SPASCA: Secure-Programming Assistant and Side-Channel Analyzer

Technical Report TUD-CS-2017-0303

November 2017

Ximeng Li, Heiko Mantel, Johannes Schickel, Markus Tasch, Iva Toteva, Alexandra Weber

Technische Universität Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Modeling and Analysis of Information Systems

This work has been funded by the DFG as part of the project Secure Refinement of Cryptographic Algorithms (E3) within the CRC 1119 CROSSING.
This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

CROSSING

CRISP
Center for Research in Security and Privacy

SPASCA: Secure-Programming Assistant and Side-Channel Analyzer

Ximeng Li, Heiko Mantel, Johannes Schickel,
Markus Tasch, Iva Toteva, and Alexandra Weber

Department of Computer Science, TU Darmstadt, Germany
lastname@mais.informatik.tu-darmstadt.de

Abstract Information leakage in computer systems can be clustered in different classes of information leakage. Two examples of such classes are information leakage through data content and information leakage through execution time. This report presents the Secure-Programming Assistant and Side-Channel Analyzer (SPASCA), an information-flow analysis tool for Java source code programs. SPASCA can be used to detect information leakage in programs written in a rich sequential Java subset. It implements two static type-based information-flow analyses: a timing-insensitive analysis detecting only leakage through data content, and a timing-sensitive analysis also detecting information leakage through execution time. The timing-sensitive analysis considers a transcript model of time where the transcript contains the program counter and array accesses. To aid the software engineer in developing secure software, SPASCA is seamlessly integrated as plug-in for the Eclipse IDE.

1 Introduction

In today's computerized society, it is an increasingly pressing concern to avoid unintended information leakage through IT systems. For instance, when a computer program is used, it is important that the user's private information is not leaked by the program to the developer of the program.

Information-flow security [SM03,Man11] aims at providing end-to-end security guarantees for IT systems. For confidentiality, an end-to-end guarantee means that after a system is given access to a secret, it is impossible to leak the secret to a public observer via the execution of the program.

Information leakage in computer systems can be clustered in different classes of information leakage. Two broad classes of information leakage are information leakage through data content and information leakage through execution time. The latter class of information leakage is also called timing side channels. It has been shown that timing side channels could lead to enough leakage of information about cryptographic keys such that an entire key could be revealed to an attacker, e.g. [Koc96].

For the detection of possible information leakage in software, information-flow analysis tools have been developed in the research area of information-flow

security, e.g. [GHM13,LMS+14,BLMS15,GKP+15]. The development of these tools builds on decades of research on the theory of information-flow security.

The static detection of potential information leakage helps to ensure security during software development. Of the underlying techniques used in the tools for static information-flow analysis, security type systems [VSI96] are a prominent one. In a type-based information-flow analysis, the confidentiality of a variable is modeled by a security type of the variable. A program is deemed secure if the program is typable with the security type system. Security type systems are a lightweight technique for information-flow analysis, and often enable the automatic verification of the security of a program or the automatic identification of potential information leaks in a program.

In this report, we present our tool Secure-Programming Assistant and Side-Channel Analyzer (SPASCA), an information-flow analysis tool for Java source code programs. SPASCA can be used to detect information leakage in programs written in a rich sequential subset of Java (with classes and sub-classes, methods, arrays, and expressions with side effects).

SPASCA implements two static type-based information-flow analyses: a timing-insensitive analysis and a timing-sensitive analysis. The two information-flow analyses are formalized as security type systems. The timing-sensitive security type system is an adaptation of the timing-insensitive security type system. It is based on a transcript model of time [MPSW05] where the transcript contains both the program counter and array accesses. By implementing both a timing-insensitive analysis and a timing-sensitive analysis, SPASCA can be used not only to detect regular information leaks via assignments but also to detect timing side channels in a given program.

SPASCA consists not only of an implementation of the two underlying security type systems but is also integrated as plug-in for the Eclipse IDE. With this integration, SPASCA can be used to provide immediate feedback on potential information leaks during development of secure software.

Structure. In Section 2, we introduce the sub-language of Java considered in SPASCA. In Section 3, we introduce the notion of security policies underlying our security type systems. In Section 4, we define the security typing rules of the timing-insensitive security type system as well as the timing-sensitive security type-system implemented in SPASCA. In Section 5, we give an overview of SPASCA's integration into the Eclipse IDE and briefly discuss selected case studies. We discuss related work in Section 6 and conclude in Section 7.

2 Programming Language

In SPASCA, we focus on a sequential object-oriented sub-language of Java. The language is an extension of the language considered in [BLMS15]. For instance, it is extended by arrays, side-effectual expressions, static fields, and static methods.

We denote the set of all possible class names with $\mathcal{C}$, the set of all possible method names with $\mathcal{M}$, the set of a possible field names with $\mathcal{F}$, the set of all

PRIM-TYPES ::= boolean | byte | short | int | long | float | double

TYPES ::= PRIM-TYPES | C | TYPES[ ]

where $C \in \mathcal{C}$

Figure 1. Data Types

possible variable names (including formal method parameters) with $\mathcal{X}$, and the set of all possible primitive values with $\mathcal{V}$. We assume that $\mathcal{M} \cap \mathcal{F} = \emptyset$ holds. We also assume that $Object \in \mathcal{C}$ and $this, result \in \mathcal{X}$ hold. We use *Object* to denote the class Object in Java, use *this* to denote the this-pointer in Java, and *result* as unique variable in a method to identify the return value. Beyond that, we leave the sets $\mathcal{C}$, $\mathcal{M}$, $\mathcal{F}$, $\mathcal{X}$, and $\mathcal{V}$ underspecified.

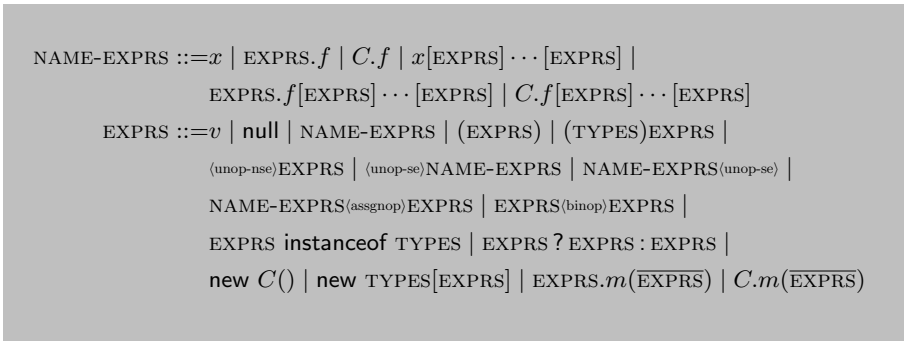
Data Types. Our language is defined based on the set of data types T that is defined by the non-terminal TYPES in the BNF shown in Figure 1. That is, the supported data types are the primitive data types of Java, classes with a name $C \in \mathcal{C}$, and arrays with an arbitrary base type. Note that our definition of data types supports multi-dimensional arrays because $T[\,] \in \mathsf{T}$ for all $T \in \mathsf{T}$. As convention, we write $(T[\,])^n$ for arrays with base type T of dimension n.

Expressions. The expressions of our language are based on the set $UNOP$ of unary operators, the set $BINOP$ of binary operators, and the set $ASSGNOP$ of assignment operators. All operators in these three sets correspond to the syntactically equivalent Java operators.

$$
\begin{aligned}
UNOP\text{-}SE &= \{{+}{+}, -{-}\} \\
UNOP\text{-}NSE &= \{+, -, !, \sim\} \\
UNOP &= UNOP\text{-}SE \cup UNOP\text{-}NSE \\
BINOP &= \{==, !=, <, <=, >, >=, \&, |, \&\&, ||, \hat{}\} \\
&\quad \cup \{+, -, *, /, \%, >>, <<, >>>, <<<\} \\
ASSGNOP &= \{=, +=, -=, *=, /=, \%=, \&=, |=, \hat{}=, <<=, >>>=, >>=\}
\end{aligned}
$$

The unary operators are partitioned into unary operators with potential side-effects (i.e. operators that might lead to changes on the heap) and unary operators without side-effects. The unary operators with potential side-effects are the increment ++ and the decrement --. The unary operators without potential side-effects are the plus + that indicates a positive value, the negation - that negates a value, the logical complement !, and the bitwise complement ~.

The binary operators are the usual comparison operators, the usual logical connectives (including lazy and eager variants of conjunction and disjunction), and the usual arithmetic expressions including both signed and unsigned shifts.



where $v \in \mathcal{V}$, $x \in \mathcal{X}$, $f \in \mathcal{F}$, $m \in \mathcal{M}$, $C \in \mathcal{C}$, ⟨unop-nse⟩ $\in UNOP\text{-}NSE$, ⟨unop-se⟩ $\in UNOP\text{-}SE$, ⟨unop⟩ $\in UNOP$, ⟨binop⟩ $\in BINOP$, and ⟨assgnop⟩ $\in ASSGNOP$

Figure 2. Expressions

The assignment operators are the standard assignment = and the compound assignments that modify the value stored in the referenced variable or field and then reassign the resulting value to the referenced variable or field. Note that all assignment operators have side-effects because a variable or field is overwritten.

The set of expressions E in our language is defined by the non-terminal EXPRS in the BNF shown in Figure 2. In the BNF $\overline{x}$ denotes arbitrarily but finitely many repetitions of a term x. An expression in our language is a literal expression (v or null), a variable access, a (static) field access, an expression in parentheses, a cast expression, a composition of expressions using a unary operator or a binary operator, an instance check, a conditional, an instance creation, or a (static) method call. Note that unary operators with side-effects are only composed with the expressions for variable access and field access.

Statements, Methods Definitions, Class Definitions and Programs. The set of statements S, the set of methods definitions M and the set of class definitions C in our language are, respectively, defined by the non-terminals STMT, MTHDS, and CLASSES in the BNF shown in Figure 3. In the BNF, $\overline{x}$ denotes arbitrarily but finitely many repetitions of a term x. The special statement empty denotes an empty block in Java. Variable declarations (with and without initialization) in our language must occur at the beginning of a statement. Beyond these two special cases, the statements of our language are selected side-effectual expressions, sequential composition, conditional branching (with and without an else-branch), while-loops, and for-loops.

In a non-void method definition, m denotes the name of the method, TYPES declares the return type of the method, $\overline{\text{TYPES } x}$ declares the (potentially empty) list of the method's formal parameters, and the method body is composed out of the declaration of the variable *result*, the body of the method and the return statement. A void method definition is equivalent to a non-void method definition except that no return type is declared and the variable *result* is not declared.

$$
\begin{array}{rl}
\text{DECLS} ::= & x \mid x = \text{EXPRS} \mid \text{DECLS}, \text{DECLS} \\
\text{VAR-DECLS} ::= & \text{TYPES DECLS} \\
\text{STMT} ::= & \mathsf{empty} \mid \text{VAR-DECLS}; \text{STMT} \mid \langle\text{unop-se}\rangle\text{NAME-EXPRS} \mid \text{NAME-EXPRS}\langle\text{unop-se}\rangle \mid \\
& \text{NAME-EXPRS}\langle\text{assgnop}\rangle\text{EXPRS} \mid \text{EXPRS}.m(\overline{\text{EXPRS}}) \mid C.m(\overline{\text{EXPRS}}) \\
& \text{STMT}; \text{STMT} \mid \mathsf{if}\,(\text{EXPRS})\,\{\text{STMT};\}\,\mathsf{else}\,\{\text{STMT};\} \mid \mathsf{if}\,(\text{EXPRS})\,\{\text{STMT};\} \mid \\
& \mathsf{while}\ (\text{EXPRS})\ \{\text{STMT};\} \mid \mathsf{for}(\text{STMT};\ \text{EXPRS};\ \text{STMT})\ \{\text{STMT};\} \\
\text{MTHDS} ::= & \text{TYPES}\ m(\overline{\text{TYPES}\ x})\{\ \text{TYPES}\ \mathit{result} = \text{EXPRS}; \text{STMT}; \mathsf{return}\ \mathit{result};\} \mid \\
& \mathsf{void}\ m(\overline{\text{TYPES}\ x})\{\text{STMT}; \mathsf{return};\} \\
\text{CLASSES} ::= & \mathsf{class}\ C\ \mathsf{extends}\ C'\ \{\overline{\text{TYPES}\ f};\ \overline{\text{MTHDS}}\ \}
\end{array}
$$

where $x \in \mathcal{X}$, $\langle\text{unop-se}\rangle \in UNOP\text{-}SE$, $\langle\text{assgnop}\rangle \in ASSGNOP$, $f \in \mathcal{F}$, $m \in \mathcal{M}$, and $C, C' \in \mathcal{C}$,

Figure 3. Statements, Methods and Classes

In a class definition, C denotes the name of the class, C' declares the intermediate superclass of the class, and the body of the class definition declares its fields and methods.

Based on this syntax of our language, we define programs in our language as a subset of class definitions, i.e. $P \subseteq \mathsf{C}$.

Class Hierarchy of a Program. The class definitions of a program $P \subseteq \mathsf{C}$ indirectly specify the inheritance hierarchy of classes in P. Given a class definition $\mathsf{class}\ C\ \mathsf{extends}\ C'\ \{T_1\ f_1; \ldots T_n\ f_j;\ \ m_1 \ldots m_n\ \} \in P$, the class C is an intermediate subclass of C' in P, written $C \leq^1_P C'$. We write $\leq_P$ for the reflexive, transitive closure of the immediate subclass relation $\leq^1_P$. That is, the relation $\leq_P$ is the subclass relationship for classes defined in P.

A subclass C of a class C' inherits all field declarations and methods definitions from its superclass C'. If a method m defined in C' is redefined in C, then the method is overridden by its new definition in C.

For the remainder of this report, we assume that *Object* is a superclass of all classes defined in a program P, i.e. $C \leq_P \mathit{Object}$ for any class C defined in P.

Well-Formedness. We call a program $P \subseteq \mathsf{C}$ in our language *well-formed* if and only if the following conditions are satisfied:

(1) *Type-correctness:* The program satisfies type-safety conditions commonly imposed by Java compilers.
(2) *Unique names:* Each class has a unique name. Within each class, fields and methods have unique names. Within each method, local variables and formal parameters have unique names.

(3) *Well-formed overriding*: Field names declared in a class are not redeclared in subclasses, and methods are only overridden by methods with the same formal parameters with the same data types.

For the remainder of this report, we only consider well-formed programs.

Semantics. The semantics of programs in our language corresponds to the semantics of an equivalent Java subset. We do not provide a formal semantics for our language in this report. We refer the interested reader to the addendum of [BLMS15] for the formal semantics of a similar Java sublanguage.

Variables, Fields and Methods of a Program. To uniquely identify method names, field names, and variable names across multiple class definitions, we use elements of the set $MID = \mathcal{C} \times \mathcal{M}$ to refer to a method in a specific class, elements of the set $FID = \mathcal{C} \times \mathcal{F}$ to refer to a field in a specific class, and elements of the set $VID = \mathcal{C} \times \mathcal{M} \times \mathcal{X}$ to refer to variables in a specific method of a specific class. That is, intuitively we uniquely identify methods, fields and variables across multiple class definitions by their fully-qualified name.

