An Information Flow Calculus for Non-Interference

Clément Aubert
Augusta University
Augusta, GA, USA
caubert@augusta.edu, ORCID

Neea Rusch
Augusta University
Augusta, GA, USA
nrusch@augusta.edu, ORCID

***Abstract*—Sensitive data exposure is persistently ranked among the top-ten web application security risks, thus every software developer should actively combat data exposure vulnerabilities. Information flow controls offer mechanisms to enforce data confidentiality. Unfortunately, strict controls are too restrictive for real applications and innovation is needed to obtain practical solutions. We present a brave new idea: an information flow calculus for non-interference. Our formulation enforces that command composition does not create nor erase non-interference violations, and the sound calculus pinpoints precisely where violations occur. The calculus can be implemented as an automatic, compositional, and annotation-free static security analyzer to obtain confidentiality guarantees in practice.**

***Index Terms*—Automated Security Analysis, Confidentiality, Information Flow Control, Language-based Security**

I. Introduction

Careless exposure of sensitive information is at the core of numerous modern software security vulnerabilities and exploits. Access control, firewalls, and encryption are among the control mechanisms to enforce *confidentiality*, i.e., disclosure of information to authorized parties only. Unfortunately, those controls are insufficient to provide end-to-end guarantees, since confidentiality is not ensured once information has been used as input data to a program [1]. *Non-interference* [2] is a classical program security property, that constrains information flow during computation. A program is non-interfering when secret data does not affect calculation of public outputs [3].

Although non-interference is a desirable property, it is in general undecidable by Rice's Theorem [4]. Instead, various approximative approaches have emerged to reason about non-interference in theory [5]–[7] and practice [8]–[12]. In this work, we focus on *language-based security*, a family of techniques founded on programming languages principles e.g., semantics, analysis, type systems, and rewriting; as information-flow controls for enforcing non-interference [3], [13]. A well-established technique is using a security type system that insures non-interference via type-checking at compile-time.

In this work we take a different approach by presenting an *information flow calculus* for non-interference. It is an automatable, fully-static analysis of imperative program syntax, for deriving the semantic property of non-interference. The technique is adapted from prior work [14], with substantial modification to adjust it to security property analysis. Coincidentally, the theoretical adjustments led us to use only commutative operations, so that non-interference violations are preserved through program composition. Our analysis is not bound by a particular model; instead it outputs a set of constraints for the analyzed program to be non-interfering, and those constraints can be analyzed and manipulated independently of the program that generated them. Thus, our calculus offers theoretical interest *and* practical potential for automatic analysis.

II. High-Level Overview

We manipulate the conventional lattice model for information flow à la Denning [15]. An *information flow policy* is a lattice $(\mathrm{SC}, <)$ where SC is a partially $<$-ordered finite set of *security classes* [5]. We write ℓ for the level assignment that assigns statically and definitely to each variable $\mathtt{x}$ its security class (or *level*) $\ell(\mathtt{x}) \in \mathrm{SC}$. A simple policy has two security classes, e.g., $\mathrm{LH} = (\{l, h\}, \{l < h\})$—for *low* and *high*; but a policy can be extended to arbitrary classes, refer to Ex. 4 for an example.

For program analysis, we consider deterministic imperative programs, with conventional operational semantics and variables of simple data types. Let our expository program be:

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

In this program certain data flows are potentially problematic. The assignment `x=y` is an *explicit flow*, since variable `y` directly impacts `x`. The control expressions `z==1` and `x==1` represent *implicit flows*, as they reveal information over execution paths and indirectly expose values of control statement variables. Admissibility of these data flows depends on the security classes of the variables, as non-interference forbids "read-up", i.e., data flows from higher to lower level. A sound analysis technique detects such issues and raises an alarm.

Our flow calculus produces a result by applying inference rules to statements of a program. The result is a matrix of coefficients. In a single derivation, it captures dependencies between program variables for all execution paths and security classes. The matrix of our expository program is

$$\begin{array}{c} \\ \mathtt{x} \\ \mathtt{y} \\ \mathtt{z} \end{array} \begin{array}{c} \begin{array}{ccc} \mathtt{x} & \mathtt{y} & \mathtt{z} \end{array} \\ \begin{pmatrix} \cdot & \blacklozenge & \cdot \\ \blacklozenge & \cdot & \cdot \\ \blacklozenge & \blacklozenge & \cdot \end{pmatrix} \end{array}.$$

The matrix is interpreted by matching in-variables, $\mathtt{v}_{\mathrm{in}}$ (rows) with out-variables, $\mathtt{v}_{\mathrm{out}}$ (columns). The coefficients indicate:
$\cdot$ – *no violation*, no dependency from $\mathtt{v}_{\mathrm{in}}$ to $\mathtt{v}_{\mathrm{out}}$,
$\blacklozenge$ – a non-interference *violation*, if $\ell(\mathtt{v}_{\mathrm{in}}) \not\leqslant \ell(\mathtt{v}_{\mathrm{out}})$.

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

Figure 1. A simple imperative **while** language

For our expository program, the matrix expectedly shows potential violations in data flows from z to x, z to y and x to y (implicit), and from y to x (explicit). In isolation, the matrix gives a summary of potential violations for the program that induced it. The matrix can be *evaluated* after it has been paired with variable security class assignments to determine if the program is non-interfering. It can also be used to determine if a security class assignment exists that makes the program non-interfering. Throughout the paper when making such evaluations, we assume a program-centric attacker model [16], where the attacker can control public inputs and observe public outputs.

