\mathtt{C}')(\mathtt{x},\mathtt{y}) =$ 💧 $\implies$ $\mathbb{M}(\mathtt{C})(\mathtt{x},\mathtt{y}) =$ 💧. This simple observation comes from our "additive" interpretation of commands, and is useful in proving our theorem. One should also note that if ℓ is not anytime non-interfering for C', then any class assignment extending ℓ to $\mathrm{Occ}(\mathtt{C})$ is not anytime non-interfering for C.

▶ **Theorem 16** (Correspondance). *A program* C *is anytime non-interfering for ℓ (Def. 14) if and only if ℓ is anytime non-interfering for* C *(Def. 15).*

Proof. Let us assume given C, $\ell : \mathrm{Occ}(\mathtt{C}) \to \mathrm{SC}$, and that $\mathbb{M}(\mathtt{C})$ has been computed.

For the if part Suppose that ℓ is anytime non-interfering for C, but that C is not anytime non-interfering for ℓ. Then there must exist a class $c \in \mathrm{SC}$ and a counter t such that for some $\vec{v}$ and $\vec{w}'$,

$$\vec{v} \overset{c}{\sim} \vec{w} \qquad (1) \qquad\qquad \mathtt{C}[\vec{w} \to \vec{w}']_t \qquad (3)$$

$$\mathtt{C}[\vec{v} \to \vec{v}']_t \qquad (2) \qquad\qquad \vec{v}' \overset{c}{\nsim} \vec{w}' \qquad (4)$$

For $\overset{c}{\nsim}$ the negation of $\overset{c}{\sim}$, i.e., there must exists $\mathtt{x}_i$ such that

$$\ell(\mathtt{x}_i) \leqslant c \qquad (5) \qquad\qquad v'_i \neq w'_i \qquad (6)$$

For Equation 6 to hold, it must be the case that C contains a statement of the form $\mathtt{x}_i\texttt{=}\ \mathtt{e}_1$,[5] possibly guarded by `while` and `if` statements using the expressions $\mathtt{e}_2, \dots, \mathtt{e}_n$. Let $\mathtt{x}_1, \dots, \mathtt{x}_m = \bigcup_{j=1}^{n} \mathrm{Occ}(\mathtt{e}_j)$, and observe that since by Equation 1 our input value lists are up-to c equivalent, it must be the case that there exists $j \in \{1, \dots, m\}$ such that

$$\ell(\mathtt{x}_j) > c \text{ or } \ell(\mathtt{x}_j) \perp c, \qquad (7)$$

otherwise Equation 6 could not hold.[6] Furthermore, thanks to Equation 5 we know that

$$j \neq i. \qquad (8)$$

Let $\mathtt{C}'$ be the smallest sub-program of C where $\mathtt{x}_i\texttt{=}\ \mathtt{e}_1$ occurs and either $\mathtt{x}_j \in \mathrm{Occ}(\mathtt{e}_1)$ or $\mathtt{x}_j$ occurs in the condition of a `while` or `if` command guarding the command $\mathtt{x}_i\texttt{=}\ \mathtt{e}_1$. Intuitively, $\mathtt{C}'$ has one of the following forms:

```
xi = ··· xj ···;
```

Listing 1 (A)ssignment Case

```
while(··· xj ···){
  ...
  xi = ···;
  ...
}
```

Listing 2 (L)oop Case

```
if(··· xj ···){
  ...
  xi = ···;
  ...
}
```

Listing 3 (B)ranching Case

[5] Which can be $\mathtt{t[e}_1^1\mathtt{]}\ \texttt{=}\ \mathtt{e}_1^2$, in which case we let $\mathrm{Occ}(\mathtt{e}_1) = \mathrm{Occ}(\mathtt{e}_1^1) \cup \mathrm{Occ}(\mathtt{e}_1^2)$ and carry out the same reasoning.

[6] To be more rigorous, it could be the case that the classes of $\mathtt{x}_1, \dots, \mathtt{x}_m$ are c or below, but that *one of them is itself* impacted by a variable at a higher or incomparable class. To handle, this case, one simply replaces $\mathtt{x}_i$ and $\mathtt{x}_j$ with those "problematic" variables, decreases the counter t to when the value of the one with the lower class was changed, and carry out the same reasoning, possibly repeating this step again. Since t decreased, we are guaranteed to identify "the first" anytime non-interference violation and to reason about it.