For a given program $P \subseteq \mathsf{C}$, we use the partial functions $\mathbf{methodsof}_P : \mathcal{C} \rightharpoonup \mathcal{P}(\mathcal{M})$, $\mathbf{fieldsof}_P : \mathcal{C} \rightharpoonup \mathcal{P}(\mathcal{F})$, and $\mathbf{varsof}_P : MID \rightharpoonup \mathcal{P}(\mathcal{X})$ to, respectively, retrieve all method names of methods declared or inherited by the class with name C of P, all field names of fields declared or inherited by the class with name C of P, and all variable names of variables declared as local variables or as formal parameters by a method that is defined in the class with name C of P. More specifically, $\mathbf{methodsof}_P$ is defined for C if and only if P contains a definition of a class with name C. If $\mathbf{methodsof}_P$ is defined for C then $\mathbf{methodsof}_P(C)$ is the set of all method names of methods declared or inherited by the class with name C. The function $\mathbf{fieldsof}_P$ is defined for a class with name C if and only if P contains a definition of a class with name C. If $\mathbf{fieldsof}_P$ is defined for C, then $\mathbf{fieldsof}_P(C)$ is the set of all field names of fields declared or inherited by the class with name C. The function $\mathbf{varsof}_P$ is defined for a method identifier $mid = (C, m)$ if and only if P contains a definition of a class with name C that defines or inherits a method with name m. If $\mathbf{varsof}_P$ is defined for mid, then $\mathbf{varsof}_P(mid)$ is the set of all variables names of variables declared as local variables or as formal parameters by the method with name m. Finally, we use $\mathbf{names}_P$ to denote the set of identifiers of P. We define $\mathbf{names}_P$ by:

$$\begin{aligned}\mathbf{names}_P = &\{(C, m) \in MID \mid m \in \mathbf{methodsof}_P(C)\} \\ &\cup \{(C, f) \in FID \mid f \in \mathbf{fieldsof}_P(C)\} \\ &\cup \{(C, m, x) \in VID \mid x \in \mathbf{varsof}_P((C, m))\}\end{aligned}$$

For a given program $P \subseteq \mathsf{C}$, we use the partial function $\mathbf{parsof}_P : MID \rightharpoonup \mathcal{X}^*$ to retrieve the formal parameters of a method defined in a class of P in their order of declaration. The function $\mathbf{parsof}_P$ is defined for a method identifier $mid = (C, m)$ if and only if P contains a definition of a class with name C

$$
\begin{aligned}
\mathbf{type}_P((C,m),v) &= \mathbf{primtype}(v)\\
\mathbf{type}_P((C,m),\mathsf{null}) &= \mathit{Object}\\
\mathbf{type}_P((C,m),x) &= \mathbf{vtype}_P((C,m,x))\\
\mathbf{type}_P((C,m),E.f) &= \mathbf{ftype}_P((\mathbf{type}_P((C,m),E),f))\\
\mathbf{type}_P((C,m),E.f) &= \mathbf{ftype}_P((C',f))\\
\mathbf{type}_P((C,m),x[E]\dots[E]) &= T \quad \text{if } \mathbf{vtype}_P((C,m,x)) = T[]\dots[]\\
\mathbf{type}_P((C,m),E.f[E]\dots[E]) &= \mathbf{ftype}_P((C',f))\\
\mathbf{type}_P((C,m),C.f[E]\dots[E]) &= \mathbf{ftype}_P((C',f))\\
\mathbf{type}_P((C,m),(E)) &= \mathbf{type}_P((C,m),E)\\
\mathbf{type}_P((C,m),(T)\ E) &= T\\
\mathbf{type}_P((C,m),\langle\text{unop}\rangle E) &= \mathbf{type}_P((C,m),E)\\
\mathbf{type}_P((C,m),E\langle\text{unop}\rangle) &= \mathbf{type}_P((C,m),E)\\
\mathbf{type}_P((C,m),E\langle\text{assgnop}\rangle E') &= \mathbf{type}_P((C,m),E)\\
\mathbf{type}_P((C,m),E\langle\text{binop}\rangle E') &= \mathbf{merge}(\mathbf{type}_P((C,m),E),\\
&\qquad\qquad \mathbf{type}_P((C,m),E'))\\
\mathbf{type}_P((C,m),E\ \mathsf{instanceof}\ T) &= \mathsf{boolean}\\
\mathbf{type}_P((C,m),E\,?\,E' : E'') &= \mathbf{merge}(\mathbf{type}_P((C,m),E'),\\
&\qquad\qquad \mathbf{type}_P((C,m),E''))\\
\mathbf{type}_P((C,m),\mathsf{new}\ C') &= C'\\
\mathbf{type}_P((C,m),\mathsf{new}\ T[E]) &= T[]\\
\mathbf{type}_P((C,m),E.m(T_1\ E_1,\dots,T_n\ E_n)) &= \mathbf{vtype}_P((C',m,\mathit{result}))\\
\mathbf{type}_P((C,m),C'.m(T_1\ E_1,\dots,T_n\ E_n)) &= \mathbf{vtype}_P((C',m,\mathit{result}))
\end{aligned}
$$

where $v \in \mathcal{V}$, $x \in \mathcal{X}$, $T, T_1, \dots, T_n \in \mathsf{T}$, $E, E', E'', E_1, \dots, E_n \in \mathsf{E}$, $\langle\text{unop}\rangle \in UNOP$, $\langle\text{assgnop}\rangle \in ASSGNOP$, $\langle\text{binop}\rangle \in BINOP$, $f \in \mathcal{F}$, $m \in \mathcal{M}$, $C, C' \in \mathcal{C}$, $\mathbf{primtype}(v)$ is the primitive type of v, and $\mathbf{merge}(T, T')$ is the correct merged type of T and T'.

Figure 4. Types of Expressions in a Method

that defines or inherits a method with name m. If $\mathbf{parsof}_P$ is defined for mid, then $\mathbf{parsof}_P(mid) = (x_1, ..., x_n)$ if the method with name m is defined in the class with name C with the list of formal parameters $(x_1, ..., x_n)$ and, otherwise, $\mathbf{parsof}_P((C,m)) = \mathbf{parsof}_P((C',m))$ where $C \leq^1_P C'$.

Concrete Data Types in a Program. For a given program $P \subseteq \mathsf{C}$, we use the partial functions $\mathbf{ftype}_P : FID \rightharpoonup \mathsf{T}$ and $\mathbf{vtype}_P : VID \rightharpoonup \mathsf{T}$ to, respectively, retrieve the data types of fields and variables declared in P. More specifically, the function $\mathbf{ftype}_P$ is defined for a field identifier $fid = (C, f)$ if and only if $f \in \mathbf{fieldsof}_P(C)$. If f is declared with data type T in the definition of the class with name C, then $\mathbf{ftype}_P(fid) = T$. Otherwise, f is declared in a superclass of the class with name C and $\mathbf{ftype}_P(fid) = \mathbf{ftype}_P((C', fid))$ where

C$\leq^1_P$C'. The function $\mathbf{vtype}_P$ is defined for a variable identifier $vid = (C, m, x)$ if and only if $x \in \mathbf{varsof}_P((C, m))$. If the method with name m is defined in the class with name C and x is declared (either as local variable or formal parameter) with date type T in this method, then $\mathbf{vtype}_P(vid) = T$. Otherwise, the method with name m is defined in a superclass of the class with name $\mathcal{C}$ and $\mathbf{vtype}_P(vid) = \mathbf{vtype}_P((C', m, x))$ where C$\leq^1_P$C'.

Utilizing these functions, we use the partial function $\mathbf{type}_P : MID \times \mathsf{E} \rightharpoonup \mathsf{T}$ to retrieve the type of an expression in a method of P (cf. Figure 4).

3 Security Policy and Security Typing

Information-Flow Policy. We capture the permitted flow of information by information-flow policies. An information-flow policy defines a set of security domains $\mathcal{D}$, an interference-relation $\sqsubseteq \subseteq \mathcal{D} \times \mathcal{D}$, and a domain assignment $\mathbf{da} : FID \cup VID \rightharpoonup \mathcal{D}$. The security domains represent abstract levels of confidentiality. The interference-relation defines between which pairs of security domains information flow is permitted. That is, if $\mathsf{d} \sqsubseteq \mathsf{d}'$, then information flow from d to d' is permitted. Complementary, if $\mathsf{d} \not\sqsubseteq \mathsf{d}'$, then information flow from d to d' is forbidden. The domain assignment associates some information containers of a program (i.e. some variables and fields of a program) with a security domain. That is, for two information containers $a, b \in FID \cup VID$ writing information from a into b is permitted if and only if $\mathbf{da}(a) \sqsubseteq \mathbf{da}(b)$.

For the remainder of this report, we consider two-level information-flow policies $(\mathcal{D}, \sqsubseteq, \mathbf{da})$ where $\mathcal{D} = \{\mathsf{low}, \mathsf{high}\}$ and $\sqsubseteq = \{(\mathsf{low}, \mathsf{high}), (\mathsf{low}, \mathsf{low}), (\mathsf{high}, \mathsf{high})\}$. We leave the domain assignment for a concrete program P underspecified but assume that it is *consistent* for P.

Definition 1. *Let $\mathcal{D}$ be a set of security domains. Let $P \subseteq \mathsf{C}$ be a program. Let $g : \mathbf{names}_P \rightharpoonup \mathcal{D}$. The partial function g is consistent for P if and only if for all $C, C' \in \mathcal{C}$ such that $C \leq_P C'$ the following conditions are satisfied:*

(1) for all field names $f \in \mathbf{fieldsof}_P(\mathcal{C}')$, if g is defined for (C, f) and (C', f) then $g(C, f) = g(C', f)$

(2) for all method names $m \in \mathbf{methodsof}_P(\mathcal{C}')$, if g is defined for (C, m) and (C', m) then $g(C, m) = g(C', m)$

(3) for all method names $m \in \mathbf{methodsof}_P(\mathcal{C}')$, and variable names of m's formal parameters $x \in \{x \mid (x_1, \ldots, x_n) = \mathbf{parsof}_P((C', m)) \wedge \exists i \in \{1, ..., n\}.x = x_i\}$, if g is defined for (C, m, x) and (C', m, x), then $g(C, m, x) = g(C', m, x)$

That is, a consistent domain assignment for a program P respects the inheritance hierarchy of P, and the overriding of methods in P.

Security Typing. The domain assignment of an information-flow policy does not necessarily relate all information containers of a given program to a security domain. To check the validity of the security requirements using our security type systems defined in Section 4, a complete security typing of all information containers, including methods is necessary.

Definition 2. *Let $\mathcal{D}$ be a set of security domains. Let $P \subseteq \mathsf{C}$ be a program. A complete security typing of P is a function $t : \mathbf{names}_P \to \mathcal{D}$ consistent for P.*

A security typing t is *compatible* with a given domain assignment $\mathbf{da}$ for a program P if and only if $\mathbf{da}(a) = t(a)$ whenever $\mathbf{da}$ is defined for $a \in \mathbf{names}_P$.

A security typing of P induces what security domains are associated with the this-pointer of a method, the formal parameters of a method, the return value of a method, and the heap effect of a method (i.e. a lower bound on the confidentiality of fields which the method modifies). Given a security typing t of a program P, we use *method signatures* $\mathbf{msig}^t_P$ to capture this induced association of security domains. That is, for a method with name m defined in the class with name C such that $\mathbf{msig}^t_P((C, m)) = (\mathsf{d}_{this}, \mathsf{d}_1, \ldots, \mathsf{d}_n, \mathsf{d}_h, \mathsf{d}_{result})$, the security domain associated with the this-pointer is d_{this}, the security domains associated with the formal parameters are $\mathsf{d}_1, \ldots, \mathsf{d}_n$, the security domain associated with the heap effect is d_h, and the security domain associated with the return value is d_{result}. For a concrete method signature, we write $\langle \mathsf{d}_{this}, (\mathsf{d}_1, \ldots, \mathsf{d}_n), \xrightarrow{\mathsf{d}_h} \mathsf{d}_{result}) \rangle$ instead of $(\mathsf{d}_{this}, \mathsf{d}_1, \ldots, \mathsf{d}_n, \mathsf{d}_h, \mathsf{d}_{result})$.

Furthermore, a security typing of P induces what security domains are associated with the array identifiers of an expression in a specific method. Given a security typing t of a program P, we use the partial function $\mathbf{arrayDoms}^t_P : (MID \times \mathsf{E}) \rightharpoonup \mathcal{P}(\mathcal{D})$ to denote the set of security domains associated with an expression in a specific method. That is, $\mathbf{arrayDoms}^t_P$ is defined for a method identifier $mid = (C, m)$ and an expression E if and only if $m \in \mathbf{methodsof}_P(C)$. If $\mathbf{arrayDoms}^t_P$ is defined for mid and E, then $\mathbf{arrayDoms}^t_P(mid, E)$ is the set of security domains consisting of all security domains associated with an array referenced in E in the method with name m defined in the class with name C.

4 Security Type Systems

The two information-flow analyses implemented in SPASCA are formally defined by two security type systems. The first security type system is a timing-insensitive security type system for the language defined in Section 2. The second security type system is a timing-sensitive adaptation of the timing-insensitive security type system that considers timing leaks caused by confidential conditionals and array accesses on confidential indices. We present the timing-insensitive security type system in Section 4.1 and present its adaptation to the timing-sensitive security type system (i.e., the adapted rules) in Section 4.2.

4.1 Timing-Insensitive Security Type System

Security Typing Judgments. The typability of a given program for a given security typing is defined in terms of the derivability of judgements using the security typing rules. The security type system is defined modularly based on separate judgments for expressions, statements, method definitions, class definitions, and programs.

The judgment for expressions

$$m, C, P; t \vdash E : (\mathsf{r}, \mathsf{ve}, \mathsf{he}) \qquad \text{where } m \in \mathcal{M},\ C \in \mathcal{C},\ P \subseteq \mathsf{C},\ t : \mathbf{names}_P \to \mathcal{D}$$
$$E \in \mathsf{E}, \text{ and } \mathsf{r}, \mathsf{ve}, \mathsf{he} \in \mathcal{D}$$

denotes that for the security typing t, the expression E is associated with the triple of security domains $(\mathsf{r}, \mathsf{ve}, \mathsf{he})$ in the context of the method with name m in the class with name C of the program P. Intuitively, this judgment shall only be derivable if the security domain r is an upper bound on the security domains of variables and fields read in E, the security domain ve is a lower bound on the security domains associated with variables written in E, and the security domain he is a lower bound on the security domains associated with fields written in E. In the remainder of this report, we refer to r as the *read domain*, to ve as the *variable effect domain*, and to he as the *heap effect domain*.

The judgement for statements

$$m, C, P; t \vdash S : (\mathsf{ve}, \mathsf{he}) \qquad \text{where } m \in \mathcal{M},\ C \in \mathcal{C},\ P \subseteq \mathsf{C},\ t : \mathbf{names}_P \to \mathcal{D}$$
$$S \in \mathsf{S}, \text{ and } \mathsf{ve}, \mathsf{he} \in \mathcal{D}$$

denotes that for the security typing t, the statement S is associated with the pair of security domains $(\mathsf{ve}, \mathsf{he})$ in the context of the method with name m in the class with name C of the program P. Like the previous judgement, it shall only be derivable if the security domains ve and he, respectively, are lower bounds on the security domains associated with variables and fields written in S.

The judgement for methods

$$C, P; t \vdash M \qquad \text{where } C \in \mathcal{C},\ P \subseteq \mathsf{C},\ M \in \mathsf{M}, \text{ and } t : \mathbf{names}_P \to \mathcal{D}$$

denotes that for the security typing t, the method M defined in the class with name C of the program P can be typed. This judgement shall only be derivable if the method body only writes to fields that are associated with a lower security domain than the security domain associated with the method M itself.

The judgement for classes

$$P; t \vdash C \qquad \text{where } P \subseteq \mathsf{C},\ C \in \mathsf{C}, \text{ and } t : \mathbf{names}_P \to \mathcal{D}$$

denotes that for the security typing t, the class C of the program P can be typed. This judgement shall only be derivable if all methods defined in C can be typed.

Finally, the judgement for programs

$$t \vdash P \qquad \text{where } P \subseteq \mathsf{C}, \text{ and } t : \mathbf{names}_P \to \mathcal{D}$$

denotes that for the security typing t, all classes composing P can be typed.

A program P is *accepted* by one of our security type systems for a complete security typing t of P if and only if $t \vdash P$ is derivable in the security type system. If a program is accepted, (1) each security domain associated with a variable or field is an upper bound on the security domains of variables and fields from which information is permitted to be written to this variable or field, and (2) each security domain associated with a method is a lower bound on

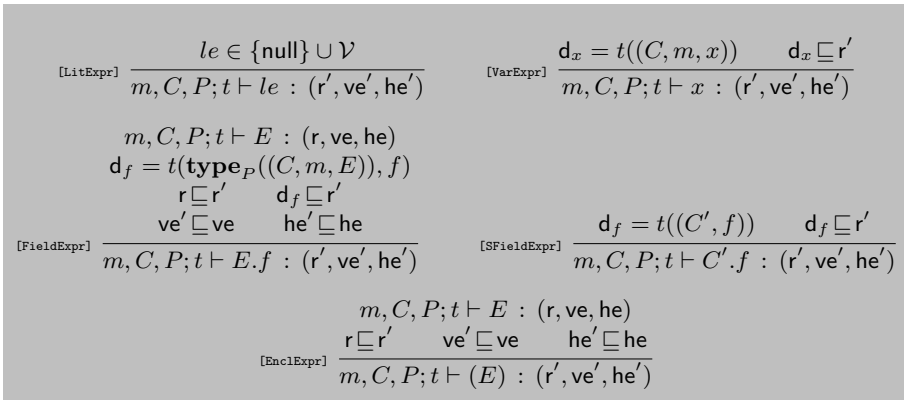


Figure 5. Rules for Literals, Non-Array Named Expressions and Enclosed Expressions

all security domains associated with fields that the method writes. Hence, if a program P is accepted by our security type system for a complete security typing t the program intuitively adheres to information-flow policies $(\mathcal{D}, \sqsubseteq, \mathbf{da})$ for which the complete security typing t is consistent with $\mathbf{da}$.