The analysis compositionality, syntax extendability, built-in automatable inference algorithm, and handling of transitive dependencies and implicit flows are among the strengths of our technique. These features are of practical interest, since static language-based security analyses often ignore implicit flows [12, pg. 144]. Furthermore, as a syntactic calculus, it captures precisely the statement and variables involved in a violation, allowing informative error-reporting to program developers.

In this paper we present the mathematical foundations and soundness of our information flow calculus; demonstrate the technique in action on illustrative examples, and sketch a plan for practical information flow analysis and implementation.

III. The Non-interference Calculus

A. A Simple While Imperative Language

We use a simple imperative **while** language, with semantics similar to C. The grammar is given in Fig. 1. Our language supports arrays and we let **for** and **do...while** loops be represented using **while** loops. It is easy to map to fragments of C, Java, or any other imperative programming language, and it subsumes (up to letvar construct) the "core block-structured language" of Volpano et al. [5].

A variable 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 V (resp. e, C) range over variables (resp. expressions, 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 is thus a sequence of statements, each statement being either an *assignment*, a *skip*, a *conditional*, or a *while* loop. *Statements* are abstracted into *commands*, which can be a statement or a sequence of commands.

For convenience, we define the following sets of variables.

Definition 1. Given an expression e, we define the variables occurring in e by:

$$
\mathrm{Occ}(\mathtt{x}) = \mathtt{x} \quad \mathrm{Occ}(\mathtt{t[e]}) = \mathtt{t} \cup \mathrm{Occ}(\mathtt{e}) \quad \mathrm{Occ}(\mathit{val}) = \emptyset
$$
$$
\mathrm{Occ}(\mathtt{op}(\mathtt{e}_1, \ldots, \mathtt{e}_n)) = \mathrm{Occ}(\mathtt{e}_1) \cup \cdots \cup \mathrm{Occ}(\mathtt{e}_n)
$$

Definition 2. Let C be a command, we let $\mathrm{Out}(\mathtt{C})$ (resp. $\mathrm{In}(\mathtt{C})$, $\mathrm{Occ}(\mathtt{C})$) be the set of variables *modified* by (resp. *used* by, *occurring* in) C as defined in Table I.

B. Security-Flow Matrices for Non-interference Violation

Our non-interference calculus 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 [14], and was influenced by a large body of works related to static analysis [17]–[21] and optimization [22]. We assume familiarity with semi-groups and matrices addition.

A security-flow matrix (SFM) for a command is a hollow matrix (i.e., a matrix with only $\cdot$ on the diagonal[1]) over a semi-group[2] with an implicit choice of a denumeration of $\mathrm{Occ}(\mathtt{C})$.[3]

Definition 3 (Security semi-group). We let $\mathrm{SSG} = (\{\cdot, \blacklozenge\}, \max)$ with $\cdot < \blacklozenge$ be the *security semi-group*.

This semi-group is isomorphic to the two-element Boolean algebra, with the intuition that $\blacklozenge$ represents a possible violation (or information "leak").

Definition 4 (SFM). We write $\mathbb{M}(\mathtt{C})$ the SFM of C and $\mathbb{M}(\mathtt{C})(\mathtt{x}, \mathtt{y})$ for the coefficient in $\mathbb{M}(\mathtt{C})$ at the row corresponding to x and column corresponding to y. If there exists x and y such that $\mathbb{M}(\mathtt{C})(\mathtt{x}, \mathtt{y}) = \blacklozenge$ and $\ell(\mathtt{x})$ is not less than nor equal to $\ell(\mathtt{y})$ (denoted $\ell(\mathtt{x}) \not\leqslant \ell(\mathtt{y})$), then C *has a violation*:

$$
\begin{array}{c} \\ \mathtt{x} \end{array}
\begin{array}{c}
\begin{array}{ccc} \cdots & \mathtt{y} & \cdots \end{array} \\
\begin{pmatrix} \ddots & & \iddots \\ & \blacklozenge & \\ \iddots & & \ddots \end{pmatrix}
\end{array}
\implies \mathtt{C} \text{ has a violation if } \ell(\mathtt{x}) \not\leqslant \ell(\mathtt{y}).
$$

Stated negatively, the program will not have a violation (at least for those variables and levels, refer to Sect. IV for a complete definition of non-interference) if the level of x is less than the level of y, or if they are of equal level. Note that if the levels of x and y are different but not ordered by $<$, then C has a violation. Since for all x, $\ell(\mathtt{x}) \not\leqslant \ell(\mathtt{x})$ is always false,

[1]This choice is clarified after Def. 4.

[2]The original data-flow graph construction requires a semi-ring, but product is not necessary for our purpose here.

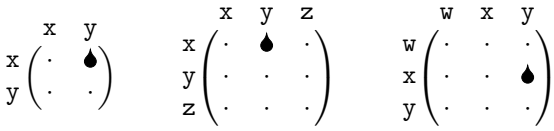
[3]We will use the order in which the variables occur in the program as their implicit order most of the time.

Table I
DEFINITION OF Out, In AND Occ FOR COMMANDS

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

there is no point keeping track of the values on the diagonal: this is why hollow matrices are enough.

