A Translation Layer for Information Flow Verification Systems: Bridging Type Systems with Theorem Provers

Bachelor's Thesis by

Felix Mühlenberend

at the KIT Department of Informatics
Institute of Information Security and Dependability (KASTEL)

Reviewer:	Prof. Dr. Bernhard Beckert
Advisor:	Florian Lanzinger, M. Sc.

01 May 2023 – 01 September 2023

Karlsruher Institut für Technologie
KIT-Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Abstract

This bachelor's thesis addresses the challenge of Information Flow Control (IFC) in software engineering, with a focus on balancing expressiveness and usability. It identifies a common compromise in current IFC methods between expressiveness and simplicity. Inspired by an existing combined approach that leverages system-dependency-graphs and a theorem solver, the thesis proposes a novel approach combining the simplicity of type systems with the expressiveness of theorem provers. The combined system permits the direct incorporation of information flow specifications, in the form of type annotations, into the source code. Many of these can be automatically inferred, thereby lowering the annotation overhead while harnessing the power of a deductive verification system when the type system falls short. The mathematical soundness of this approach is demonstrated, and a practical implementation, employing the SFLow type system and the KeY theorem prover, is presented. A series of comprehensive unit tests, encompassing various types of information flow and language elements, demonstrates that this method facilitates efficient specification and comprehensive verification of information flow. This advocates for further research into the practicality and potential implications of this method.

Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Herausforderung der Informationsflusskontrolle (IFC) in der Softwareentwicklung, mit einem Fokus auf das Gleichgewicht zwischen Ausdruckskraft und Benutzerfreundlichkeit. Sie identifiziert einen gängigen Kompromiss in aktuellen IFC-Methoden zwischen Ausdruckskraft und Einfachheit. Inspiriert von einem bestehenden kombinierten Ansatz, der Systemabhängigkeitsgraphen und einen Theoremlöser nutzt, schlägt die Arbeit einen neuartigen Ansatz vor. Dieser Ansatz kombiniert der die Einfachheit von Typsystemen mit der Ausdruckskraft von Theoremlösern. Das kombinierte System ermöglicht die direkte Spezifikation des erlaubten Informationsflusses in Form von Typannotationen im Quellcode. Viele davon können automatisch abgeleitet werden, wodurch der Spezifikationsaufwand reduziert wird. Die Leistungsfähigkeit eines deduktiven Verifikationssystems wird genutzt, wenn das Typsystem an seine Grenzen stößt. Die mathematische Stichhaltigkeit dieses Ansatzes wird demonstriert und eine praktische Implementierung, die das SFLow-Typsystem und den KeY-Theoremlöser verwendet, wird vorgestellt. Eine Reihe von umfassenden Unit-Tests, die verschiedene Informationsflussmechanismen und Sprachelemente umfassen, zeigt, dass diese Methode effiziente Spezifikation und Verifikation von unerwünschten Informationsflüssen ermöglicht. Dies spricht für weitere Forschungen zur Praktikabilität und zu den potenziellen Auswirkungen dieser Methode.

Contents

List of Figures ix

1 Introduction 1

2 Theoretical Foundations 3
- 2.1 Security Lattices . . . 3
- 2.2 Type Systems . . . 5
- 2.3 The Core Language . . . 7
- 2.4 Volpano Smith Secure Flow Type System . . . 8
- 2.5 Information Flow Verification in KeY . . . 11

3 Mathematical Formulation of the Translation Layer 17
- 3.1 The Basic Lattice . . . 17
- 3.2 Noninterference in the Context of the Basic Lattice . . . 18
- 3.3 Equivalence of the Noninterference Formulations with Respect to the Basic Lattice . . . 19
- 3.4 Combined Verification System . . . 20

4 Implementation 23
- 4.1 Architecture . . . 23
- 4.2 Requirements . . . 24
- 4.3 The Checker Framework . . . 25
- 4.4 The SFlow Type System . . . 26
- 4.5 JML Information Flow Contracts in KeY . . . 27
- 4.6 Extending SFlow . . . 28
- 4.7 Translation Layer . . . 36

5 Evaluation 39

6 Conclusion 43

Bibliography 45

List of Figures

2.1 Security lattice combining secrecy and integrity 4
2.2 Simple security lattice combining secrecy and integrity 4
2.3 Semantics of the core language 13
2.4 Syntax-directed typing rules for the core language 14
2.5 Subtyping rules 15

3.1 The basic lattice 17

4.1 Architecture of the complete verification system 23
4.2 Selective declassification in SFlow 27
4.3 Example of a `determines` clause 27
4.4 Program declassifying the sum of an array 28
4.5 Recursive type inference 29
4.6 Recursion for blocks 30
4.7 Recursion for conditionals 31
4.8 Recursion for loops 32
4.9 Assignment base case 33
4.10 Example of implicit flow due to early termination of a loop 33
4.11 Control flow base case 34
4.12 Example of cyclic method calls 34
4.13 Modified typecheking entry point 34
4.14 Example of a method declaration with conflicting annotations 35
4.15 Example of a method with accidental return type declaration 35
4.16 Example for dynamic dispatch causing implicit flow 36