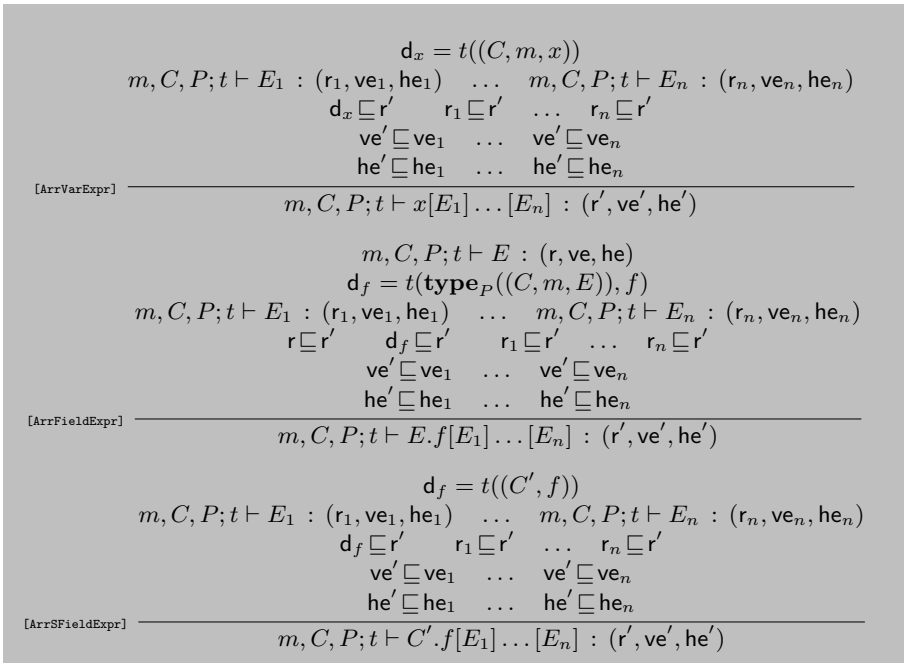
Security Typing Rules for Expressions.

Literals, Non-Array Named Expressions, and Enclosed Expressions. The security typing rules for literal expressions, named expressions referencing no arrays and enclosed expressions are defined by Figure 5. The rule `LitExpr` permits that a literal expression is associated with an arbitrary triple of security domains. Since a literal expression does not read variables or fields, and does not write variables or fields, this faithfully captures the intention of our judgement for expressions.

The rule `VarExpr` permits that a variable reference is associated with any triple of security domains as long as the read domain r' is an upper bound on the security domain d_x associated with the referenced variable. This rule faithfully captures the intention of our judgement for expressions because the only variable or field read is x and neither variables nor fields are written.

The rule `FieldExpr` permits that a non-static field reference is associated with any triple of security domains that is an upper bound on the expression's read domain and the field's security domain, and a lower bound on any potential side effects of the expression. The bounding of the variable effect domain and heap effect domain are necessary because the expression E might write variables and fields, respectively, with security domains ve and he or higher. Since the associated security domain is an upper bound on all variables or fields referenced, and a lower bound on variables and fields written by the non-static field reference, this rule faithfully captures the intention of the judgement.

The rule `SFieldExpr` permits that a static field reference is associated with triples of security domains such that the read domain is an upper bound on the



$$\text{[ArrVarExpr]}\ \frac{\begin{array}{c}\mathsf{d}_x = t((C,m,x))\\ m,C,P;t \vdash E_1 : (\mathsf{r}_1,\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n : (\mathsf{r}_n,\mathsf{ve}_n,\mathsf{he}_n)\\ \mathsf{d}_x \sqsubseteq \mathsf{r}' \qquad \mathsf{r}_1 \sqsubseteq \mathsf{r}' \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{r}'\\ \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n\\ \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n\end{array}}{m,C,P;t \vdash x[E_1]\ldots[E_n] : (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}$$

$$\text{[ArrFieldExpr]}\ \frac{\begin{array}{c}m,C,P;t \vdash E : (\mathsf{r},\mathsf{ve},\mathsf{he})\\ \mathsf{d}_f = t(\mathbf{type}_P((C,m,E)),f)\\ m,C,P;t \vdash E_1 : (\mathsf{r}_1,\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n : (\mathsf{r}_n,\mathsf{ve}_n,\mathsf{he}_n)\\ \mathsf{r} \sqsubseteq \mathsf{r}' \qquad \mathsf{d}_f \sqsubseteq \mathsf{r}' \qquad \mathsf{r}_1 \sqsubseteq \mathsf{r}' \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{r}'\\ \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n\\ \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n\end{array}}{m,C,P;t \vdash E.f[E_1]\ldots[E_n] : (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}$$

$$\text{[ArrSFieldExpr]}\ \frac{\begin{array}{c}\mathsf{d}_f = t((C',f))\\ m,C,P;t \vdash E_1 : (\mathsf{r}_1,\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n : (\mathsf{r}_n,\mathsf{ve}_n,\mathsf{he}_n)\\ \mathsf{d}_f \sqsubseteq \mathsf{r}' \qquad \mathsf{r}_1 \sqsubseteq \mathsf{r}' \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{r}'\\ \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n\\ \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n\end{array}}{m,C,P;t \vdash C'.f[E_1]\ldots[E_n] : (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}$$

Figure 6. Rules for Array Named Expressions

security domain associated with the static field referenced. This rule faithfully captures our intention of the judgement because the only variable or field read is the referenced static field and a static field access has no side effects.

The rule `EnclExpr` permits to adjust the security domains associated with an enclosed expression to an upper bound for the read domain, and lower bounds for the variable effect domain and heap effect domain. The rule is faithful given that the security domains associated with the subexpression are faithfully derived.

Array Named Expressions. The security typing rules for named expressions referencing arrays are defined by Figure 6. The rule `ArrVarExpr` permits that an access to an array referenced by a variable is associated with a triple of security domains such that (1) the read domain is an upper bound on the security domain of the variable and the read domains of the expressions determining the indices of the array access, and (2) the variable effect domain and heap effect domain, respectively, are lower bounds on the variable effect domains and heap effect domains of the expressions determining the indices of the array access. Since the associated triple of security domains is an upper bound on security domains of variables and fields read and a lower bound on the security domains of variables or fields written, the rule faithfully captures the intention of our judgement.

The rule `ArrFieldExpr` is defined analogously to the rule `ArrVarExpr`. That is, the rule permits that an access to an array reference by a field is associated

$$\text{[CastExpr]}\ \frac{m, C, P; t \vdash E\ :\ (\mathsf{r}, \mathsf{ve}, \mathsf{he}) \qquad \mathsf{r} \sqsubseteq \mathsf{r}' \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{he}' \sqsubseteq \mathsf{he}}{m, C, P; t \vdash (T)\ E\ :\ (\mathsf{r}', \mathsf{ve}', \mathsf{he}')}$$

$$\text{[InstExpr]}\ \frac{m, C, P; t \vdash E\ :\ (\mathsf{r}, \mathsf{ve}, \mathsf{he}) \qquad \mathsf{r} \sqsubseteq \mathsf{r}' \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{he}' \sqsubseteq \mathsf{he}}{m, C, P; t \vdash E\ \mathsf{instanceof}\ T\ :\ (\mathsf{r}', \mathsf{ve}', \mathsf{he}')}$$

Figure 7. Rules for Cast and Instance-Check

with a triple of security domains such that (1) the read domain is an upper bound on the security domain associated with the field f and all read domains associated with the expressions composing the array access, and (2) the variable effect domain and heap effect domain, respectively, are lower bounds on the variable effect domains and heap effect domains of all expressions composing the access. The argument of faithfulness is analogous to the one for `ArrVarExpr`.

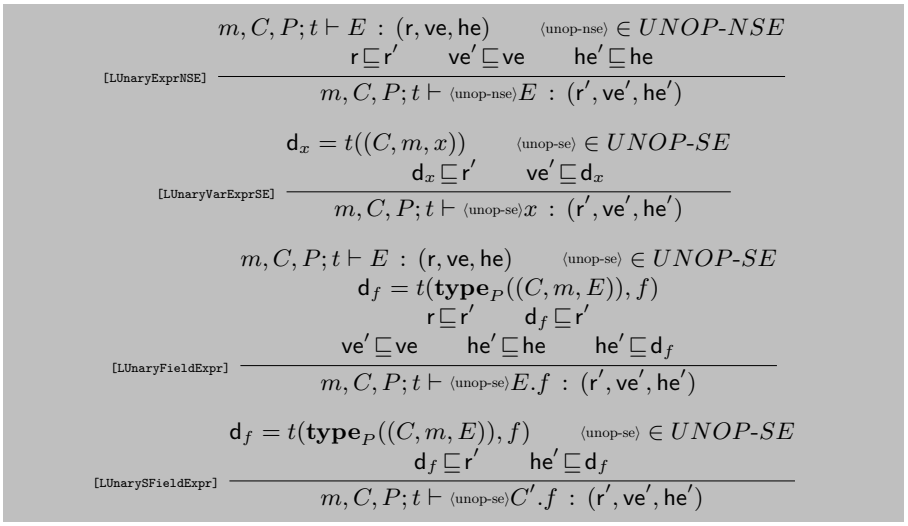
The rule `ArrSFieldExpr` is identical to the rule `ArrFieldExpr` except that the expression determining the object referenced is replaced by a static reference which is not associated with a security domain.

Casts and Instance-Checks. The security typing rules for casts and instance-checks are defined by Figure 7. The rule `CastExpr` and the rule `InstExpr` are equivalent to the rule `EnclExpr` for enclosed expressions. That is, the security domains associated are derived based on the security domains of the respective sub-expressions. Both rules are faithful if the security domains associated with the sub-expressions are faithfully derived.

Left-Unary Expressions with Non-Array References The security typing rules for left-unary expressions using non-array references are defined by Figure 8. The rule `LUnaryExprNSE` permits that a left-unary expression without side effects is associated with triples of security domains such that (1) the read domain is an upper bound on the sub-expression's read domain, and (2) the variable effect domain and the heap effect domain are lower bounds on the domains associated with the sub-expression. The rule is faithful if the security domains associated with the sub-expression are faithfully derived.

The rule `LUnaryVarExpr` permits that a left-unary expression composed out of a variable reference and a unary operator with side effects is associated with triples of security domains such that the read domain is a upper bound on the variable's security domain, and the variable effect domain is a lower bound of the variable's security domain. Since the associated read domain is a upper bound on the security domains of variables read and the variable effect domain is a lower bound on the security domains of variables written, the rule faithfully captures our intention of the judgement for expressions.

The rule `LUnaryFieldExpr` permits that a left-unary expression composed out of a field reference and a unary operator with side effects is associated with



$$\texttt{[LUnaryExprNSE]}\ \frac{m, C, P; t \vdash E : (\mathsf{r}, \mathsf{ve}, \mathsf{he}) \qquad \langle\text{unop-nse}\rangle \in UNOP\text{-}NSE \qquad \mathsf{r} \sqsubseteq \mathsf{r}' \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{he}' \sqsubseteq \mathsf{he}}{m, C, P; t \vdash \langle\text{unop-nse}\rangle E : (\mathsf{r}', \mathsf{ve}', \mathsf{he}')}$$

$$\texttt{[LUnaryVarExprSE]}\ \frac{\mathsf{d}_x = t((C, m, x)) \qquad \langle\text{unop-se}\rangle \in UNOP\text{-}SE \qquad \mathsf{d}_x \sqsubseteq \mathsf{r}' \qquad \mathsf{ve}' \sqsubseteq \mathsf{d}_x}{m, C, P; t \vdash \langle\text{unop-se}\rangle x : (\mathsf{r}', \mathsf{ve}', \mathsf{he}')}$$

$$\texttt{[LUnaryFieldExpr]}\ \frac{m, C, P; t \vdash E : (\mathsf{r}, \mathsf{ve}, \mathsf{he}) \qquad \langle\text{unop-se}\rangle \in UNOP\text{-}SE \qquad \mathsf{d}_f = t(\mathbf{type}_P((C, m, E)), f) \qquad \mathsf{r} \sqsubseteq \mathsf{r}' \qquad \mathsf{d}_f \sqsubseteq \mathsf{r}' \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{he}' \sqsubseteq \mathsf{he} \qquad \mathsf{he}' \sqsubseteq \mathsf{d}_f}{m, C, P; t \vdash \langle\text{unop-se}\rangle E.f : (\mathsf{r}', \mathsf{ve}', \mathsf{he}')}$$

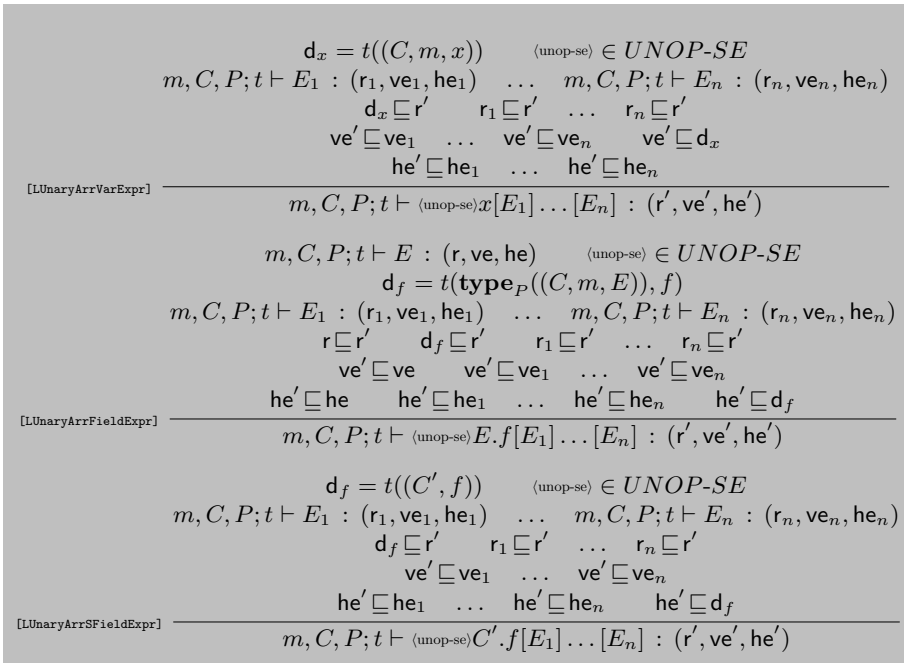
$$\texttt{[LUnarySFieldExpr]}\ \frac{\mathsf{d}_f = t(\mathbf{type}_P((C, m, E)), f) \qquad \langle\text{unop-se}\rangle \in UNOP\text{-}SE \qquad \mathsf{d}_f \sqsubseteq \mathsf{r}' \qquad \mathsf{he}' \sqsubseteq \mathsf{d}_f}{m, C, P; t \vdash \langle\text{unop-se}\rangle C'.f : (\mathsf{r}', \mathsf{ve}', \mathsf{he}')}$$

Figure 8. Rules for Left-Unary Expressions with Non-Array References

triples of security domains such that (1) the read domain is a upper bound on the read domain associated with the sub-expression E and the security domain of the referenced field, (2) the variable effect domain is a lower bound on the variable effect domain associated with the sub-expression E, and (3) the heap effect domain is a lower bound on the heap effect domain associated with the sub-expression E and the security domain of the referenced field. This rule faithfully captures our intention of the judgement for expressions because the read domain is an upper bound on all the security domains of variables or fields read, and the variable effect domain and the heap effect domain, respectively, are lower bounds on all security domains of variables and fields written.

The rule `LUnarySFieldExpr` is similar to the rule `LUnaryFieldExpr`. The key difference is that no sub-expression needs to be taken into account for the derivation of the associated triple of security domains. Hence, the premises corresponding to this sub-expression are removed.

Left-Unary Expressions with Array References The security typing rules for left-unary expressions using array references are defined by Figure 9. The rule `LUnaryArrVarExpr` permits that a left-unary expression composed out of a variable array reference and a unary operator with side effects is associated with triples of security domains such that (1) the read domain is an upper bound on the variable's security domain and all read domains of the expressions determining the accessed array index, (2) the variable effect domain is a lower bound on all variable effect domains of the expressions determining the accessed array index and the variable's security domain, and (3) the heap effect domain is a lower bound on all heap effect domains of the expressions determining the accessed array index. Since a left-unary expression composed out of a variable



$$\frac{\begin{array}{c}\mathsf{d}_x = t((C,m,x)) \qquad \langle\text{unop-se}\rangle \in UNOP\text{-}SE \\ m,C,P;t \vdash E_1 \,:\, (\mathsf{r}_1,\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n \,:\, (\mathsf{r}_n,\mathsf{ve}_n,\mathsf{he}_n) \\ \mathsf{d}_x \sqsubseteq \mathsf{r}' \qquad \mathsf{r}_1 \sqsubseteq \mathsf{r}' \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{r}' \\ \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n \qquad \mathsf{ve}' \sqsubseteq \mathsf{d}_x \\ \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n\end{array}}{m,C,P;t \vdash \langle\text{unop-se}\rangle x[E_1]\ldots[E_n] \,:\, (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}\ \texttt{[LUnaryArrVarExpr]}$$

$$\frac{\begin{array}{c}m,C,P;t \vdash E \,:\, (\mathsf{r},\mathsf{ve},\mathsf{he}) \qquad \langle\text{unop-se}\rangle \in UNOP\text{-}SE \\ \mathsf{d}_f = t(\mathbf{type}_P((C,m,E)),f) \\ m,C,P;t \vdash E_1 \,:\, (\mathsf{r}_1,\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n \,:\, (\mathsf{r}_n,\mathsf{ve}_n,\mathsf{he}_n) \\ \mathsf{r} \sqsubseteq \mathsf{r}' \qquad \mathsf{d}_f \sqsubseteq \mathsf{r}' \qquad \mathsf{r}_1 \sqsubseteq \mathsf{r}' \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{r}' \\ \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n \\ \mathsf{he}' \sqsubseteq \mathsf{he} \qquad \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n \qquad \mathsf{he}' \sqsubseteq \mathsf{d}_f\end{array}}{m,C,P;t \vdash \langle\text{unop-se}\rangle E.f[E_1]\ldots[E_n] \,:\, (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}\ \texttt{[LUnaryArrFieldExpr]}$$

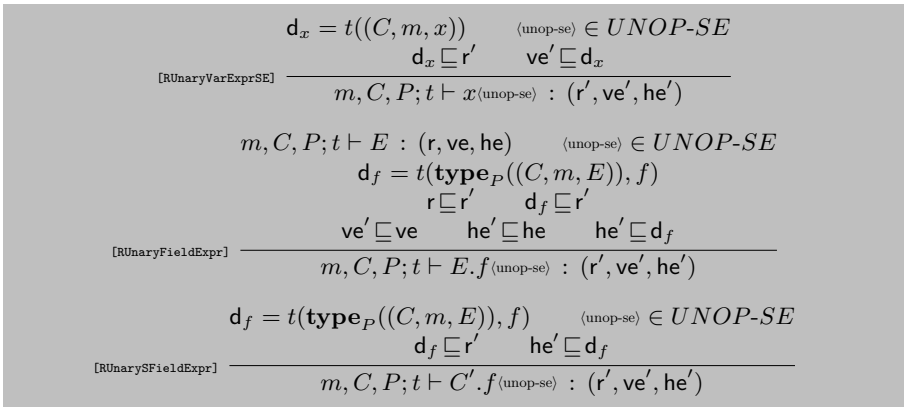
$$\frac{\begin{array}{c}\mathsf{d}_f = t((C',f)) \qquad \langle\text{unop-se}\rangle \in UNOP\text{-}SE \\ m,C,P;t \vdash E_1 \,:\, (\mathsf{r}_1,\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n \,:\, (\mathsf{r}_n,\mathsf{ve}_n,\mathsf{he}_n) \\ \mathsf{d}_f \sqsubseteq \mathsf{r}' \qquad \mathsf{r}_1 \sqsubseteq \mathsf{r}' \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{r}' \\ \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n \\ \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n \qquad \mathsf{he}' \sqsubseteq \mathsf{d}_f\end{array}}{m,C,P;t \vdash \langle\text{unop-se}\rangle C'.f[E_1]\ldots[E_n] \,:\, (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}\ \texttt{[LUnaryArrSFieldExpr]}$$

Figure 9. Rules for Left-Unary Expressions with Array References

array reference and a unary operator with side effects reads and writes the variables and fields read and written by the sub-expressions determining the accessed index, and it reads and writes the variable itself, the rule faithfully captures the intention of the judgement for expressions.