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



C. Constructing Security-Flow Matrices

The security-flow matrix (SFM) of a command is constructed by induction on the structure of the command, using the security semi-group SSG.

1) Base cases (assignment, skip): The SFM for an assignment C is computed using $\mathrm{In}(\mathtt{C})$ and $\mathrm{Out}(\mathtt{C})$:

Definition 5 (Assignment). Given an assignment C, its SFM is given by:

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

We illustrate in Fig. 3 some basic cases: note that in the case of violations, $\mathrm{In}(\mathtt{C})$ is exactly the set of variables that are sources of a violation, while $\mathrm{Out}(\mathtt{C})$ is the set of variables that are targets of violations.

We consider an array a single entity, and that changing one value in it means being able to access it completely. We also consider an expression $\mathtt{t[i]}$ to be a non-interference violation if $\ell(\mathtt{i}) \not\leqslant \ell(\mathtt{t})$: indeed, it would otherwise mean that a lower-level variable ($\mathtt{i}$) can access the value of a higher-level variable ($\mathtt{t}$). However, $\mathtt{t[i]}$ is acceptable if $\ell(\mathtt{t}) \geqslant \ell(\mathtt{i})$, since $\mathtt{t}$ is "using" $\mathtt{i}$ to perform internal calculation without exposing its values publicly.

The SFM for **skip** is simply an empty matrix:

Definition 6 (Skip). We let $\mathbb{M}(\mathtt{skip})$ be the matrix with 0 rows and columns.[4]

[4]Identifying the SFM with its embeddings, it is hence the matrix containing only $\cdot$ of any size.

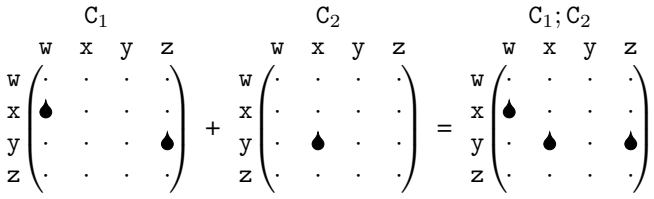


Figure 2. Security-Flow Matrix of Compositions.

2) Composition: The definition of SFM for a (sequential) *composition* of commands is an abstraction that allows treating a block of statements as one command with its own SFM.

Definition 7 (Composition). We let $\mathbb{M}(\mathtt{C_1}; \ldots; \mathtt{C_n})$ be $\mathbb{M}(\mathtt{C_1}) + \cdots + \mathbb{M}(\mathtt{C_n})$.

The composition of commands $\mathtt{C_1}$ and $\mathtt{C_2}$–themselves already the result of compositions of assignments involving disjoint variables–is illustrated in Fig. 2.

3) Correction: To account for implicit flows correctly, conditionals and loops require a *correction* to compute their SFMs. Indeed, the SFMs of **if** e **then** $\mathtt{C_1}$ **else** $\mathtt{C_2}$ and **while** e **do** C require more than the SFM of its body. All the modified variables in $\mathtt{C_1}$ and $\mathtt{C_2}$ or C (e.g., $\mathrm{Out}(\mathtt{C_1}) \cup \mathrm{Out}(\mathtt{C_2})$ or $\mathrm{Out}(\mathtt{C})$) depend on the variables occurring in e (e.g., in $\mathrm{Occ}(\mathtt{e})$).

Definition 8 (Correction). For an expression e and a command C, we define e*'s correction for* C, $\mathrm{Cr}(\mathtt{e})_{\mathtt{C}}$, as

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

Intuitively, the correction simply says that if the variable y is modified in the body of either branch of the conditional or in the body of the loop and x occurs in the expression, then there is a violation if $\ell(\mathtt{x}) \not\leqslant \ell(\mathtt{y})$–but we can discard the case where $\mathtt{x} = \mathtt{y}$ as always.

As an example, let us re-use the programs $\mathtt{C_1}$ and $\mathtt{C_2}$ from Fig. 2, to construct $\mathtt{w} > \mathtt{x}$'s correction for $\mathtt{C_1};\mathtt{C_2}$, that we write $\mathrm{Cr}(\mathtt{w} > \mathtt{x})_{\mathtt{C_1};\mathtt{C_2}}$. Variables w and x, through the expression w>x, control the values of w, x and z since $\mathtt{C_1}$ and $\mathtt{C_2}$ set those