Hence, $\mathtt{x}_j \in \mathrm{In}(\mathtt{C}')$, $\mathtt{x}_i \in \mathrm{Out}(\mathtt{C}')$, and inspecting the rules of our interpretation allows us to conclude that $\mathbb{M}(\mathtt{C})(\mathtt{x}_j, \mathtt{x}_i) = \text{🌢}$, since $\mathbb{M}(\mathtt{C}')$ is included in $\mathbb{M}(\mathtt{C})$.[7] Hence, $\mathtt{x}_j, \mathtt{x}_i \in \mathrm{Occ}(\mathtt{C})$ and $j \neq i$ by Equation 8, so we can use our assumption that ℓ is non-interfering for C to conclude that $\ell(\mathtt{x}_j) \leqslant \ell(\mathtt{x}_i)$–which, in conjunction with Equation 5, contradicts Equation 7.

For the only if part Let us assume that C is anytime non-interfering for ℓ, we need to prove that ℓ is anytime non-interfering for C e.g., that C has no violation: for all i, j,

$$\mathbb{M}(\mathtt{C})(\mathtt{x}_j, \mathtt{x}_i) = \text{🌢} \tag{9}$$

implies

$$\ell(\mathtt{x}_j) \leqslant \ell(\mathtt{x}_i). \tag{10}$$

Note that if $i = j$, then Equation 9 cannot hold since security-flow matrices are hollow, hence we only have to prove the $i \neq j$ case. We prove it below, factoring-in as previously the remarks about $\mathtt{x}_i$ possibly being an array and having to "chase down" the exact pair of variables violating anytime non-interference.

Equation 9 implies that there is a sub-program $\mathtt{C}'$ of C such that $\mathtt{x}_j \in \mathrm{In}(\mathtt{C}')$ and $\mathtt{x}_i \in \mathrm{Out}(\mathtt{C}')$.[8] By a reasoning similar to the previous case, it means that $\mathtt{C}'$ has one of the three forms (A), (L) or (B) presented in Listings 1–3.

Now, consider two values lists $\vec{v}$ and $\vec{w}$ that are up-to $\ell(\mathtt{x}_i)$ equivalent, and assume by contradiction that Equation 10 does not hold. It means that $\vec{v}$ and $\vec{w}$ can diverge on the value of v_j that gets attributed to $\mathtt{x}_j$, but that at any time counter t, we should have $\mathtt{C}'[\vec{v} \to \vec{v'}]_t, \mathtt{C}'[\vec{w} \to \vec{w'}]_t \implies \vec{v'} \overset{c}{\sim} \vec{w'}$. However, depending on the form of $\mathtt{C}'$, the value held by $\mathtt{x}_j$ will impact directly (A) or indirectly ((L), (B)) the value held by $\mathtt{x}_i$ at a particular time, or the number of time it is updated,[9] contradicting anytime non-interference of $\mathtt{C}'$ and hence of C. □

B Examples

To ease the presentation, we present the construction equations as inference rules, treating the inductive ones as inference rules with hypothesis, and the base cases (assignment, **skip**, but also computing the correction) as axioms.

▶ **Example 17** (Transitive information flow). Consider a program of two commands:

```
if (h==0) then y = 1 else skip  // C1
if (y==0) then z = 1 else y = z   // C2
```

Although no direct assignment exists from h to z, the variables are transitively dependent through y. The matrix labels are h y z but omitted for compactness. The derivation is

$$\dfrac{\dfrac{\dfrac{}{\texttt{y==1} : \begin{pmatrix}\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\end{pmatrix}}\,\text{Asgn} \quad \dfrac{}{\texttt{Skip} : \begin{pmatrix}\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\end{pmatrix}}\,\text{skip} \quad \dfrac{}{\texttt{h==0} : \begin{pmatrix}\cdot&\text{🌢}&\cdot\\\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\end{pmatrix}}\,\text{Cr}}{\texttt{C1} : \begin{pmatrix}\cdot&\text{🌢}&\cdot\\\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\end{pmatrix}}\,\text{Cond} \qquad \dfrac{\dfrac{}{\texttt{z==1} : \begin{pmatrix}\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\end{pmatrix}}\,\text{Asgn} \quad \dfrac{}{\texttt{y=z} : \begin{pmatrix}\cdot&\cdot&\cdot\\\cdot&\cdot&\cdot\\\cdot&\text{🌢}&\cdot\end{pmatrix}}\,\text{Asgn} \quad \dfrac{}{\texttt{y==0} : \begin{pmatrix}\cdot&\cdot&\cdot\\\cdot&\cdot&\text{🌢}\\\cdot&\cdot&\cdot\end{pmatrix}}\,\text{Cr}}{\texttt{C2} : \begin{pmatrix}\cdot&\cdot&\cdot\\\cdot&\cdot&\text{🌢}\\\cdot&\text{🌢}&\cdot\end{pmatrix}}\,\text{Cond}}{\texttt{C1;C2} : \begin{pmatrix}\cdot&\text{🌢}&\cdot\\\cdot&\cdot&\text{🌢}\\\cdot&\text{🌢}&\cdot\end{pmatrix}}\,\text{Comp}$$

[7] In brief terms, this comes from Table 1, remembering that $\mathtt{x}_j$ being in the condition in the (L) and (B) cases implies that it is in $\mathrm{In}(\mathtt{C}')$ and that a 🌢 was introduced between its in-variable and $\mathtt{x}_i$'s out-variable in $\mathbb{M}(\mathtt{C}')$.