The rule `LUnaryArrFieldExpr` permits that a left-unary expression composed out of a field array reference and a unary operator with side effects is associated with triples of security domains such that (1) the read domain is an upper bound on the field's security domain and all read domains of the expressions determining the accessed field and the accessed array index, (2) the variable effect domain is a lower bound on all variable effect domains of the expressions determining the accessed field and the accessed array index, and (3) the heap effect domain is a lower bound on all heap effect domains of the expressions determining the accessed array index. The rule faithfully captures the intention of the judgement for expressions because the read domain associated with the expression is an upper bound on the security domains of variables and fields read, and the variable effect domain and the heap effect domain, respectively, are lower bounds on the security domains of variables and fields written.

The rule `LUnaryArrSFieldExpr` is similar to the rule `LUnaryArrFieldExpr`. The key difference is that no sub-expression determining the referenced field needs to be taken into account for the derivation of the associated triple of



$$\texttt{[RUnaryVarExprSE]}\ \frac{\mathsf{d}_x = t((C,m,x)) \qquad \langle\text{unop-se}\rangle \in \mathit{UNOP\text{-}SE} \qquad \mathsf{d}_x \sqsubseteq \mathsf{r}' \qquad \mathsf{ve}' \sqsubseteq \mathsf{d}_x}{m,C,P;t \vdash x\langle\text{unop-se}\rangle\ :\ (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}$$

$$\texttt{[RUnaryFieldExpr]}\ \frac{m,C,P;t \vdash E\ :\ (\mathsf{r},\mathsf{ve},\mathsf{he}) \qquad \langle\text{unop-se}\rangle \in \mathit{UNOP\text{-}SE} \qquad \mathsf{d}_f = t(\mathbf{type}_P((C,m,E)),f) \qquad \mathsf{r} \sqsubseteq \mathsf{r}' \qquad \mathsf{d}_f \sqsubseteq \mathsf{r}' \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{he}' \sqsubseteq \mathsf{he} \qquad \mathsf{he}' \sqsubseteq \mathsf{d}_f}{m,C,P;t \vdash E.f\langle\text{unop-se}\rangle\ :\ (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}$$

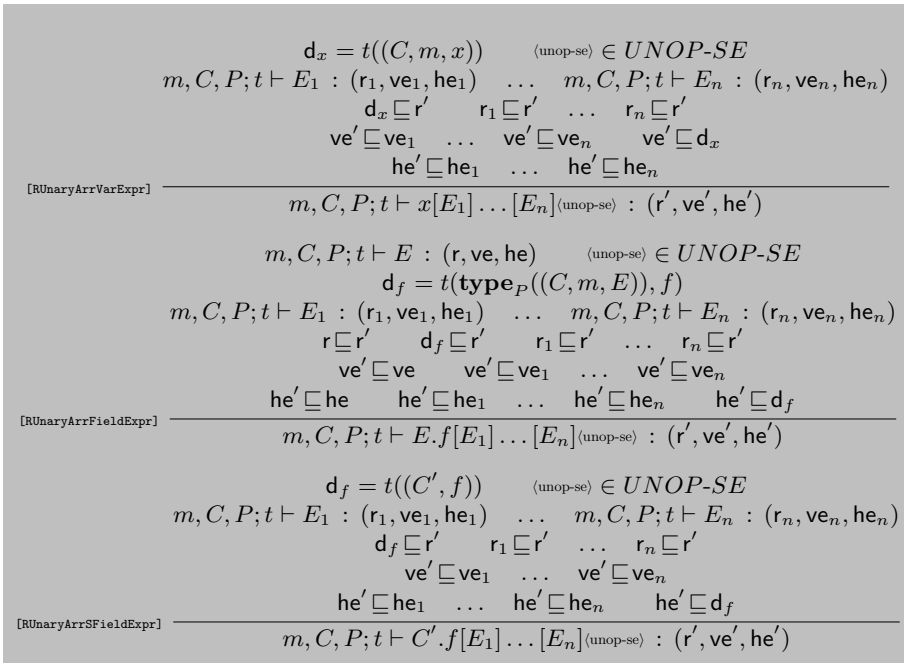
$$\texttt{[RUnarySFieldExpr]}\ \frac{\mathsf{d}_f = t(\mathbf{type}_P((C,m,E)),f) \qquad \langle\text{unop-se}\rangle \in \mathit{UNOP\text{-}SE} \qquad \mathsf{d}_f \sqsubseteq \mathsf{r}' \qquad \mathsf{he}' \sqsubseteq \mathsf{d}_f}{m,C,P;t \vdash C'.f\langle\text{unop-se}\rangle\ :\ (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}$$

Figure 10. Rules for Right-Unary Expressions with Non-Array References

security domains. Hence, the premises corresponding to the sub-expression determining the referenced field are removed.

Right-Unary Expressions. The security typing rules for right-unary expressions are defined by Figure 10 and Figure 11. The rules are identical to the rules for left-unary expressions except that the unary operator is now applied on the right-hand side of the expressions.

Assignments to Variables and Fields. The security typing rules for assignments to variables and fields are defined by Figure 12. The rule `AssgnVarExpr` permits that an assignment to a variable is associated with triples of security domains such that (1) the read domain is an upper bound on the variable's security domain and the read domain of the assigned expression, (2) the variable effect domain is a lower bound on the variable's security domain and the variable effect domain of the assigned expression, and (3) the heap effect domain is a lower bound on the assigned expression's heap effect domain. In addition, it is required that the variable's security domain is an upper bound on the read domain of the assigned expression. In case the variable references an array, it is required that the security domain associated with the variable is a lower bound on the security domains associated with the referenced arrays in the expression E. Together with the premise $\mathsf{r} \sqsubseteq \mathsf{d}_x$, it is ensured that the security domain of the variable is equal to the security domain associated with all arrays referenced in E. This additional restriction is necessary to properly deal with aliasing of arrays. The rule `AssgnVarExpr` faithfully captures our intention for the judgement because the read domain associated with the expression is an upper bound on the security domains of all variables and fields read, and the variable effect domain and the heap effect domain, respectively, are lower bounds on the security domains of variables and fields written. Moreover, an object cannot be assigned to a variable with a lower security domain than the object reference, and arrays can only be assigned to variable with the same security domain than all arrays referenced.



$$\frac{\begin{array}{c}\mathsf{d}_x = t((C,m,x)) \qquad \langle\text{unop-se}\rangle \in UNOP\text{-}SE \\ m,C,P;t \vdash E_1 \,:\, (\mathsf{r}_1,\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n \,:\, (\mathsf{r}_n,\mathsf{ve}_n,\mathsf{he}_n) \\ \mathsf{d}_x \sqsubseteq \mathsf{r}' \qquad \mathsf{r}_1 \sqsubseteq \mathsf{r}' \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{r}' \\ \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n \qquad \mathsf{ve}' \sqsubseteq \mathsf{d}_x \\ \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n\end{array}}{m,C,P;t \vdash x[E_1]\ldots[E_n]\langle\text{unop-se}\rangle \,:\, (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}\ \texttt{[RUnaryArrVarExpr]}$$

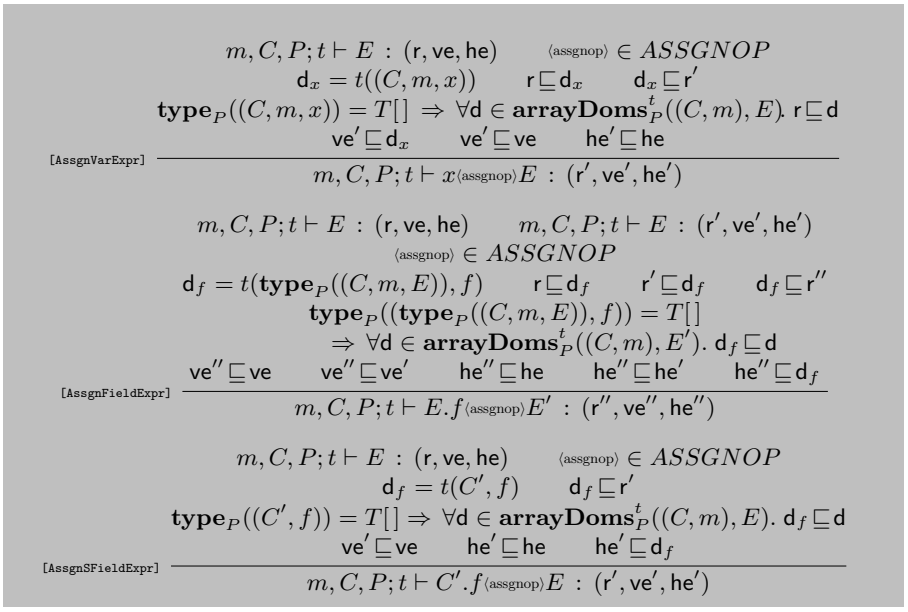
$$\frac{\begin{array}{c}m,C,P;t \vdash E \,:\, (\mathsf{r},\mathsf{ve},\mathsf{he}) \qquad \langle\text{unop-se}\rangle \in UNOP\text{-}SE \\ \mathsf{d}_f = t(\mathbf{type}_P((C,m,E)),f) \\ m,C,P;t \vdash E_1 \,:\, (\mathsf{r}_1,\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n \,:\, (\mathsf{r}_n,\mathsf{ve}_n,\mathsf{he}_n) \\ \mathsf{r} \sqsubseteq \mathsf{r}' \qquad \mathsf{d}_f \sqsubseteq \mathsf{r}' \qquad \mathsf{r}_1 \sqsubseteq \mathsf{r}' \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{r}' \\ \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n \\ \mathsf{he}' \sqsubseteq \mathsf{he} \qquad \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n \qquad \mathsf{he}' \sqsubseteq \mathsf{d}_f\end{array}}{m,C,P;t \vdash E.f[E_1]\ldots[E_n]\langle\text{unop-se}\rangle \,:\, (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}\ \texttt{[RUnaryArrFieldExpr]}$$

$$\frac{\begin{array}{c}\mathsf{d}_f = t((C',f)) \qquad \langle\text{unop-se}\rangle \in UNOP\text{-}SE \\ m,C,P;t \vdash E_1 \,:\, (\mathsf{r}_1,\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n \,:\, (\mathsf{r}_n,\mathsf{ve}_n,\mathsf{he}_n) \\ \mathsf{d}_f \sqsubseteq \mathsf{r}' \qquad \mathsf{r}_1 \sqsubseteq \mathsf{r}' \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{r}' \\ \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n \\ \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n \qquad \mathsf{he}' \sqsubseteq \mathsf{d}_f\end{array}}{m,C,P;t \vdash C'.f[E_1]\ldots[E_n]\langle\text{unop-se}\rangle \,:\, (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}\ \texttt{[RUnaryArrSFieldExpr]}$$

Figure 11. Rules for Right-Unary Expressions with Array References

The rule `AssgnFieldExpr` permits that an assignment to a non-static field is associated with triples of security domains such that (1) the read domain is an upper bound on the field's security domain and the read domain of the expression determining the accessed field, (2) the variable effect domain is a lower bound on the variable effect domain of the assigned expression and the expression determining the accessed field, and (3) the heap effect domain is a lower bound on the heap effect domain of the assigned expressions, the heap effect domain of the expression determining the accessed field and the field's security domain. Additionally, it is required that the field's security domain is an upper bound on the read domain of the assigned expression and the expression determining the accessed field. In case the field references an array, it is required that the field's security domain is a lower bound on the security domains associated with any arrays referenced in the assigned expression. Likewise to the assignment to variables, this requirement ensures together with the premise $\mathsf{r} \sqsubseteq \mathsf{d}_f$ that the field's security domain is equal to the security domain of all arrays referenced by the assigned expression. With these restrictions, the rule `AssgnFieldExpr` faithfully captures our intention of the judgement for expressions properly taking potential aliases of objects and arrays into account.

The rule `AssgnSFieldExpr` for an assignment to a static field is similar to the rule `AssgnFieldExpr`. The key difference is that the referenced field is not determined by an expression but instead by the name of a class. That is, all



$$\texttt{[AssgnVarExpr]}\ \frac{\begin{array}{c} m, C, P; t \vdash E\ :\ (\mathsf{r}, \mathsf{ve}, \mathsf{he}) \qquad \langle \text{assgnop} \rangle \in ASSGNOP \\ \mathsf{d}_x = t((C, m, x)) \qquad \mathsf{r} \sqsubseteq \mathsf{d}_x \qquad \mathsf{d}_x \sqsubseteq \mathsf{r}' \\ \mathbf{type}_P((C, m, x)) = T[\,] \Rightarrow \forall \mathsf{d} \in \mathbf{arrayDoms}^t_P((C, m), E).\ \mathsf{r} \sqsubseteq \mathsf{d} \\ \mathsf{ve}' \sqsubseteq \mathsf{d}_x \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{he}' \sqsubseteq \mathsf{he} \end{array}}{m, C, P; t \vdash x \langle \text{assgnop} \rangle E\ :\ (\mathsf{r}', \mathsf{ve}', \mathsf{he}')}$$

$$\texttt{[AssgnFieldExpr]}\ \frac{\begin{array}{c} m, C, P; t \vdash E\ :\ (\mathsf{r}, \mathsf{ve}, \mathsf{he}) \qquad m, C, P; t \vdash E\ :\ (\mathsf{r}', \mathsf{ve}', \mathsf{he}') \\ \langle \text{assgnop} \rangle \in ASSGNOP \\ \mathsf{d}_f = t(\mathbf{type}_P((C, m, E)), f) \qquad \mathsf{r} \sqsubseteq \mathsf{d}_f \qquad \mathsf{r}' \sqsubseteq \mathsf{d}_f \qquad \mathsf{d}_f \sqsubseteq \mathsf{r}'' \\ \mathbf{type}_P((\mathbf{type}_P((C, m, E)), f)) = T[\,] \\ \Rightarrow \forall \mathsf{d} \in \mathbf{arrayDoms}^t_P((C, m), E').\ \mathsf{d}_f \sqsubseteq \mathsf{d} \\ \mathsf{ve}'' \sqsubseteq \mathsf{ve} \qquad \mathsf{ve}'' \sqsubseteq \mathsf{ve}' \qquad \mathsf{he}'' \sqsubseteq \mathsf{he} \qquad \mathsf{he}'' \sqsubseteq \mathsf{he}' \qquad \mathsf{he}'' \sqsubseteq \mathsf{d}_f \end{array}}{m, C, P; t \vdash E.f \langle \text{assgnop} \rangle E'\ :\ (\mathsf{r}'', \mathsf{ve}'', \mathsf{he}'')}$$

$$\texttt{[AssgnSFieldExpr]}\ \frac{\begin{array}{c} m, C, P; t \vdash E\ :\ (\mathsf{r}, \mathsf{ve}, \mathsf{he}) \qquad \langle \text{assgnop} \rangle \in ASSGNOP \\ \mathsf{d}_f = t(C', f) \qquad \mathsf{d}_f \sqsubseteq \mathsf{r}' \\ \mathbf{type}_P((C', f)) = T[\,] \Rightarrow \forall \mathsf{d} \in \mathbf{arrayDoms}^t_P((C, m), E).\ \mathsf{d}_f \sqsubseteq \mathsf{d} \\ \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{he}' \sqsubseteq \mathsf{he} \qquad \mathsf{he}' \sqsubseteq \mathsf{d}_f \end{array}}{m, C, P; t \vdash C'.f \langle \text{assgnop} \rangle E\ :\ (\mathsf{r}', \mathsf{ve}', \mathsf{he}')}$$

Figure 12. Rules for Assignments to Variables and Fields

premises related to the expression determining the accessed field in the non-static case are removed. Beyond that, the premises of the rule `AssgnSFieldExpr` are equivalent to the premises of the rule `AssgnFieldExpr`.