C	Out(C), In(C)	$\mathbb{M}$(C)	Violation(s)
$\mathtt{w} = 3$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{w}\}$ $\mathrm{In}(\mathtt{C}) = \emptyset$	$\begin{array}{c}\mathtt{w}\\ \mathtt{w}\ (\cdot)\end{array}$	None
$\mathtt{x} = \mathtt{y}$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{x}\}$ $\mathrm{In}(\mathtt{C}) = \{\mathtt{y}\}$	$\begin{array}{c}\quad\mathtt{x}\ \ \mathtt{y}\\ \mathtt{x}\begin{pmatrix}\cdot & \cdot\\ \bullet & \cdot\end{pmatrix}\\ \mathtt{y}\end{array}$	If $\ell(\mathtt{y}) \not\leqslant \ell(\mathtt{x})$
$\mathtt{w} = \mathtt{t}[\mathtt{x}+1]$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{w}\}$ $\mathrm{In}(\mathtt{C}) = \{\mathtt{t},\mathtt{x}\}$	$\begin{array}{c}\quad\mathtt{w}\ \ \mathtt{t}\ \ \mathtt{x}\\ \mathtt{w}\begin{pmatrix}\cdot & \cdot & \cdot\\ \bullet & \cdot & \cdot\\ \bullet & \cdot & \cdot\end{pmatrix}\\ \mathtt{t}\\ \mathtt{x}\end{array}$	If $\ell(\mathtt{t}) \not\leqslant \ell(\mathtt{w})$ or $\ell(\mathtt{x}) \not\leqslant \ell(\mathtt{w})$.
$\mathtt{t}[\mathtt{i}] = \mathtt{u} + \mathtt{j}$	$\mathrm{Out}(\mathtt{C}) = \{\mathtt{t}\}$ $\mathrm{In}(\mathtt{C}) = \{\mathtt{i},\mathtt{u},\mathtt{j}\}$	$\begin{array}{c}\quad\mathtt{t}\ \ \mathtt{i}\ \ \mathtt{u}\ \ \mathtt{j}\\ \mathtt{t}\begin{pmatrix}\cdot & \cdot & \cdot & \cdot\\ \bullet & \cdot & \cdot & \cdot\\ \bullet & \cdot & \cdot & \cdot\\ \bullet & \cdot & \cdot & \cdot\end{pmatrix}\\ \mathtt{i}\\ \mathtt{u}\\ \mathtt{j}\end{array}$	If $\ell(\mathtt{i}) \not\leqslant \ell(\mathtt{t})$, $\ell(\mathtt{u}) \not\leqslant \ell(\mathtt{t})$, or $\ell(\mathtt{j}) \not\leqslant \ell(\mathtt{t})$.

Figure 3. Statement Examples, Sets, Representations of their Possible Non-interference Violation(s).

values, and their execution depend on it:

$$\begin{array}{c}\quad\mathtt{w}\ \ \mathtt{x}\ \ \mathtt{y}\ \ \mathtt{z}\\ \mathtt{w}\\ \mathtt{x}\\ \mathtt{y}\\ \mathtt{z}\end{array}\begin{pmatrix}\cdot & \bullet & \cdot & \bullet\\ \bullet & \cdot & \cdot & \bullet\\ \cdot & \cdot & \cdot & \cdot\\ \cdot & \cdot & \cdot & \cdot\end{pmatrix}$$

4) Conditionals and While Loops: Following our previous observation, conditionals and loops are interpreted similarly.

Definition 9 (Conditional branching). We let $\mathbb{M}(\texttt{if e then C}_1\texttt{ else C}_2)$ be $\mathbb{M}(\mathtt{C}_1)+\mathbb{M}(\mathtt{C}_2)+\mathrm{Cr}(\mathtt{e})_{\mathtt{C}_1;\mathtt{C}_2}$.

Adding $\mathrm{Cr}(\mathtt{w} > \mathtt{x})_{\mathtt{C}_1;\mathtt{C}_2}$ to the SFMs of $\mathtt{C}_1$ and $\mathtt{C}_2$ from Fig. 2, we obtain

$$\mathbb{M}\left(\begin{array}{l}\texttt{if (w>x)}\\ \quad\texttt{then } \mathtt{w} = \mathtt{w} + \mathtt{x};\\ \qquad\quad \mathtt{z} = \mathtt{y} + 2\\ \quad\texttt{else } \mathtt{x} = \mathtt{y};\\ \qquad\quad \mathtt{z} = \mathtt{z} * 2\end{array}\right) = \begin{array}{c}\quad\mathtt{w}\ \ \mathtt{x}\ \ \mathtt{y}\ \ \mathtt{z}\\ \mathtt{w}\\ \mathtt{x}\\ \mathtt{y}\\ \mathtt{z}\end{array}\begin{pmatrix}\cdot & \bullet & \cdot & \bullet\\ \bullet & \cdot & \cdot & \bullet\\ \cdot & \bullet & \cdot & \bullet\\ \cdot & \cdot & \cdot & \cdot\end{pmatrix}.$$

Observe that there is a violation if $\ell(\mathtt{x}) \not\leqslant \ell(\mathtt{w})$ inherited from the statement $\mathtt{w} = \mathtt{w} + \mathtt{x}$, and that there is a violation if $\ell(\mathtt{w}) \not\leqslant \ell(\mathtt{x})$. The latter comes from the fact that the value of w will decide if $\mathtt{x} = \mathtt{y}$ will execute through the expression. This means that, to be non-interfering, such a program must be given levels satisfying $\ell(\mathtt{w}) = \ell(\mathtt{x})$, and the other constraints recorded in the matrix.

Definition 10 (While loop). We let $\mathbb{M}(\texttt{while e do C})$ be $\mathbb{M}(\mathtt{C}) + \mathrm{Cr}(\mathtt{e})_{\mathtt{C}}$.

As an example, we let the reader convince themselves that the SFM of

```
while(t[i]!=j){
  s1[i]=j*j;
  s2[i]=1/j;
  i++
}
```

is

$$\begin{array}{c}\quad\mathtt{t}\ \ \mathtt{i}\ \ \mathtt{j}\ \ \mathtt{s1}\ \ \mathtt{s2}\\ \mathtt{t}\\ \mathtt{i}\\ \mathtt{j}\\ \mathtt{s1}\\ \mathtt{s2}\end{array}\begin{pmatrix}\cdot & \bullet & \cdot & \bullet & \bullet\\ \cdot & \cdot & \cdot & \bullet & \bullet\\ \cdot & \bullet & \cdot & \bullet & \bullet\\ \cdot & \cdot & \cdot & \cdot & \cdot\\ \cdot & \cdot & \cdot & \cdot & \cdot\end{pmatrix}.$$

Intuitively, the rows for s1 and s2 are filled with ·s, since those variables do not control any other variable and are assigned in the body of the loop. On the other hand, t, i and j all three control the values of i, s1 and s2, since they determine if the body of the loop will execute.