[8] Remembering Table 1, if $\mathtt{x}_j$ occurs in the expression of a condition, it occurs in the In set of the overall program.

[9] This is where our "falsifiability of expressions" hypothesis is used.

The concluding matrix captures the flows: `z` to `y`, `h` to `y` and `y` to `z`, but does not show (at top-right) the transitive flow from `h` to `z`. A violation depends on the assignment of security classes. Non-interference requires $\ell(\texttt{h}) \leqslant \ell(\texttt{y})$ and $\ell(\texttt{y}) \leqslant \ell(\texttt{z})$. This exposes the transitive flow $\ell(\texttt{h}) \leqslant \ell(\texttt{z})$. A SFM contains more information than what is immediately visible. □

▶ **Example 18** (Composition irrelevance). We derive a matrix for program

```
z = 3; x = y; x = z
```

by

$$\dfrac{\dfrac{\dfrac{}{\texttt{z=3} : \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}}\ \text{Asgn} \qquad \dfrac{}{\texttt{x=y} : \begin{pmatrix} \cdot & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}}\ \text{Asgn}}{\texttt{z=3;x=y} : \begin{pmatrix} \cdot & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{pmatrix}}\ \text{Comp} \qquad \dfrac{}{\texttt{x=z} : \begin{pmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot \end{pmatrix}}\ \text{Asgn}}{\texttt{z=3;x=y;x=z} : \begin{pmatrix} \cdot & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot \end{pmatrix}}\ \text{Comp}$$

The matrix labels are `x y z`. If `y` holds secret data, and `x` is a public with $\ell(\texttt{x}) < \ell(\texttt{y})$, the program violates anytime non-interference. Although `x` is overwritten in a later command, a violation cannot be erased once it has occurred. Also observe that composition is commutative – composing the commands in any order would yield the same program matrix. □

▶ **Example 19** (Context sensitivity). The following program (from [18] adjusted to Fig. 1) shows assignments to string buffers `sb1` and `sb2`. The potentially sensitive `request` does not interfere with `query`. A context-sensitive analysis detects this and does not raise an unnecessary alarm.

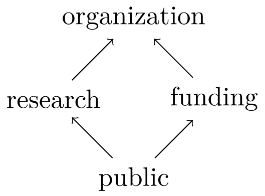
```
user = request["user"];
sb1 = "SELECT * FROM Users WHERE name=";
sb2 = "SELECT * FROM Users WHERE name=";
sb1 += user;
sb2 += "John";
query=sb2;
// execute query
```

$$\begin{array}{r|ccccc|} & \texttt{user} & \texttt{request} & \texttt{sb1} & \texttt{sb2} & \texttt{query} \\ \texttt{user} & \cdot & \cdot & \blacklozenge & \cdot & \cdot \\ \texttt{request} & \blacklozenge & \cdot & \cdot & \cdot & \cdot \\ \texttt{sb1} & \cdot & \cdot & \cdot & \cdot & \cdot \\ \texttt{sb2} & \cdot & \cdot & \cdot & \cdot & \blacklozenge \\ \texttt{query} & \cdot & \cdot & \cdot & \cdot & \cdot \end{array}$$

The program matrix is on the right. An anytime non-interference violation is avoided if $\ell(\texttt{request}) \leqslant \ell(\texttt{user})$, $\ell(\texttt{user}) \leqslant \ell(\texttt{sb1})$, and $\ell(\texttt{sb2}) \leqslant \ell(\texttt{query})$. Since `request` and `query` are disjoint in the matrix, the variables are anytime non-interfering for all security classes. □

Our next analysis example requires a policy with incomparable security classes, like the one we present now.