Assignment to Array Cells. The security typing rules for assignments to array cells are defined by Figure 13. The rule `AssgnArrVarExpr` is defined analogous to the rule `AssgnVarExpr`. The only notable differences are (1) the read domain is also an upper bound on the read domain of the expressions determining the index of the array cell assigned to, and (2) the variable effect domain and heap effect domain, respectively, are also lower bounds on the variable effect domain and heap effect domain of the expressions determining the index of the array cell assigned to. Beyond these differences, the premises of the rule `AssgnArrVarExpr` are identical to the premises of the rule `AssgnVarExpr`.

In the same fashion, the rule `AssgnArrFieldExpr` is defined analogous to the rule `AssgnFieldExpr`. The only notable differences are (1) the read domain is also an upper bound on the read domain of the expressions determining the index of the array cell assigned to, and (2) the variable effect domain and heap effect domain, respectively, are also lower bounds on the variable effect domain and heap effect domain of the expressions determining the index of the array cell assigned to. Beyond these differences, the premises of the rule `AssgnArrFieldExpr` are identical to the premises of the rule `AssgnFieldExpr`.

Finally, the rule `AssgnArrSFieldExpr` for an assignment to a cell of an array referenced by a static field is similar to the rule `AssgnArrFieldExpr`. The

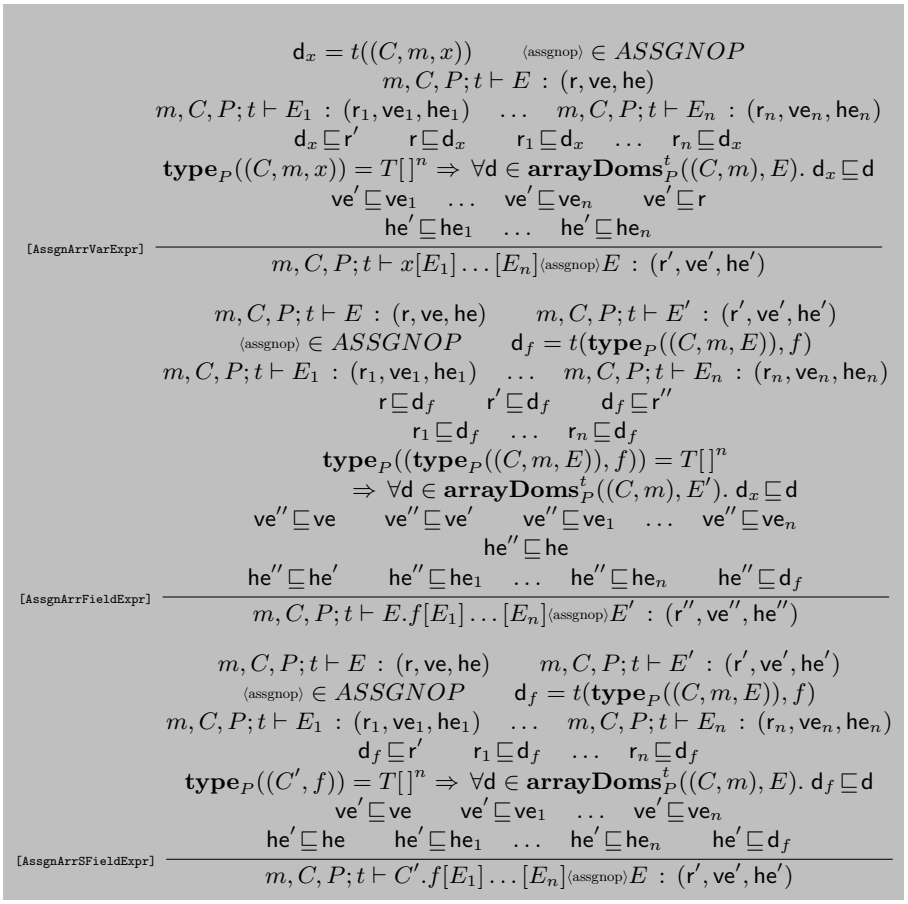
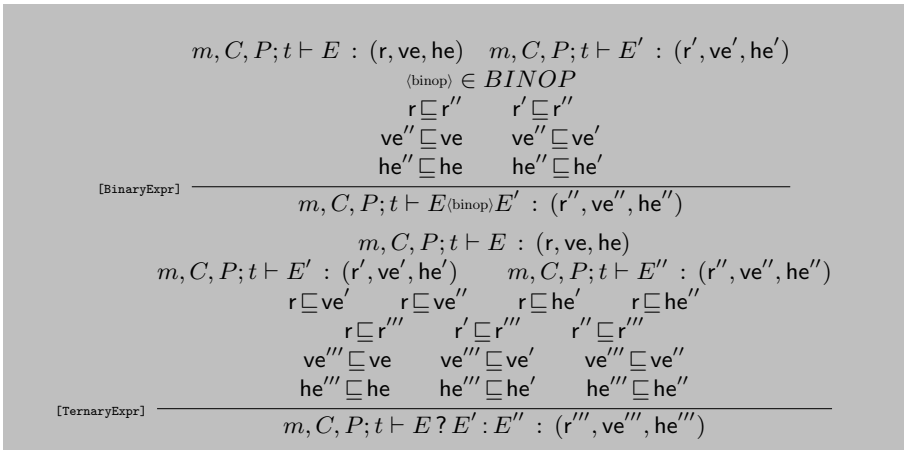


Figure 13. Rules for Assignments to Array Cells

key difference is that the field referencing the array is not determined by an expression but instead by the name of a class. That is, all premises related to the expression determining the field referencing the array in the non-static case are removed. Beyond that, the premises of the rule `AssgnArrSFieldExpr` are equivalent to the premises of the rule `AssgnArrFieldExpr`.

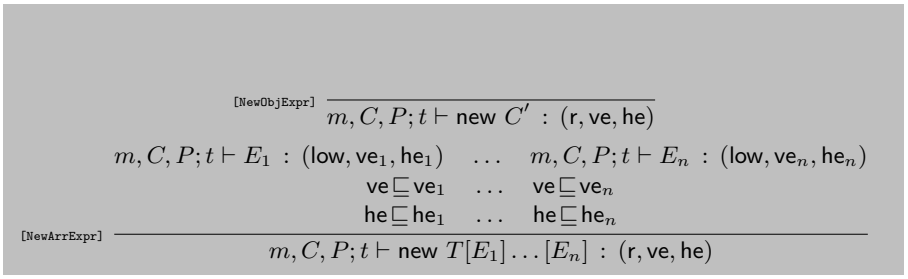
Binary and Ternary Operators. The security typing rules for binary operators and ternary operators are defined by Figure 14. The rule `BinaryExpr` permits that a binary operation is associated with triples of security domains such that the read domain is an upper bound on the read domain of both sub-expressions, and the variable effect domain and heap effect domain, respectively, are lower bounds on the variable effect domain and heap effect domain of both subexpressions. The rule faithfully captures the intention of the judgement for expressions if the judgement for the two sub-expressions is faithfully derived.



$$\text{[BinaryExpr]}\ \frac{\begin{array}{c} m,C,P;t \vdash E\ :\ (\mathsf{r},\mathsf{ve},\mathsf{he}) \quad m,C,P;t \vdash E'\ :\ (\mathsf{r}',\mathsf{ve}',\mathsf{he}') \\ \langle \text{binop} \rangle \in BINOP \\ \mathsf{r} \sqsubseteq \mathsf{r}'' \quad \mathsf{r}' \sqsubseteq \mathsf{r}'' \\ \mathsf{ve}'' \sqsubseteq \mathsf{ve} \quad \mathsf{ve}'' \sqsubseteq \mathsf{ve}' \\ \mathsf{he}'' \sqsubseteq \mathsf{he} \quad \mathsf{he}'' \sqsubseteq \mathsf{he}' \end{array}}{m,C,P;t \vdash E \langle \text{binop} \rangle E'\ :\ (\mathsf{r}'',\mathsf{ve}'',\mathsf{he}'')}$$

$$\text{[TernaryExpr]}\ \frac{\begin{array}{c} m,C,P;t \vdash E\ :\ (\mathsf{r},\mathsf{ve},\mathsf{he}) \\ m,C,P;t \vdash E'\ :\ (\mathsf{r}',\mathsf{ve}',\mathsf{he}') \quad m,C,P;t \vdash E''\ :\ (\mathsf{r}'',\mathsf{ve}'',\mathsf{he}'') \\ \mathsf{r} \sqsubseteq \mathsf{ve}' \quad \mathsf{r} \sqsubseteq \mathsf{ve}'' \quad \mathsf{r} \sqsubseteq \mathsf{he}' \quad \mathsf{r} \sqsubseteq \mathsf{he}'' \\ \mathsf{r} \sqsubseteq \mathsf{r}''' \quad \mathsf{r}' \sqsubseteq \mathsf{r}''' \quad \mathsf{r}'' \sqsubseteq \mathsf{r}''' \\ \mathsf{ve}''' \sqsubseteq \mathsf{ve} \quad \mathsf{ve}''' \sqsubseteq \mathsf{ve}' \quad \mathsf{ve}''' \sqsubseteq \mathsf{ve}'' \\ \mathsf{he}''' \sqsubseteq \mathsf{he} \quad \mathsf{he}''' \sqsubseteq \mathsf{he}' \quad \mathsf{he}''' \sqsubseteq \mathsf{he}'' \end{array}}{m,C,P;t \vdash E\,?\,E' : E''\ :\ (\mathsf{r}''',\mathsf{ve}''',\mathsf{he}''')}$$

Figure 14. Rules for Expressions with Binary and Ternary Operators



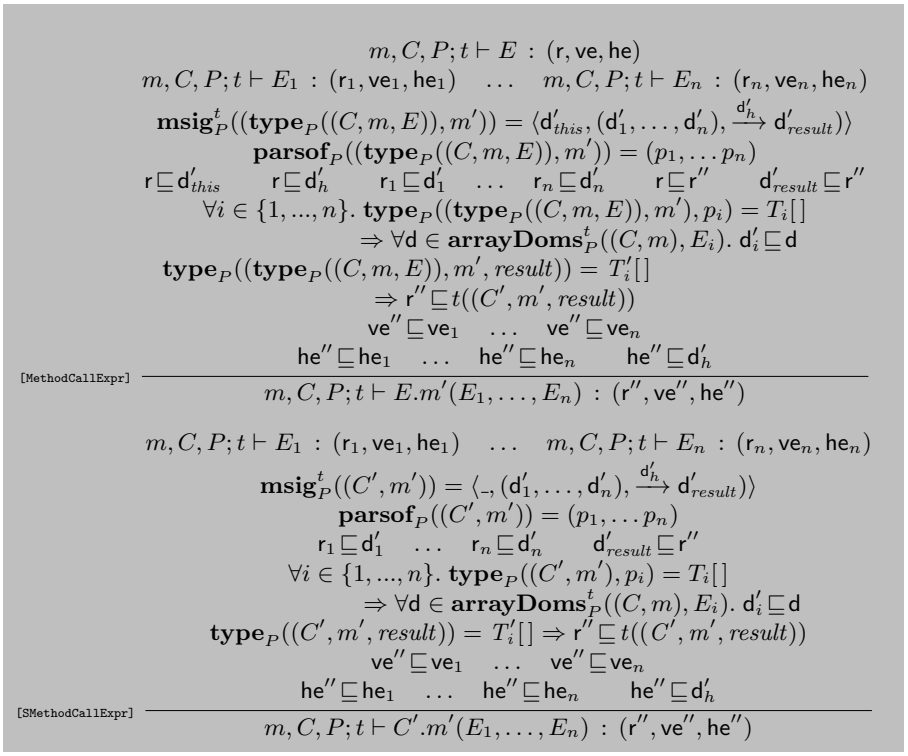
$$\text{[NewObjExpr]}\ \frac{}{m,C,P;t \vdash \mathsf{new}\ C'\ :\ (\mathsf{r},\mathsf{ve},\mathsf{he})}$$

$$\text{[NewArrExpr]}\ \frac{\begin{array}{c} m,C,P;t \vdash E_1\ :\ (\mathsf{low},\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n\ :\ (\mathsf{low},\mathsf{ve}_n,\mathsf{he}_n) \\ \mathsf{ve} \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve} \sqsubseteq \mathsf{ve}_n \\ \mathsf{he} \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he} \sqsubseteq \mathsf{he}_n \end{array}}{m,C,P;t \vdash \mathsf{new}\ T[E_1]\ldots[E_n]\ :\ (\mathsf{r},\mathsf{ve},\mathsf{he})}$$

Figure 15. Rules for Instance-Creation Expressions

The rule `TernaryExpr` permits that a ternary expression is associated with triples of security domains such that (1) the read domain is an upper bound on the read domains associated with all three sub-expressions, and (2) the variable effect domain and the heap effect domain, respectively, are lower bounds on the variable effect domain and heap effect domain of all three sub-expressions. The rule can only be applied if the read domain of the condition permitted to flow to the variable effect domain and heap effect domain of both branches. This restriction is necessary to properly take potential implicit flows into account. The rule `TernaryExpr` is faithful because the associated read domain is an upper bound on the security domains of variables and fields read taking implicit flow into account, and the variable effect domain and heap effect domain, respectively, are lower bounds on variables and fields written.

Instance Creation. The security typing rules for instance creations are defined by Figure 15. The rule `NewObjExpr` permits that an object instance creation is associated with an arbitrary triple of security domains. This treatment is faithful because the expression does neither read or write any variables or fields.



$$\frac{\begin{array}{c} m, C, P; t \vdash E \,:\, (\mathsf{r}, \mathsf{ve}, \mathsf{he}) \\ m, C, P; t \vdash E_1 \,:\, (\mathsf{r}_1, \mathsf{ve}_1, \mathsf{he}_1) \quad \ldots \quad m, C, P; t \vdash E_n \,:\, (\mathsf{r}_n, \mathsf{ve}_n, \mathsf{he}_n) \\ \mathbf{msig}^t_P((\mathbf{type}_P((C, m, E)), m')) = \langle \mathsf{d}'_{this}, (\mathsf{d}'_1, \ldots, \mathsf{d}'_n), \xrightarrow{\mathsf{d}'_h} \mathsf{d}'_{result} \rangle \\ \mathbf{parsof}_P((\mathbf{type}_P((C, m, E)), m')) = (p_1, \ldots p_n) \\ \mathsf{r} \sqsubseteq \mathsf{d}'_{this} \quad \mathsf{r} \sqsubseteq \mathsf{d}'_h \quad \mathsf{r}_1 \sqsubseteq \mathsf{d}'_1 \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{d}'_n \quad \mathsf{r} \sqsubseteq \mathsf{r}'' \quad \mathsf{d}'_{result} \sqsubseteq \mathsf{r}'' \\ \forall i \in \{1, ..., n\}.\ \mathbf{type}_P((\mathbf{type}_P((C, m, E)), m'), p_i) = T_i[\,] \\ \Rightarrow \forall \mathsf{d} \in \mathbf{arrayDoms}^t_P((C, m), E_i).\ \mathsf{d}'_i \sqsubseteq \mathsf{d} \\ \mathbf{type}_P((\mathbf{type}_P((C, m, E)), m', \mathit{result})) = T'_i[\,] \\ \Rightarrow \mathsf{r}'' \sqsubseteq t((C', m', \mathit{result})) \\ \mathsf{ve}'' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}'' \sqsubseteq \mathsf{ve}_n \\ \mathsf{he}'' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}'' \sqsubseteq \mathsf{he}_n \quad \mathsf{he}'' \sqsubseteq \mathsf{d}'_h \end{array}}{m, C, P; t \vdash E.m'(E_1, \ldots, E_n) \,:\, (\mathsf{r}'', \mathsf{ve}'', \mathsf{he}'')}\ [\texttt{MethodCallExpr}]$$

$$\frac{\begin{array}{c} m, C, P; t \vdash E_1 \,:\, (\mathsf{r}_1, \mathsf{ve}_1, \mathsf{he}_1) \quad \ldots \quad m, C, P; t \vdash E_n \,:\, (\mathsf{r}_n, \mathsf{ve}_n, \mathsf{he}_n) \\ \mathbf{msig}^t_P((C', m')) = \langle \_, (\mathsf{d}'_1, \ldots, \mathsf{d}'_n), \xrightarrow{\mathsf{d}'_h} \mathsf{d}'_{result} \rangle \\ \mathbf{parsof}_P((C', m')) = (p_1, \ldots p_n) \\ \mathsf{r}_1 \sqsubseteq \mathsf{d}'_1 \quad \ldots \quad \mathsf{r}_n \sqsubseteq \mathsf{d}'_n \quad \mathsf{d}'_{result} \sqsubseteq \mathsf{r}'' \\ \forall i \in \{1, ..., n\}.\ \mathbf{type}_P((C', m'), p_i) = T_i[\,] \\ \Rightarrow \forall \mathsf{d} \in \mathbf{arrayDoms}^t_P((C, m), E_i).\ \mathsf{d}'_i \sqsubseteq \mathsf{d} \\ \mathbf{type}_P((C', m', \mathit{result})) = T'_i[\,] \Rightarrow \mathsf{r}'' \sqsubseteq t((C', m', \mathit{result})) \\ \mathsf{ve}'' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}'' \sqsubseteq \mathsf{ve}_n \\ \mathsf{he}'' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}'' \sqsubseteq \mathsf{he}_n \quad \mathsf{he}'' \sqsubseteq \mathsf{d}'_h \end{array}}{m, C, P; t \vdash C'.m'(E_1, \ldots, E_n) \,:\, (\mathsf{r}'', \mathsf{ve}'', \mathsf{he}'')}\ [\texttt{SMethodCallExpr}]$$

Figure 16. Rules for Method Calls

The rule `NewArrExpr` permits that an array instance creation is associated with triples of security domains such that the variable effect domain and heap effect domain, respectively, are lower bounds on the variable effect domain and heap effect domain of the sub-expressions determining the array size. The rule forbids the creation of arrays with confidential length (read domain low) to avoid implicit leaks of the secrets involved in computing the array length. This rule is faithful because the variable effect domain and heap effect domain, respectively, are lower bounds on the security domains of all variables and fields written.