To ease the presentation, we will in Sect. V present those 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.

IV. Soundness

Originally, in the seminal work of Volpano et al. [5, pg. 173], "[s]oundness [wa]s formulated as a kind of noninterference property. [...I]f a variable v has security level τ, then one can change the initial values of any variables whose security levels are not dominated by τ, execute the program, and the final value of v will be the same, provided the program terminates successfully." We now give the definitions to formulate soundness in our set-up.

First remember, that for a given command C, we assumed a fixed order for the variables in $\mathrm{Occ}(\mathtt{C})$.[3]

Definition 11 (Command evaluation). Given a command C with $|\mathrm{Occ}(\mathtt{C})| = n$ variables $\vec{\mathtt{x}} = \mathtt{x}_1, \ldots, \mathtt{x}_n$ and n values $\vec{v} = v_1, \ldots, v_n$,[5] we write $\mathtt{C}[\vec{v}]$ for the command C where the variable $\mathtt{x}_i$ received the value v_i before execution, for $1 \leqslant i \leqslant n$. We also write $\mathtt{C}[\vec{v} \to \vec{v'}]$ if

- $\mathtt{C}[\vec{v}]$ terminates and

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

- after executing all the commands in $\mathtt{C}[\vec{v}]$, $\mathtt{x}_i$ contains the value v'_i, for $1 \leqslant i \leqslant n$.

Remember from Sect. II that, given an information flow policy $(\mathrm{SC}, <)$, a *level assignment* ℓ is a mapping from $|\operatorname{Occ}(\mathtt{C})|$ to SC. We write $\leqslant$ for the reflexive closure of $<$.

Definition 12 (Non-interfering level assignment). Given a SFM $\mathbb{M}(\mathtt{C})$, a level assignment ℓ is *non-interfering* iff for all $\mathtt{x}_i$, $\mathtt{x}_j$ in $\operatorname{Occ}(\mathtt{C})$, $i \neq j$, $\mathbb{M}(\mathtt{C})(\mathtt{x}_i, \mathtt{x}_j) = \blacklozenge \implies$ not $\ell(\mathtt{x}_j) \not\leqslant \ell(\mathtt{x}_i)$.

As we observe previously, $\ell(\mathtt{x}_j) \not\leqslant \ell(\mathtt{x}_i)$ being false is equivalent to $\ell(\mathtt{x}_j)$ being greater than or equal to $\ell(\mathtt{x}_i)$, but does not include the case where those levels are incomparable. Note that the trivial level assignments that assigns to all values the same security class are always non-interfering, since in that case $\ell(\mathtt{x}_i) \not\leqslant \ell(\mathtt{x}_j)$ is always false.

Definition 13 (Up-to equivalence). Given $\mathtt{C}$, $(\mathrm{SC}, <)$, ℓ and a level $l \in \mathrm{SC}$, two values lists $\vec{v}$ and $\vec{v'}$ are *up-to l equivalent*, written $\vec{v} =^{\ell}_{l\leqslant} \vec{v'}$ iff $\ell(\mathtt{x}_i) \leqslant l \implies v_i = v'_i$.

More intuitively, two value lists are up-to l equivalent if they agree on the values of all the variables of level l or below.

We can now formally state the non-interference property:

Definition 14 (Non-interference). A command $\mathtt{C}$ is *non-interfering for* ℓ if for all level $l \in \mathrm{SC}$, and all $\vec{v_1}$ and $\vec{v_2}$,

$$\vec{v_1} =^{\ell\leqslant}_{l} \vec{v_2}, \mathtt{C}[\vec{v_1} \to \vec{v'_1}], \mathtt{C}[\vec{v_2} \to \vec{v'_2}] \implies \vec{v'_1} =^{\ell\leqslant}_{l} \vec{v'_2}$$

Informally, non-interference states that changing the value received by higher-level variables does not impact the final value of lower-level variables. We conjecture that this property can be established using our analysis:

Conjecture 1 (Correspondence). A command is non-interfering for ℓ (Def. 14) iff ℓ is non-interfering (Def. 14).

V. Illustrative Derivations

Example 1 (Transitive dependence). Consider a program of two commands. Although there is no direct assignment from `h` to `z`, they are transitively dependent through `y`.