▶ **Example 20** (HMO information flow policy). The HMO (for Health Maintenance Organization) information flow policy, represented as a Hasse diagram, is:



□

▶ **Example 21** (Incomparable security classes). The mvt–kernel, from the PolyBench/C [25] parallel programming benchmark suite, calculates a **m**atrix **v**ector product and **t**ranspose.

```
void kernel_mvt(...){
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      x1[i] = x1[i]+A[i][j]*y1[j];
  for (i=0; i<N; i++)
    for (j=0; j<N; j++)
      x2[i] = x2[i]+A[j][i]*y2[j];
}
```

$$\begin{array}{c|cccccccc|}
 & \mathtt{i} & \mathtt{j} & \mathtt{N} & \mathtt{x1} & \mathtt{x2} & \mathtt{y1} & \mathtt{y2} & \mathtt{A} \\
\mathtt{i} & \cdot & \blacklozenge & \cdot & \blacklozenge & \blacklozenge & \cdot & \cdot & \cdot \\
\mathtt{j} & \cdot & \cdot & \cdot & \blacklozenge & \blacklozenge & \cdot & \cdot & \cdot \\
\mathtt{N} & \blacklozenge & \blacklozenge & \cdot & \blacklozenge & \blacklozenge & \cdot & \cdot & \cdot \\
\mathtt{x1} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\mathtt{x2} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\mathtt{y1} & \cdot & \cdot & \cdot & \blacklozenge & \cdot & \cdot & \cdot & \cdot \\
\mathtt{y2} & \cdot & \cdot & \cdot & \cdot & \blacklozenge & \cdot & \cdot & \cdot \\
\mathtt{A} & \cdot & \cdot & \cdot & \blacklozenge & \blacklozenge & \cdot & \cdot & \cdot
\end{array}$$

Observe that x1 and x2 are disjoint, sharing no observable information flow in the matrix. Their security classes may be incomparable. Using the HMO policy, the assignment

$$\{\mathtt{i},\mathtt{j},\mathtt{N},\mathtt{A}\} \mapsto \text{public} \qquad \{\mathtt{y1},\mathtt{x1}\} \mapsto \text{research} \qquad \{\mathtt{y2},\mathtt{x2}\} \mapsto \text{funding}$$

is satisfactory. Similarly, assignment $\ell(\mathtt{x1}) = \text{organization}$ satisfies anytime non-interference. However, $\ell(\mathtt{A}) = \text{research}$ is a violation because we have $\mathbb{M}(\mathtt{C})(\mathtt{A},\mathtt{x2}) = \blacklozenge$. It would require that research $\leqslant$ funding, but by the HMO policy the classes are incomparable. □

▶ **Example 22** (Function calls and arrays). The program references two functions (treated as pure), called inside the condition and inside the body of a `while` loop:

```
while(y < f(b)){
  t[f(b)] = x;
  a = t[a] + b;
  y = g(b, x);
  x = f(a);
}
```

$$\begin{array}{c|cccccccc|}
 & \mathtt{t} & \mathtt{a} & \mathtt{b} & \mathtt{x} & \mathtt{y} & \mathtt{f}^{\mathtt{out}}_{\mathtt{a}} & \mathtt{f}^{\mathtt{out}}_{\mathtt{b}} & \mathtt{g}^{\mathtt{out}}_{\mathtt{b,x}} \\
\mathtt{t} & \cdot & \blacklozenge & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\mathtt{a} & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\mathtt{b} & \cdot & \blacklozenge & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\mathtt{x} & \blacklozenge & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\
\mathtt{y} & \blacklozenge & \blacklozenge & \cdot & \blacklozenge & \cdot & \cdot & \cdot & \cdot \\
\mathtt{f}^{\mathtt{in}}_{\mathtt{a}} & \cdot & \cdot & \cdot & \blacklozenge & \cdot & \cdot & \cdot & \cdot \\
\mathtt{f}^{\mathtt{in}}_{\mathtt{b}} & \blacklozenge & \blacklozenge & \cdot & \blacklozenge & \blacklozenge & \cdot & \cdot & \cdot \\
\mathtt{g}^{\mathtt{in}}_{\mathtt{b,x}} & \cdot & \cdot & \cdot & \cdot & \blacklozenge & \cdot & \cdot & \cdot
\end{array}$$

Since variables introduced for (pure) function calls correspond to *expressions*, they do not belong to any Out set, and their columns in a security flow matrix will always be empty. However, the class of function parameters can be considered with additional constraints, not reported in the security-flow matrix. Typically, one can require that $\ell(\mathtt{g}^{\mathtt{in}}_{\mathtt{b,x}}) = \max(\ell(\mathtt{b}), \ell(\mathtt{x}))$ for all pairs x, b such that g(b, x) occurs in the program. This would invalidate a level assignment with e.g., $\ell(\mathtt{t}) < \ell(\mathtt{b})$, since $\ell(\mathtt{b}) \leqslant \ell(\mathtt{g}^{\mathtt{in}}_{\mathtt{b,x}})$ would be required by this condition, and $\ell(\mathtt{g}^{\mathtt{in}}_{\mathtt{b,x}}) \leqslant \ell(\mathtt{y}) \leqslant \ell(\mathtt{t})$ are required by the security-flow matrix. □