Method Calls. The security typing rules for method calls are defined by Figure 16. The rule `MethodCallExpr` permits that a non-static method call is associated with triples of security domains such that (1) the read domain is an upper bound on the read domain of the expression determining on which object the method is called and the security domain associated with the result variable of the method called, (2) the variable effect domain is a lower bound on the variable effect domains of all expressions instantiating the formal parameters of the method called, and (3) the heap effect domain is a lower bound on the heap effect domains of all expressions instantiating the formal parameters of the method called and the security domain associated to the heap effect of the

method called. Additionally, it is required that the read domain of the expression determining on which object the method is called is permitted to flow to the security domain associated with the called method's this-pointer. Furthermore, the read domains of all expressions must be permitted to flow to the security domain associated with the corresponding parameters of the method called.

In case any of the formal parameters is of an array type, the security domain associated with this formal parameter must be permitted to flow to all security domains of arrays referenced in the expression instantiating this formal parameter. Similarly, if the result variable is of an array type, the read domain of the method call must be permitted to flow to the security domain associated with the method's result. These two additional requirements ensure together with the other premises regarding the read domain of the expressions and the result variable's security domain that the security domains of references to arrays are equal to properly deal with aliasing of arrays.

The rule `MethodCallExpr` faithfully captures our intention of the judgement for expressions because it ensures that (1) the method signature induced by the security typing is respected by the method call, (2) the associated read domain is an upper bound on all variables and fields read by the expression determining the object on which the method is called, the expressions instantiating the formal parameters of the method called, and the result variable's security domain, (3) the associated variable effect domain is a lower bound on all variables written by the expression determining the object on which the method is called and the expressions instantiating the formal parameters of the method called, and (4) the associated heap effect domain is a lower bound on all fields written by the expression determining the object on which the method is called, the expressions instantiating the formal parameters of the method called, and the security domain associated with the heap effect of the method called.

The rule `SMethodCallExpr` for static method calls is defined analogous to the rule `MethodCallExpr`. The key difference is that the method is not called on an object determined by an expression but on a class. That is, all premises related to the expression determining the object on which the method is called in the non-static case are removed. Beyond that, the premises of the rule `SMethodCallExpr` are equivalent to the premises of the rule `MethodCallExpr`.

Security Typing Rules for Statements.

Primitive Statements and Sequential Composition. The security typing rules for primitive statements and sequential composition are defined by Figure 17. The rule `EmptyStmt` permits that the empty statement is associated with an arbitrary pair of security domains. This association is faithful because the empty statement writes neither to variables nor to fields.

The rule `ExprStmt` permits that expressions as statements are associated with pairs of security domains under-approximating its variable effect domain and heap effect domain. This rule is faithful assuming that the variable effect domain and heap effect domain derived using the typing rules are, respectively, lower bounds on the security domains of variables and fields written.

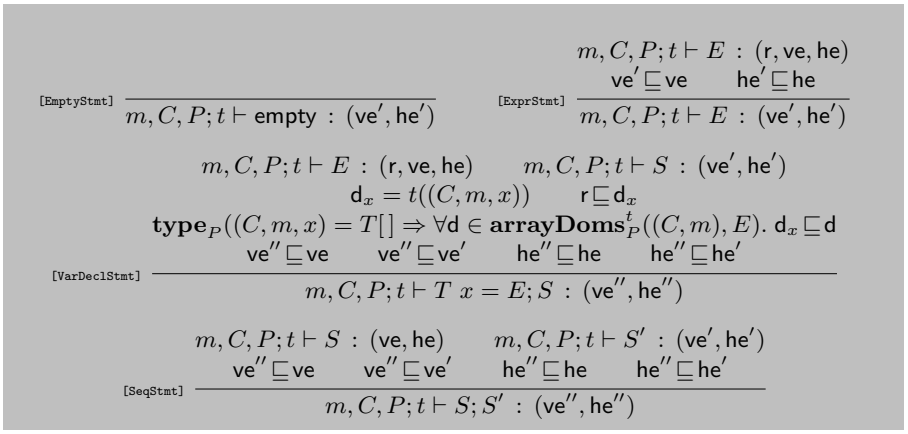
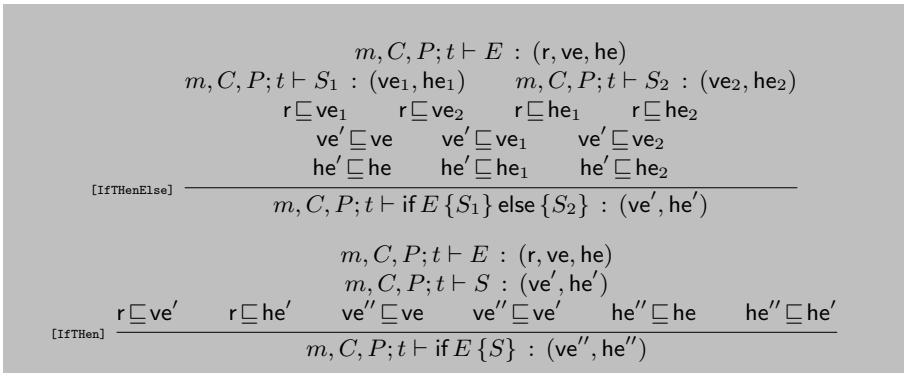


Figure 17. Rules for Primitive Statements and Sequential Composition

The rule `VarDeclStmt` permits that variable declarations are associated with pairs of security domains such that (1) the read domain of the variable initialization is permitted to flow to the security domain of the declared variable, (2) the variable effect domain and the heap effect domain, respectively, are lower bounds on the variable effect domain and heap effect domain of the initialization expression and the consecutive statement, and (3) if the variable declared is of an array type, the security domain of the declared variable is permitted to flow to the security domains associated with arrays referenced in the initialization expression. The third condition is necessary to properly deal with the potential aliasing of arrays. Together with the first condition, it ensures that the security domain of the declared variable is equal to the security domain of any arrays referenced in the initialization expression.

The rule `SeqStmt` permits that sequential compositions of statements are associated with a pair of security domains such that the variable effect domain and heap effect domain, respectively, are lower bounds on the variable effect domain and heap effect domain of the composed statements. This rule is faithful assuming the variable effect domain and heap effect domain derived for the sub-statements are, respectively, lower bounds on the security domains of variables and fields written in the sub-statements as intended for the judgement.

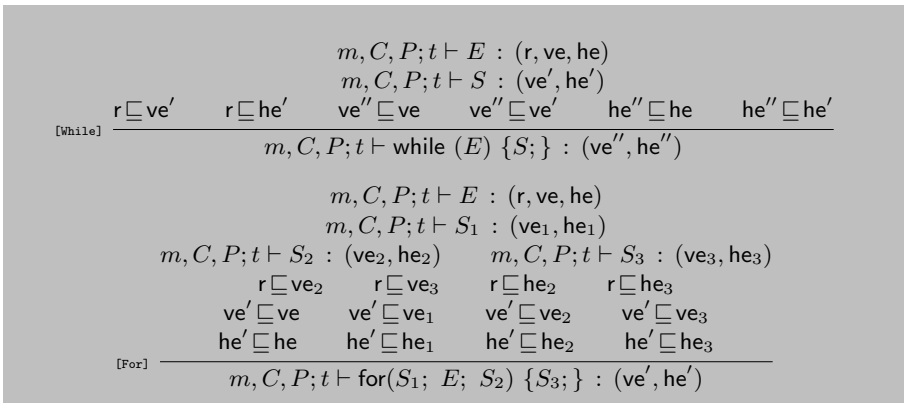
Conditional Branching. The security typing rules for conditional branchings are defined by Figure 18. The rule `IfThenElse` permits that a conditional branching is associated to pairs of security domains such that the variable effect domain and the heap effect domain, respectively, are lower bounds on the variable effect domain and heap effect domain of the condition and the two sub-statements. The rule can only be applied if the read domain of the condition is permitted to flow to the variable effect domain and heap effect domain of the two branches. This rule is faithful because it properly takes the implicit flow from the condition to the variable effect and heap effect of the branches into account, and the variable



$$\text{[IfThenElse]}\ \frac{\begin{array}{c} m,C,P;t \vdash E\ :\ (\mathsf{r},\mathsf{ve},\mathsf{he}) \\ m,C,P;t \vdash S_1\ :\ (\mathsf{ve}_1,\mathsf{he}_1) \qquad m,C,P;t \vdash S_2\ :\ (\mathsf{ve}_2,\mathsf{he}_2) \\ \mathsf{r} \sqsubseteq \mathsf{ve}_1 \quad \mathsf{r} \sqsubseteq \mathsf{ve}_2 \quad \mathsf{r} \sqsubseteq \mathsf{he}_1 \quad \mathsf{r} \sqsubseteq \mathsf{he}_2 \\ \mathsf{ve}' \sqsubseteq \mathsf{ve} \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_2 \\ \mathsf{he}' \sqsubseteq \mathsf{he} \quad \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \mathsf{he}' \sqsubseteq \mathsf{he}_2 \end{array}}{m,C,P;t \vdash \mathsf{if}\ E\ \{S_1\}\ \mathsf{else}\ \{S_2\}\ :\ (\mathsf{ve}',\mathsf{he}')}$$

$$\text{[IfThen]}\ \frac{\begin{array}{c} m,C,P;t \vdash E\ :\ (\mathsf{r},\mathsf{ve},\mathsf{he}) \\ m,C,P;t \vdash S\ :\ (\mathsf{ve}',\mathsf{he}') \\ \mathsf{r} \sqsubseteq \mathsf{ve}' \quad \mathsf{r} \sqsubseteq \mathsf{he}' \quad \mathsf{ve}'' \sqsubseteq \mathsf{ve} \quad \mathsf{ve}'' \sqsubseteq \mathsf{ve}' \quad \mathsf{he}'' \sqsubseteq \mathsf{he} \quad \mathsf{he}'' \sqsubseteq \mathsf{he}' \end{array}}{m,C,P;t \vdash \mathsf{if}\ E\ \{S\}\ :\ (\mathsf{ve}'',\mathsf{he}'')}$$

Figure 18. Rules for Conditional Branchings



$$\text{[While]}\ \frac{\begin{array}{c} m,C,P;t \vdash E\ :\ (\mathsf{r},\mathsf{ve},\mathsf{he}) \\ m,C,P;t \vdash S\ :\ (\mathsf{ve}',\mathsf{he}') \\ \mathsf{r} \sqsubseteq \mathsf{ve}' \quad \mathsf{r} \sqsubseteq \mathsf{he}' \quad \mathsf{ve}'' \sqsubseteq \mathsf{ve} \quad \mathsf{ve}'' \sqsubseteq \mathsf{ve}' \quad \mathsf{he}'' \sqsubseteq \mathsf{he} \quad \mathsf{he}'' \sqsubseteq \mathsf{he}' \end{array}}{m,C,P;t \vdash \mathsf{while}\ (E)\ \{S;\}\ :\ (\mathsf{ve}'',\mathsf{he}'')}$$

$$\text{[For]}\ \frac{\begin{array}{c} m,C,P;t \vdash E\ :\ (\mathsf{r},\mathsf{ve},\mathsf{he}) \\ m,C,P;t \vdash S_1\ :\ (\mathsf{ve}_1,\mathsf{he}_1) \\ m,C,P;t \vdash S_2\ :\ (\mathsf{ve}_2,\mathsf{he}_2) \qquad m,C,P;t \vdash S_3\ :\ (\mathsf{ve}_3,\mathsf{he}_3) \\ \mathsf{r} \sqsubseteq \mathsf{ve}_2 \quad \mathsf{r} \sqsubseteq \mathsf{ve}_3 \quad \mathsf{r} \sqsubseteq \mathsf{he}_2 \quad \mathsf{r} \sqsubseteq \mathsf{he}_3 \\ \mathsf{ve}' \sqsubseteq \mathsf{ve} \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_2 \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_3 \\ \mathsf{he}' \sqsubseteq \mathsf{he} \quad \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \mathsf{he}' \sqsubseteq \mathsf{he}_2 \quad \mathsf{he}' \sqsubseteq \mathsf{he}_3 \end{array}}{m,C,P;t \vdash \mathsf{for}(S_1;\ E;\ S_2)\ \{S_3;\}\ :\ (\mathsf{ve}',\mathsf{he}')}$$

Figure 19. Rules for Loops

effect domain and heap effect domain, respectively, are lower bounds on the security domains of variables and fields written by the condition and the two sub-statements of the conditional branching.

The rule `IfThen` is defined analogously to the rule `IfThenElse`. The only difference is that all premises regarding the missing else-branch are removed.

Loops. The security typing rules for loops are defined by Figure 19. The rule `While` is equal to the rule `IfThen` because the dependency between conditional and body is identical for both statements from an information-flow perspective.

The rule `For` permits that a for-loop is associated with pairs of security domains such that the variable effect domain and the heap effect domain, respectively, are lower bounds on the variable effect domain and heap effect domain of the initialization statement, the condition, the increment statement and the loop body. In addition, the rule requires that the read domain of the condition is permitted to flow to the variable effect domain and heap effect domain of the

$$\text{[Method]}\ \frac{m, C, P; t \vdash T'\ \mathit{result} = E; S\ :\ (\mathsf{ve}, \mathsf{he}) \qquad t(C, m) \sqsubseteq \mathsf{he}}{C, P; t \vdash T\ m(T_1\ x_1, \ldots, T_n\ x_n)\{\ T'\ \mathit{result} = E; S; \mathsf{return}\ \mathit{result}; \}}$$

$$\text{[VoidMethod]}\ \frac{m, C, P; t \vdash S\ :\ (\mathsf{ve}, \mathsf{he}) \qquad t(C, m) \sqsubseteq \mathsf{he}}{C, P; t \vdash \mathsf{void}\ m(T_1\ x_1, \ldots, T_n\ x_n)\{S; \mathsf{return}; \}}$$

$$\text{[Class]}\ \frac{C, P; t \vdash m_1 \quad \ldots \quad C, P; t \vdash m_n}{P; t \vdash \mathsf{class}\ C\ \mathsf{extends}\ C'\ \{T_1\ f_1; \ldots T_n\ f_j; \quad m_1 \ldots m_n\ \}}$$

Figure 20. Rules for Method and Class Definitions

increment statement and the loop body. This restriction is necessary to properly take implicit flows into account. The rule is faithful because implicit flows are taken into account and the associated variable effect domain and heap effect domain, respectively, are lower bounds on the security domains of variables and fields written by the statement.

Security Typing Rules for Method and Class Definitions. The security typing rules for method definitions and class definitions are defined by Figure 20.

The rule `Method` permits that non-void methods are typed if the statement representing the body of the method can be typed and the security domain associated with the method is permitted to flow to the body's heap effect domain. Note that this typing rule only restricts the heap effect domain of the method. Information flows related to the return value of the method are considered by the typing rules `MethodCallExpr` and `SMethodCallExpr` for method calls.