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

Omitting the empty matrices induced by assignments `y=1` and `z=1` and **`skip`**, the derivation for the program is:

$$\dfrac{\dfrac{\dfrac{}{\mathtt{h==0} : \begin{pmatrix}\cdot & \blacklozenge \\ \cdot & \cdot\end{pmatrix}}\,\mathrm{Cr} \quad \dfrac{}{\mathtt{y==0} : \begin{pmatrix}\cdot & \cdot \\ \blacklozenge & \cdot\end{pmatrix}}\,\mathrm{Cr}}{\mathtt{C1} : \begin{pmatrix}\cdot & \blacklozenge \\ \cdot & \cdot\end{pmatrix}}\,\mathrm{Cond} \quad \dfrac{}{\mathtt{y=z} : \begin{pmatrix}\cdot & \cdot \\ \cdot & \blacklozenge\end{pmatrix}}\,\mathrm{Asgn}}{\mathtt{C2} : \begin{pmatrix}\cdot & \cdot \\ \blacklozenge & \blacklozenge\end{pmatrix}}\,\mathrm{Cond}\,\mathrm{Comp}$$

$$\mathtt{C1;C2} : \begin{pmatrix}\cdot & \blacklozenge \\ \cdot & \blacklozenge\end{pmatrix}$$

The final matrix captures the explicit flow from `z` to `y`, and the implicit flows from `h` to `y`, and `y` to `z`, but does not show, in top-right, the transitive dependence from `h` to `z`. However, a violation depends on the security classes. A non-interfering program requires $\ell(\mathtt{h}) \leqslant \ell(\mathtt{y})$ and $\ell(\mathtt{y}) \leqslant \ell(\mathtt{z})$, thus exposing the transitive dependency $\ell(\mathtt{h}) \leqslant \ell(\mathtt{z})$. A secondary observation is that a matrix shows violations arising from direct assignments and implicit flows, but contains more information than what is immediately visible. □

Example 2 (Composition irrelevance). We derive a matrix for the program `z=3;x=y;x=z` as follows:

$$\dfrac{\dfrac{}{\mathtt{z=3} : \begin{pmatrix}\cdot & \cdot \\ \cdot & \cdot\end{pmatrix}}\,\mathrm{Asgn} \quad \dfrac{}{\mathtt{x=y} : \begin{pmatrix}\blacklozenge & \cdot \\ \cdot & \cdot\end{pmatrix}}\,\mathrm{Asgn}}{\mathtt{z=3;x=y} : \begin{pmatrix}\blacklozenge & \cdot \\ \cdot & \cdot\end{pmatrix}}\,\mathrm{Comp} \quad \dfrac{}{\mathtt{x=z} : \begin{pmatrix}\cdot & \cdot \\ \blacklozenge & \cdot\end{pmatrix}}\,\mathrm{Asgn}$$

$$\mathtt{z=3;x=y;x=z} : \begin{pmatrix}\blacklozenge & \cdot \\ \blacklozenge & \cdot\end{pmatrix}\,\mathrm{Comp}$$

If `y` holds secret data, and `x` is a public-level variable (i.e., $\ell(\mathtt{x}) < \ell(\mathtt{y})$), the program violates non-interference policy. Even though `x` is later overwritten in another statement, a violation cannot be erased once it has occurred. In other words, statements that precede or follow an insecure operation do not cancel the interference triggered by the operation. The example also hints that composition is commutative, as composing the commands in any order would yield the same final matrix. □

Example 3 (Context sensitivity). The program below – from [12, p. 143] and adjusted to match Fig. 1 syntax – shows assignments to two string buffers, `sb1` and `sb2`. However, the potentially-sensitive variable `request` does not interfere with `query`. A context-sensitive analysis detects this fact 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
```

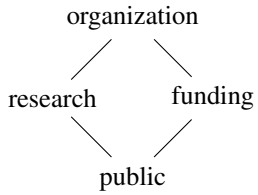
The matrix derived from the program is

$$\begin{array}{r} \\ \mathtt{user} \\ \mathtt{request} \\ \mathtt{sb1} \\ \mathtt{sb2} \\ \mathtt{query} \end{array} \begin{array}{c} \begin{array}{ccccc} \mathtt{user} & \mathtt{request} & \mathtt{sb1} & \mathtt{sb2} & \mathtt{query} \end{array} \\ \begin{pmatrix} \cdot & \cdot & \blacklozenge & \cdot & \cdot \\ \blacklozenge & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \blacklozenge \\ \cdot & \cdot & \cdot & \cdot & \cdot \end{pmatrix} \end{array}.$$

A non-interference violation is avoided when $\ell(\mathtt{request}) \leqslant \ell(\mathtt{user})$ and $\ell(\mathtt{user}) \leqslant \ell(\mathtt{sb1})$ and $\ell(\mathtt{sb2}) \leqslant \ell(\mathtt{query})$. All security class assignments that satisfy these constraints are non-interfering. Furthermore, since `request` and `query` are disconnected in the matrix, these variables are non-interfering for all security classes. □

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

Example 4. The HMO (for Health Maintenance Organization) information flow policy, represented as a Hasse diagram, is:



We sometimes abbreviate each level by its first three letters. □

Example 5 (Incomparable security classes). The mvt–kernel, from the PolyBench/C [23] parallel programming benchmark suite, calculates a Matrix Vector product and Transpose:

```
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]; }
```

Our information flow calculus assigns it a matrix

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

Besides the shared in-variables of `x1` and `x2`, a non-interfering program requires $\ell(\mathtt{y1}) \leqslant \ell(\mathtt{x1})$ and $\ell(\mathtt{y2}) \leqslant \ell(\mathtt{x2})$. Observe that `x1` and `x2` are independent, i.e., there is no visible data flow between them in the matrix; therefore, their security classes may be incomparable. Using the HMO policy with abbreviated level names, the security class assignment

$$\{\mathtt{i},\mathtt{j},\mathtt{N},\mathtt{A}\} \mapsto \text{pub.} \qquad \{\mathtt{y1},\mathtt{x1}\} \mapsto \text{res.} \qquad \{\mathtt{y2},\mathtt{x2}\} \mapsto \text{fun.}$$

is satisfactory. Similarly, setting `x1` : organization satisfies non-interference. However, `A` : research is a violation, because `A` is an in-variable of `x2` and would require establishing research $\leqslant$ funding. By the HMO policy, research and funding are incomparable.

This example also demonstrates how the calculus avoids "label creep", a phenomenon of monotonically-increasing, excessively restrictive labels [3]. Analyzing security classes *after* constructing a matrix, allows deferring decisions about security classes until all relevant data flows are know. For example, if `x1` : organization is assigned in the first loop, it was already shown that a lower security class assignment, `x2` : funding, is permissible in the second loop, even though the iteration control variables are shared between `x1` and `x2`. □

VI. Implementation Directions and Extensions

Obtaining automated security analysis is straightforward, following the lead of similar implementations [22], [24]. Unlike the existing works, our calculus bypasses several practical difficulties by being deterministic and using simple-valued matrices. As a fully-static analysis, designed for high-level imperative languages or compiler intermediate representations, it adds no runtime overhead to program execution. Automation requires two steps: (i) *analyze* a program to obtain a matrix (concretely, procedures and their corresponding matrices); then (ii) *evaluate* the matrix to obtain useful information, e.g., potential violations, and report it to program developer or software end-user. Because the calculus requires minimal program structure, and is annotation-free and compositional, it offers an viable complementary enhancement to e.g., inter-procedural static analyzes, that are precise but costly because they require a resource-intensive whole-program exploded supergraph; and type system-based static analyses, that expect manual type annotations or type inference.

A. Violation Detection With Taint Analysis

Taint analysis targets a subset of non-interference by considering two security classes of source (high) and sink (low). The goal is to detect true positive information-flow violations between source-and-sink pairs, without missing any violations and while having minimal false-alarm rate [25]. Taint analysis is widely applicable in e.g., detecting SQL-injection vulnerabilities [12], as suggested by Ex. 3.

With our non-interference flow calculus, taint analysis can be formulated as a constraint satisfaction problem. Given a matrix for a program, and its source and sink variables, we affix the source's (resp. sink's) security class to high (resp. low). Then, we encode the matrix violations as satisfiability modulo theories (SMT) constraints, and evaluate the matrix for satisfiability using an SMT solver. A satisfiable solution indicates absence of information-flow violation for all executions. An unsatisfiable solution means that for some execution paths, the program exposes confidential data. Extended variations of this implementation direction include determining if, for some choice of security classes, the program becomes non-interfering; and if so, finding the minimal security classes for the program to be non-interfering.

B. Extending the Language Syntax

Practical information flow analyses are frequently implemented on Java-like languages and bytecodes [9], [11], [12], [25]. While the Fig. 1 syntax captures the essence of these languages, extending it to object-oriented programming constructs would be beneficial. Such extensions would increase the class of analyzable programs and improve precision by accommodating various analysis *sensitivities*, e.g., object and field-sensitivities, to track method call behaviors, reference pointers, and fields of different objects [9], [11]. Further practical concerns include handling advanced language-features e.g., exceptions, reflection, and dynamic loading.

VII. Conclusion and Future Work

We have presented an information flow calculus that tracks soundly the property of non-interference in imperative programs. For a program, the calculus produces a security-flow

matrix that represents the program's potentially interfering data flows. As an abstraction, the matrix captures precisely all critical data flows and omits irrelevant details. Evaluating the matrix against assigned variable security classes gives a method to determine if the program satisfies non-interference. While our calculus has many theoretically compelling features, it can also be implemented to obtain practical automated security analysis.

Of course there is more work to be done. On the theoretical side, we are interested in characterizing larger classes of programs and introducing enhancements to increase analysis precision. Although not discussed earlier, we hypothesize that *integrity*, generally regarded as dual of confidentiality [26], could be achieved with a few straightforward adjustments of the calculus. On the practical side, we want to prototype our analysis as an automatic analyzer. A tool implementation would enable comparison against other static analyzers; would drive deeper understanding of the underlying theory, and guide the directions in which to extend the calculus syntax. Since much of the existing implementations focus on Java, and its related variants like Java bytecode and Android Runtime, the tool should focus on similar inputs.