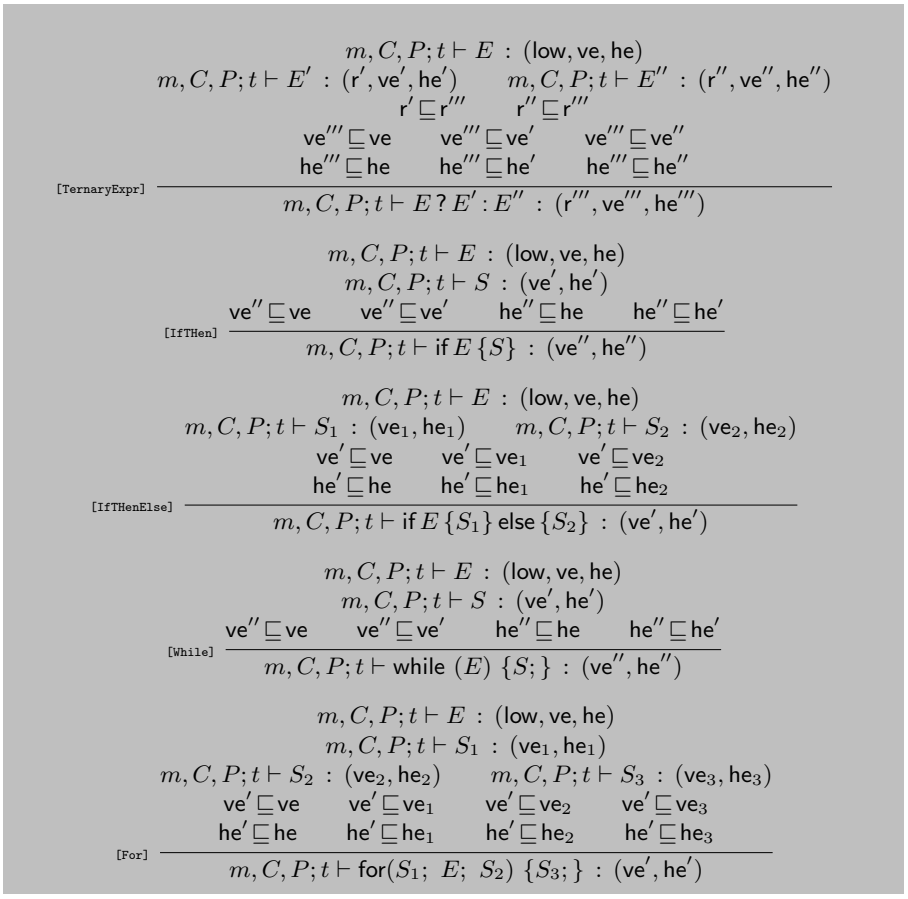
Likewise to the rule `Method`, the rule `VoidMethod` permits void methods to be typed if the statement representing the body of the method can be typed and the security domain associated with the method is permitted to flow to the body's heap effect domain.

Both rules faithfully capture our intention of the judgement for methods because the rules are only applicable if the security domain associated with the method is permitted to flow to the fields written by the body of the method.

The rule `class` permits that classes are typed if all methods defined inside the class are typable using the typing rules for methods. This directly matches our intention of the judgement for classes.

Security Typing Rule for Programs. The security typing rule for programs is defined below. The rule `Prog` requires that all classes composing the program are typable which matches the intention of the judgement for programs.

$$\text{[Prog]}\ \frac{P = \{C_1, \ldots C_n\} \quad P; t \vdash C_1 \quad \ldots \quad P; t \vdash C_n}{t \vdash P}$$



$$\text{[TernaryExpr]}\ \frac{\begin{array}{c} m,C,P;t \vdash E \,:\, (\mathsf{low},\mathsf{ve},\mathsf{he}) \\ m,C,P;t \vdash E' \,:\, (\mathsf{r}',\mathsf{ve}',\mathsf{he}') \qquad m,C,P;t \vdash E'' \,:\, (\mathsf{r}'',\mathsf{ve}'',\mathsf{he}'') \\ \mathsf{r}' \sqsubseteq \mathsf{r}''' \qquad \mathsf{r}'' \sqsubseteq \mathsf{r}''' \\ \mathsf{ve}''' \sqsubseteq \mathsf{ve} \qquad \mathsf{ve}''' \sqsubseteq \mathsf{ve}' \qquad \mathsf{ve}''' \sqsubseteq \mathsf{ve}'' \\ \mathsf{he}''' \sqsubseteq \mathsf{he} \qquad \mathsf{he}''' \sqsubseteq \mathsf{he}' \qquad \mathsf{he}''' \sqsubseteq \mathsf{he}'' \end{array}}{m,C,P;t \vdash E \,?\, E' : E'' \,:\, (\mathsf{r}''',\mathsf{ve}''',\mathsf{he}''')}$$

$$\text{[IfTHen]}\ \frac{\begin{array}{c} m,C,P;t \vdash E \,:\, (\mathsf{low},\mathsf{ve},\mathsf{he}) \\ m,C,P;t \vdash S \,:\, (\mathsf{ve}',\mathsf{he}') \\ \mathsf{ve}'' \sqsubseteq \mathsf{ve} \qquad \mathsf{ve}'' \sqsubseteq \mathsf{ve}' \qquad \mathsf{he}'' \sqsubseteq \mathsf{he} \qquad \mathsf{he}'' \sqsubseteq \mathsf{he}' \end{array}}{m,C,P;t \vdash \mathsf{if}\, E\, \{S\} \,:\, (\mathsf{ve}'',\mathsf{he}'')}$$

$$\text{[IfTHenElse]}\ \frac{\begin{array}{c} m,C,P;t \vdash E \,:\, (\mathsf{low},\mathsf{ve},\mathsf{he}) \\ m,C,P;t \vdash S_1 \,:\, (\mathsf{ve}_1,\mathsf{he}_1) \qquad m,C,P;t \vdash S_2 \,:\, (\mathsf{ve}_2,\mathsf{he}_2) \\ \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve}_2 \\ \mathsf{he}' \sqsubseteq \mathsf{he} \qquad \mathsf{he}' \sqsubseteq \mathsf{he}_1 \qquad \mathsf{he}' \sqsubseteq \mathsf{he}_2 \end{array}}{m,C,P;t \vdash \mathsf{if}\, E\, \{S_1\}\, \mathsf{else}\, \{S_2\} \,:\, (\mathsf{ve}',\mathsf{he}')}$$

$$\text{[While]}\ \frac{\begin{array}{c} m,C,P;t \vdash E \,:\, (\mathsf{low},\mathsf{ve},\mathsf{he}) \\ m,C,P;t \vdash S \,:\, (\mathsf{ve}',\mathsf{he}') \\ \mathsf{ve}'' \sqsubseteq \mathsf{ve} \qquad \mathsf{ve}'' \sqsubseteq \mathsf{ve}' \qquad \mathsf{he}'' \sqsubseteq \mathsf{he} \qquad \mathsf{he}'' \sqsubseteq \mathsf{he}' \end{array}}{m,C,P;t \vdash \mathsf{while}\ (E)\ \{S;\} \,:\, (\mathsf{ve}'',\mathsf{he}'')}$$

$$\text{[For]}\ \frac{\begin{array}{c} m,C,P;t \vdash E \,:\, (\mathsf{low},\mathsf{ve},\mathsf{he}) \\ m,C,P;t \vdash S_1 \,:\, (\mathsf{ve}_1,\mathsf{he}_1) \\ m,C,P;t \vdash S_2 \,:\, (\mathsf{ve}_2,\mathsf{he}_2) \qquad m,C,P;t \vdash S_3 \,:\, (\mathsf{ve}_3,\mathsf{he}_3) \\ \mathsf{ve}' \sqsubseteq \mathsf{ve} \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve}_2 \qquad \mathsf{ve}' \sqsubseteq \mathsf{ve}_3 \\ \mathsf{he}' \sqsubseteq \mathsf{he} \qquad \mathsf{he}' \sqsubseteq \mathsf{he}_1 \qquad \mathsf{he}' \sqsubseteq \mathsf{he}_2 \qquad \mathsf{he}' \sqsubseteq \mathsf{he}_3 \end{array}}{m,C,P;t \vdash \mathsf{for}(S_1;\ E;\ S_2)\ \{S_3;\} \,:\, (\mathsf{ve}',\mathsf{he}')}$$

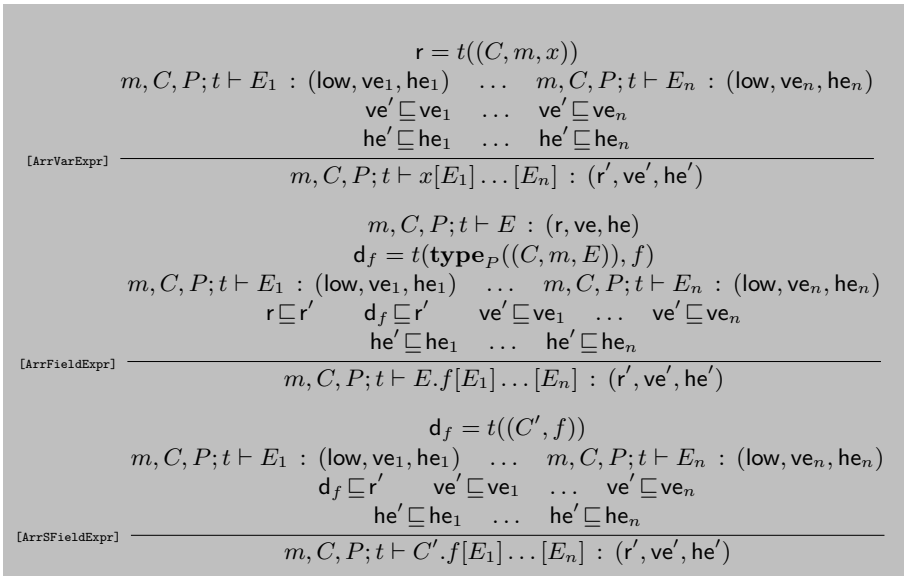
Figure 21. Adapted Rules for Branchings and Loops

4.2 Timing-Sensitive Security Type System

For the adaptation of the timing-insensitive security type system into a timing-sensitive security type system, we consider a coarse grained model of time. This model of time leads to a conservative treatment of potential timing side channels.

Concretely, we consider two causes for timing side-channels: The first cause is branching or looping depending on a confidential condition. That is, whether the then-branch or else-branch is taken for a conditional branching, or whether an iteration of a loop's body is executed depends on confidential information. In these cases, the running time of the conditional branching or the loop might differ. Hence, information about the conditional might be leaked over the statement's timing behavior to an attacker observing a program's running time.

The second cause are read and write accesses to arrays where the chosen index is computed based on confidential information. Depending on prior accesses to



$$\text{[ArrVarExpr]}\ \frac{\begin{array}{c}\mathsf{r} = t((C,m,x)) \\ m,C,P;t \vdash E_1 \,:\, (\mathsf{low},\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n \,:\, (\mathsf{low},\mathsf{ve}_n,\mathsf{he}_n) \\ \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n \\ \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n\end{array}}{m,C,P;t \vdash x[E_1]\ldots[E_n] \,:\, (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}$$

$$\text{[ArrFieldExpr]}\ \frac{\begin{array}{c}m,C,P;t \vdash E \,:\, (\mathsf{r},\mathsf{ve},\mathsf{he}) \\ \mathsf{d}_f = t(\mathbf{type}_P((C,m,E)),f) \\ m,C,P;t \vdash E_1 \,:\, (\mathsf{low},\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n \,:\, (\mathsf{low},\mathsf{ve}_n,\mathsf{he}_n) \\ \mathsf{r} \sqsubseteq \mathsf{r}' \quad \mathsf{d}_f \sqsubseteq \mathsf{r}' \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n \\ \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n\end{array}}{m,C,P;t \vdash E.f[E_1]\ldots[E_n] \,:\, (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}$$

$$\text{[ArrSFieldExpr]}\ \frac{\begin{array}{c}\mathsf{d}_f = t((C',f)) \\ m,C,P;t \vdash E_1 \,:\, (\mathsf{low},\mathsf{ve}_1,\mathsf{he}_1) \quad \ldots \quad m,C,P;t \vdash E_n \,:\, (\mathsf{low},\mathsf{ve}_n,\mathsf{he}_n) \\ \mathsf{d}_f \sqsubseteq \mathsf{r}' \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_1 \quad \ldots \quad \mathsf{ve}' \sqsubseteq \mathsf{ve}_n \\ \mathsf{he}' \sqsubseteq \mathsf{he}_1 \quad \ldots \quad \mathsf{he}' \sqsubseteq \mathsf{he}_n\end{array}}{m,C,P;t \vdash C'.f[E_1]\ldots[E_n] \,:\, (\mathsf{r}',\mathsf{ve}',\mathsf{he}')}$$

Figure 22. Adapted Rules for Array Read Accesses

the same array, the accessed part of the array is currently stored in the cache or in the main memory only. Hence, information about the confidential information used to compute the chosen index might be leaked over the timing behavior of array read and write accesses to an attacker observing a program's running time.

For both causes of potential timing leaks, attacks on cryptographic implementations exploiting these causes of timing side-channels have been reported. An attack exploiting the timing behavior of control flow depending on a confidential condition has been described e.g. in [Koc96]. An attack exploiting the changed timing behavior caused by caching has been described e.g. in [Ber05].

Intuitively, our consideration of these causes for timing side channels corresponds to a transcript model [MPSW05] of time where the transcript contains both the program counter and array accesses.

Branching or Looping Depending on Confidential Conditions. Our timing-sensitive security type system takes a conservative approach for the treatment of confidential conditions in conditional branchings or loops. It forbids any confidential conditions in these statements.

We show the adapted security typing rules in Figure 21. In comparison to the timing-insensitive security typing rules, the key adaptation is that the read domain of the condition must be low. That is, the condition may only read variables and fields associated with the security domain low. Based on this adaptation, we removed any premises requiring that the read domain of the condition is permitted to flow to another security domain.

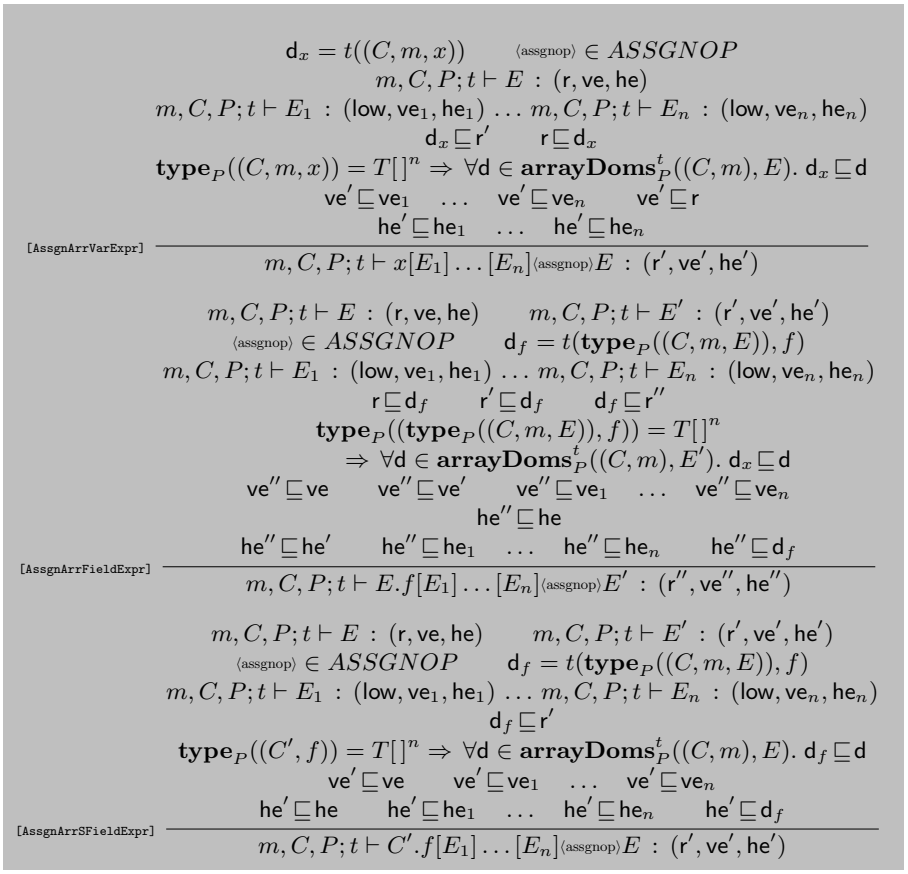


Figure 23. Adapted Rules for Array Assignments

Array Accesses on Confidential Indices. Similar to the treatment of confidential conditions, our timing-sensitive security type system also takes a conservative approach for the treatment of read and write array accesses on indices that are computed based on confidential information. It forbids any array accesses on indices that are computed based on confidential information.

We show the adapted security typing rules for array read accesses and array write accesses (assignments) in Figure 22 and Figure 23. In comparison to the timing-insensitive security typing rules, the key adaptation is that the read domain of all expressions determining the accessed array index must be low. That is, these expressions may only read variables and fields associated with the security domain low. Based on this adaptation, we removed any premises requiring that a read domain of an expression determining the accessed array index is permitted to flow to another security domain.

Beyond the adaptations explained in this section, the timing-sensitive security type system consists of the same rules as the timing-insensitive security type system presented in Section 4.1.

5 Implementation and Case Studies

SPASCA is implemented as a plug-in for the Eclipse IDE for Java. This plug-in can be used to analyze the source code of Java programs that are written in the supported Java sub-language described in Section 2. The analysis is executed in the background in a fully automated fashion, and reports detected potential information leaks and potential timing side channels to the user of SPASCA.