REFERENCES

[1] S. Zdancewic, "Challenges for information-flow security," in *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, vol. 6, 2004. [Online]. Available: https://www.cis.upenn.edu/~stevez/papers/Zda04.pdf.

[2] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*, IEEE, 1982, pp. 11–20. DOI: 10.1109/SP.1982.10014.

[3] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003. DOI: 10.1109/JSAC.2002.806121.

[4] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical society*, vol. 74, no. 2, pp. 358–366, 1953. DOI: 10.1090/S0002-9947-1953-0053041-6.

[5] D. M. Volpano, C. E. Irvine, and G. Smith, "A sound type system for secure flow analysis," *JCS*, vol. 4, no. 2/3, pp. 167–188, 1996. DOI: 10.3233/JCS-1996-42-304.

[6] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke, "A core calculus of dependency," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99, Association for Computing Machinery, 1999, 147–160, ISBN: 1581130953. DOI: 10.1145/292540.292555.

[7] W. J. Bowman and A. Ahmed, "Noninterference for free," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP 2015, Association for Computing Machinery, 2015, 101–113, ISBN: 978-1-450-33669-7. DOI: 10.1145/2784731.2784733.

[8] A. C. Myers, "Jflow: Practical mostly-static information flow control," in *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '99, Association for Computing Machinery, 1999, 228–241, ISBN: 1581130953. DOI: 10.1145/292540.292561.

[9] C. Hammer and G. Snelting, "Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs," *International Journal of Information Security*, vol. 8, no. 6, pp. 399–422, 2009. DOI: 10.1007/s10207-009-0086-1.

[10] N. Broberg, B. van Delft, and D. Sands, "Paragon for practical programming with information-flow control," in *Programming Languages and Systems*, C.-c. Shan, Ed., Springer, 2013, pp. 217–232, ISBN: 978-3-319-03542-0. DOI: 10.1007/978-3-319-03542-0_16.

[11] S. Arzt, S. Rasthofer, C. Fritz, *et al.*, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Association for Computing Machinery, 2014, 259–269, ISBN: 978-1-450-32784-8. DOI: 10.1145/2594291.2594299.

[12] W. Huang, Y. Dong, and A. Milanova, "Type-based taint analysis for java web applications," in *Fundamental Approaches to Software Engineering*, S. Gnesi and A. Rensink, Eds., vol. 8411, Springer, 2014, pp. 140–154, ISBN: 978-3-642-54804-8. DOI: 10.1007/978-3-642-54804-8_10.

[13] F. B. Schneider, G. Morrisett, and R. Harper, "A language-based approach to security," LNCS, R. Wilhelm, Ed., pp. 86–101, 2001. DOI: 10.1007/3-540-44577-3_6.

[14] C. Aubert, T. Rubiano, N. Rusch, and T. Seiller, "Distributing and parallelizing non-canonical loops," in *Verification, Model Checking, and Abstract Interpretation*, C. Dragoi, M. Emmi, and J. Wang, Eds., ser. LNCS, vol. 13881, Springer, 2023, pp. 1–24, ISBN: 978-3-031-24949-5. DOI: 10.1007/978-3-031-24950-1_1.

[15] D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976. DOI: 10.1145/360051.360056.

[16] D. Hedin and A. Sabelfeld, "A perspective on information-flow control," in *Software safety and security*, IOS Press, 2012, pp. 319–347. DOI: 10.3233/978-1-61499-028-4-319.

[17] A. Abel and T. Altenkirch, "A predicative analysis of structural recursion," *J. Funct. Program.*, vol. 12, no. 1, pp. 1–41, 2002. DOI: 10.1017/S0956796801004191.

[18] L. Kristiansen and N. D. Jones, "The flow of data and the complexity of algorithms," in *"New Computational Paradigms"*, S. B. Cooper, B. Löwe, and L. Torenvliet, Eds., ser. LNCS, vol. 3526, Springer, 2005, pp. 263–274, ISBN: 3-540-26179-6. DOI: 10.1007/11494645_33.

[19] C. S. Lee, N. D. Jones, and A. M. Ben-Amram, "The size-change principle for program termination," in *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '01, Association for Computing Machinery, 2001, 81–92, ISBN: 1581133367. DOI: 10.1145/360204.360210.

[20] C. Aubert, T. Rubiano, N. Rusch, and T. Seiller, "Mwp-analysis improvement and implementation: Realizing implicit computational complexity," in *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, A. P. Felty, Ed., ser. LIPIcs, vol. 228, Schloss Dagstuhl, 2022, 26:1–26:23, ISBN: 978-3-95977-233-4. DOI: 10.4230/LIPIcs.FSCD.2022.26.

[21] N. D. Jones and L. Kristiansen, "A flow calculus of *mwp*-bounds for complexity analysis," *ACM Trans. Comput. Log.*, vol. 10, no. 4, 28:1–28:41, 2009. DOI: 10.1145/1555746.1555752.

[22] J. Moyen, T. Rubiano, and T. Seiller, "Loop quasi-invariant chunk detection," in *Automated Technology for Verification and Analysis*, D. D'Souza and K. N. Kumar, Eds., ser. LNCS, vol. 10482, Springer, 2017, ISBN: 978-3-319-68166-5. DOI: 10.1007/978-3-319-68167-2_7.

[23] L.-N. Pouchet and T. Yuki, *PolyBench/C 4.2*. [Online]. Available: https://sourceforge.net/projects/polybench/files/.

[24] C. Aubert, T. Rubiano, N. Rusch, and T. Seiller, "pymwp: A static analyzer determining polynomial growth bounds," in *Automated Technology for Verification and Analysis*, É. André and J. Sun, Eds., ser. LNCS, vol. 14216, Springer, 2023, pp. 263–275, ISBN: 978-3-031-45331-1. DOI: 10.1007/978-3-031-45332-8_14.

[25] F. Pauck, E. Bodden, and H. Wehrheim, "Do android taint analysis tools keep their promises?" In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018, Association for Computing Machinery, 2018, 331–341, ISBN: 978-1-450-35573-5. DOI: 10.1145/3236024.3236029.

[26] K. J. Biba *et al.*, "Integrity considerations for secure computer systems," Mitre Corporation Bedford, MA, Tech. Rep., 1977. [Online]. Available: https://apps.dtic.mil/sti/pdfs/ADA039324.pdf.