5.1 Implementation of the Eclipse Plug-In

The implementation of SPASCA is building on the implementation of the tool from [BLMS15]. Users of the Eclipse plug-in provide an assignment of security domains for information containers (such as local variables, fields and method parameters) as input. The actual analysis is performed in the background based on these domain assignments. The results of the analysis are presented to the users of the Eclipse plug-in as output in the user interface (UI). To specify security domains of information containers, the SPASCA plug-in offers security-type annotations. Syntactically, these annotations are an instantiation of regular Java annotations and can be either `@High` or `@Low`. Semantically, the security-type annotations map information containers to the security domains for the analysis that are described in Section 3. More concretely, the `@High` annotation assigns containers to the security domain high, and the `@Low` annotation assigns containers to the security domain low. In order to find a complete typing that is compatible with the domain assignments, the implementation of SPASCA uses the type-inference algorithm of from [BLMS15], extended to the typing rules for the SPASCA security type system as presented in Section 4. For targeted analyses on parts of the source code, SPASCA can be configured to perform the analysis on a subset of packages that are present in the target program.

After performing the analysis based on the user input, SPASCA reports back to the UI provided by the Eclipse plug-in. The UI provides detailed information on problems that are found during the analysis. In particular, the UI lists verification errors due to potential information leaks or timing side channels. In addition, the inferred security types for containers are displayed in the UI.

We show a screenshot of the SPASCA Eclipse plug-in for a toy example implementing modular exponentiation using the square-and-multiply approach in Figure 24. In this example, the private key `d` is annotated as `@High`, assigning the information container for the method parameter to the security domain high. Analogously, the ciphertext `y` is annotated as `@Low`, assigning the information container for this method parameter to the security domain low. The lower part of Figure 24 shows the overview of the analysis results for this program. In particular, the UI shows a verification error due to branching on a confidential

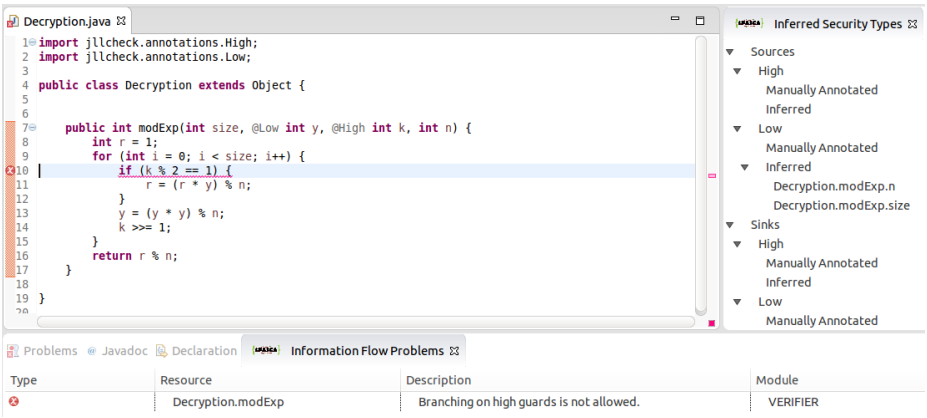


Figure 24. Screenshot of the SPASCA Eclipse plug-in

condition in Line 10. In order to further assist developers, problems are also highlighted in the source code, providing a tool-tip stating the concrete problem. The right part of Figure 24 shows the type inference window that assists developers in making the inferred types of containers explicit.

5.2 Case Studies

We provide case studies evaluating the effectiveness of SPASCA in detecting potential timing side channels in cryptographic libraries. To this end, we analyze three cryptographic implementations from GNU Classpath [GNU09] and FlexiProvider [Fle10]. More concretely, we analyze the GNU Classpath implementation of DES and the FlexiProvider implementations of AES and IDEA. These case studies have previously been evaluated with the Side Channel Finder (SCF) [LS11]. We use SCF's evaluation results as baseline for our evaluation.

SPASCA identified a timing-side-channel vulnerability in each of our case studies. In addition, SPASCA also successfully inferred type information for all information containers in the implementation. We briefly describe the discovered timing-side-channel vulnerabilities in GNU Classpath and FlexiProvider.

The detected vulnerability in GNU Classpath's DES implementation is shown in Figure 25. Here, SPASCA shows an error in Line 25 inside `DESFunGnu`'s `desFunc`. More concretely, SPASCA presents the error "High index access is not allowed". This error message notifies the developer that the array access depends on a container assigned the security domain high. Such an access can lead to differences in the program's execution time an attacker can exploit in a timing-side-channel attack to obtain information on the index accessed. Manual inspection shows that information from the high variable `key` flows into `work` in Line 24. Thus, if an attacker can learn information about `work`, he can, potentially, also learn information about the cryptographic key used.

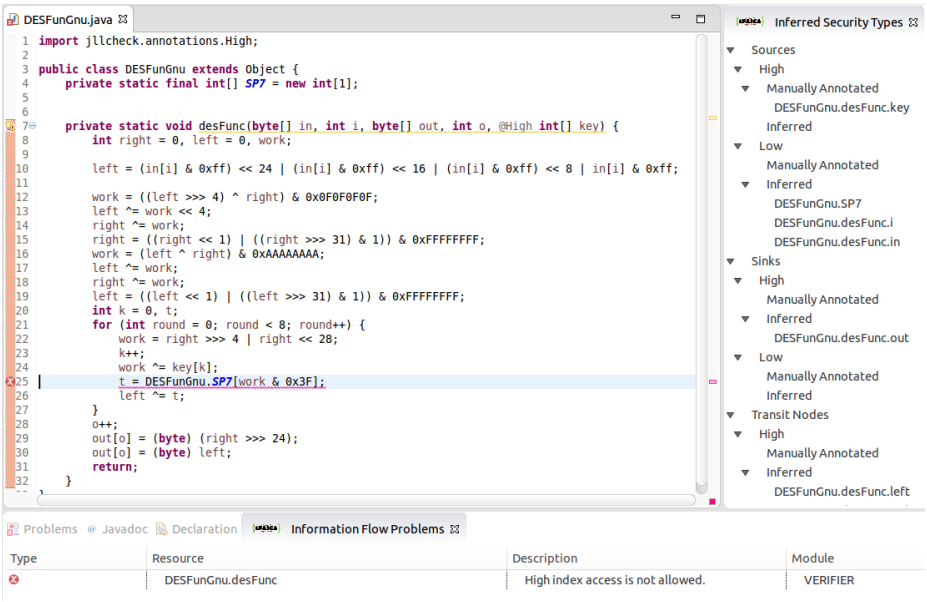


Figure 25. Screenshot of the DES case study (GNU Classpath implementation)

SPASCA detects timing-side-channel vulnerabilities in FlexiProvider's AES and IDEA implementations. In the case of the AES implementation, the detected vulnerability is, again, a secret dependent array access. This is a common cause for a side channel in AES implementations, e.g. the cache-based timing-side-channel attack on AES exploited such an access [Ber05]. In the case of the IDEA implementation, a branch depending on a secret is detected.

The side-channel vulnerabilities in the three implementations were also detected by SCF. This shows that SPASCA can be used to detect timing-side-channel vulnerabilities which are a threat to the security of an implementation. Moreover, the integration into the Eclipse IDE allows developers to easily identify such vulnerabilities and take action to mitigate them.

6 Related Work

Comparable Type-Based Information-Flow Analysis Tools. A number of tools have been developed for the static information-flow analysis of programming languages in practical use. We briefly discuss existing tools that, like SPASCA, are based on security type systems.

Jif [ML97] is one of the earliest type-based information-flow analysis tools for a practical language. It supports a rich Java subset, and a policy language based on security principals (owners, readers, and writers of data). Several advanced features of policy specification are supported by Jif. These features include robust declassification, dynamic information-flow labels, etc. Jif does not support the

detection of timing side channels. In [MUN+16], the policy language of Jif is implemented for the verification of information-flow security in C programs.

Cassandra [LMS+14] is a type-based information-flow security analysis tool for Android applications (in Dalvik bytecode). The tool is based on a security type system that is proven sound wrt. a termination-insensitive variant of noninterference. Cassandra realizes a proof-carrying code [Nec97] architecture: An application is analyzed on the server hosting it, and the certificate resulting from the server-side analysis is then validated at the client installing the application.

The tool from [BLMS15] is a type-based information-flow analysis tool for programs written in a Java subset. The underlying theory of the tool is based on Banerjee and Naumann's typed-based enforcement of noninterference in a class-based object-oriented language [BN05]. The type system used as basis for the implementation of the tool is proven sound wrt. a termination-insensitive noninterference property for the supported Java subset.

With respect to the detection of timing side channels, there exist Side-Channel Finder (SCF) [LS11] and Side Channel Finder$^{\text{AVR}}$ (SCF$^{\text{AVR}}$) [DMW17]. SCF can detect timing side channels in Java source code, while SCF$^{\text{AVR}}$ can detect timing side channels in AVR assembly code. Both tools implement a formally defined security type system as information-flow analysis. For SCF$^{\text{AVR}}$, the formally defined security type system is proven sound wrt. a timing-sensitive variant of noninterference and a formalization of the timing behavior specified in the original AVR instruction set manual.

SPASCA combines both, an information-flow analysis for the detection of information leakage through data content and the detection of timing side channels. It permits a convenient inline specification of security policies unlike, e.g. SCF where policies are specified separated from the program. SPASCA supports a rich subset of Java, going beyond the subset supported by the tool from [BLMS15].

Orthogonal to the object-oriented language supported by SPASCA, type-based information-flow analysis tools have been developed for functional programming languages. One of such developments is Flow Caml [Flo03] – an information-flow analysis tool for the Caml language.

Type-Based Information-Flow Analysis. The theoretical foundation behind type-based information-flow analysis was first developed by Volpano, Smith, and Irvine [VSI96] for a sequential While language. This type-based approach has been advanced extensively by the research community to this very day (for recent developments see, e.g. [CMA17] and [LZ17]). One goal of this subsequent advancement of the type-based approach is to address richer features of the programming language. It follows a brief non-exhaustive list of relevant developments: In [DS04], the treatment of arrays is studied, in [BN05], the treatment of classes and objects is studied, and in [SM02,MC12,LMT17], concurrency and communication are addressed. Another major goal of this subsequent advancement of the type-based approach is to support richer security policies, such as declassification policies and endorsement policies (e.g. [MSZ04,MR07,CMA17]).

Alternative theories on which static information-flow security analyses are based include program dependence graphs and abstract interpretation.

The information-flow analyses tools Joana [GHM13] and JoDroid [MGH15] are based on program dependence graphs (PDGs). In [MS12], it is shown that security type systems and program dependence graphs permit the same level of precision in information-flow analysis.

7 Conclusion

In this report, we have presented the Secure-Programming Assistant and Side-Channel Analyzer (SPASCA). SPASCA is an information-flow analysis tool implementing two formally defined security type systems: a timing-insensitive security type system and a timing-sensitive variant. The formalization and informal justification of the security type systems increase the confidence in the correctness of the information-flow analyses implemented in SPASCA.

SPASCA is seamlessly integrated in the Eclipse IDE as a plug-in. This plug-in provides immediate feedback to the developer regarding potential information-flow problems and potential timing side channels via code highlighting in the editor and error reporting in separate views. The integration of SPASCA in the Eclipse IDE as plug-in enables the usage of the two implemented information-flow analyses by software engineers directly in the software development process.

For the future, we plan to further extend the Java subset supported by SPASCA, e.g. by adding support for concurrency. Moreover, the consideration of additional causes for timing leaks beyond branching on confidential conditions and array accesses on confidential indices is an interesting direction.

Acknowledgments. We would like to thank Manuel Cremer, Yuri Gil Dantas, Richard Gay, Tobias Hamann, Matthias Perner, and Dr. Artem Starostin for helpful discussions and feedback. This work has been funded by the DFG as part of the project Secure Refinement of Cryptographic Algorithms (E3) within the CRC 1119 CROSSING. This work was supported by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within CRISP.

References

[Ber05] D. J. Bernstein. Cache-Timing Attacks on AES. Technical report, 2005.

[BLMS15] D. Bollmann, S. Lortz, H. Mantel, and A. Starostin. An Automatic Inference of Minimal Security Types. In *Proceedings of the 11th International Conference on Information Systems Security*, pages 395–415, 2015.

[BN05] A. Banerjee and D. A. Naumann. Stack-based Access Control and Secure Information Flow. *J. Funct. Program.*, 15(2):131–177, 2005.

[CMA17] E. Cecchetti, A. C. Myers, and O. Arden. Nonmalleable Information Flow Control. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS*, pages 1875–1891, 2017.

[DMW17] F. Dewald, H. Mantel, and A. Weber. AVR Processors as a Platform for Language-Based Security. In *Proceedings of the 22nd European Symposium on Research in Computer Security (ESORICS)*, pages 427–445, 2017.

[DS04] Z. Deng and G. Smith. Lenient Array Operations for Practical Secure Information Flow. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop, CSFW*, pages 115–124. IEEE, 2004.

[Fle10] FlexiProvider – A Toolkit for the Java Cryptography Architecture (JCA/JCE). see `http://www.flexiprovider.de`, 2010.

[Flo03] Flow caml. https://www.normalesup.org/~simonet/soft/flowcaml/, 2003.

[GHM13] J. Graf, M. Hecker, and M. Mohr. Using JOANA for Information Flow Control in Java Programs - A Practical Guide. In *Proceedings of the 6th Working Conference on Programming Languages*, pages 123–138, 2013.

[GKP+15] M. I. Gordon, D. Kim, J. H. Perkins, L. Gilham, N. Nguyen, and M. C. Rinard. Information Flow Analysis of Android Applications in DroidSafe. In *Proceedings of the 22nd Annual Network and Distributed System Security Symposium*, 2015.

[GNU09] GNU Classpath. see `http://www.gnu.org/software/classpath/`, 2009.

[Koc96] P. C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Proceedings of the 1996 Internation Conference on Advances in Cryptology (CRYPTO)*, LNCS 1109, pages 104–113. Springer, 1996.

[LMS+14] S. Lortz, H. Mantel, A. Starostin, T. Bähr, D. Schneider, and A. Weber. Cassandra: Towards a Certifying App Store for Android. In *Proceedings of the 4th ACM CCS Workshop on Security and Privacy in Smartphones and Mobile Devices*, pages 93–104, 2014.

[LMT17] X. Li, H. Mantel, and M. Tasch. Taming Message-Passing Communication in Compositional Reasoning About Confidentiality. In *Proceedings of the 15th Asian Symposium on Programming Languages and Systems, APLAS*, pages 45–66, 2017.

[LS11] A. Lux and A. Starostin. A Tool for Static Detection of Timing Channels in Java. volume 1, pages 303–313, 2011.

[LZ17] P. Li and D. Zhang. Towards a Flow- and Path-Sensitive Information Flow Analysis. In *30th IEEE Computer Security Foundations Symposium, CSF*, pages 53–67, 2017.

[Man11] H. Mantel. Information Flow and Noninterference. In *Encyclopedia of Cryptography and Security, 2nd Ed.*, pages 605–607. 2011.

[MC12] S. Muller and S. Chong. Towards a Practical Secure Concurrent Language. In *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 57–74, 2012.

[MGH15] M. Mohr, J. Graf, and M. Hecker. JoDroid: Adding Android Support to a Static Information Flow Control Tool. In *Gemeinsamer Tagungsband der Workshops der Tagung Software Engineering 2015*, volume 1337, pages 140–145, 2015.

[ML97] A. C. Myers and B. Liskov. A Decentralized Model for Information Flow Control. In *SOSP '97*, pages 129–142, 1997.

[MPSW05] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology (ICISC)*, pages 156–168, 2005.

[MR07] H. Mantel and A. Reinhard. Controlling the What and Where of Declassification in Language-Based Security. In *Proceedings of the 16th European Symposium on Programming, Programming Languages and Systems, ESOP*, pages 141–156, 2007.

[MS12] H. Mantel and H. Sudbrock. Types vs. PDGs in Information Flow Analysis. In *Proceedings of the 22nd International Symposium Logic-Based Program Synthesis and Transformation, LOPSTR*, pages 106–121, 2012.

[MSZ04] A. C. Myers, A. Sabelfeld, and S. Zdancewic. Enforcing Robust Declassification. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop, (CSFW*, pages 172–186, 2004.

[MUN+16] K. Muller, S. Uhrig, F. Nielson, H. R. Nielson, X. Li, M. Paulitsch, and G. Sigl. Automatic Information Flow Validation for High Assurance Systems. *International Journal on Advances in Software*, 9(3&4):191–206, 2016.

[Nec97] G. C. Necula. Proof-carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 106–119, 1997.

[SM02] A. Sabelfeld and H. Mantel. Securing Communication in a Concurrent Language. In *Proceedings of the 9th International Symposium on Static Analysis, SAS*, pages 376–394, 2002.

[SM03] A. Sabelfeld and A. C. Myers. Language-based Information-flow Security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[VSI96] D. M. Volpano, G. Smith, and C. E. Irvine. A Sound Type System for Secure Flow Analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.