5.1 Test case for IF′ needing to be typed **safe** *cmd* with a safe condition and safe branches 40
5.2 Test case for IF′ needing to be typed **tainted** *cmd* with a safe condition and safe branches 40
5.3 False positive 41

1 Introduction

Ensuring the secrecy of sensitive information and the integrity of important data has become increasingly important in the field of software engineering. A primary objective, in this context, is to prevent the flow of information from high-security data to low-security outputs. The concept of Information Flow Control (IFC), which can be traced back to the works of Denning (1976) and Bell (1973), is employed to achieve this goal.

In the domain of information flow control, individual program entities are assigned *labels*. These labels allow for the specification of *information flow policies*, designed to identify and restrict invalid information flows. Unauthorized flows can occur through explicit assignments between variables of differing security levels, such as low-security and high-security variables. However, unauthorized flows can also occur indirectly and often through complex pathways, making the task of identifying such flows challenging in real-world scenarios. To address this challenge, formal methods are employed. These methods serve a dual purpose in this regard - they are not only capable of detecting information flow but, perhaps more importantly, can also prove its absence.

The formal methods employed in this context can be primarily categorized into syntactic and semantic approaches. Syntactic approaches focus on the structure of program code and include *system-dependency-graph* (SDG) based techniques (refer to Graf, Hecker, and Mohr (2013)), *type systems* (refer to Volpano, Smith, and Irvine (2000)), and methods based on the *decentralized label* model (DLM) (refer to Myers and Liskov (2000)). Conversely, semantic methodologies, which deal with the interpretation of programs, frequently utilize logic-based approaches. One such logic is as *dynamic logic for Java* (JavaDL) (refer to Beckert, Bruns, et al. (2014)), which can be automatically verified with theorem provers.

Syntactic approaches are inherently less precise than their semantic counterparts. For instance, in the case of `l = h; l = 0`, there is no information flow from `h` to `l`, yet most syntactic approaches would still flag this as a false positive. Moreover, there exists a trade-off between the expressiveness of the information flow specification and the ease of use of the validation tools. Kozyri, Chong, and Myers (2022), in a comprehensive review of information flow research over the past three decades, observe the following:

> The inherent tension between expressiveness and simplicity is apparent in the literature of information flow policies. The more expressive the policies and labels are, the more precise the specification of allowed or forbidden flows becomes, enabling a more fine-grained control of information flow. However, increasingly expressive policies and labels become less understandable and possibly less usable. To widen the adoption of information flow policies by programmers, a balance must be struck between expressiveness and usability.

One potential solution to balancing expressiveness and usability is to combine the precision and expressiveness of deductive verification systems with the ease of use and

understandability of less complex systems. An example of such a combined approach has been developed between SDG-based systems and logic-based systems, as presented in Beckert, Bischof, et al. (2018). This thesis aims to fill the missing gap by developing a similar combined approach, this time between secure-flow type systems and logic-based systems. Type systems, being easy to use and widely familiar to programmers, serve as an excellent introduction to formal verification. They also have a substantial theoretical and practical foundation, making them a suitable basis for a combined approach. One significant difference that enhances the practicality of the verification system, as compared to previously proposed combined approaches, is the direct incorporation of type annotations into the source code. This integration facilitates the parallel development of the specification and the source code, utilizing many of the same tools such as Integrated Development Environments (IDEs) and version control. Although a comprehensive study of the practicality of such combined systems is beyond the scope of this thesis, the implementation of such a system paves the way for potential future research in this area.

Structure of the Thesis This thesis begins by demonstrating the theoretical compatibility of a type-system-based approach with a dynamic-logic-based approach to information flow control. To facilitate this, basic concepts are recapitulated, ensuring that they are presented using common terminology, which necessitates a minor adaptation of the original sources. The equivalence of the approaches is subsequently established by demonstrating that they share the same notion of noninterference, a theoretical definition synonymous with the absence of illicit information flows. Based on this equivalence, we show that a combined system maintains its soundness, as evidenced by its continued guarantee of noninterference.

Following this, a practical implementation based on the SFLow type system and the KeY theorem prover is presented, demonstrating the feasibility of such a combined approach. This implementation lays a solid foundation for further research into the practicality of these combined systems. The thesis concludes with an evaluation of the combined system and a discussion of the results.

2 Theoretical Foundations

To ensure the accuracy of the combined verification system, we will establish its theoretical correctness in Chapter 3. The proofs provided there will be grounded in the theory presented in this chapter.

We will commence with an introduction to the concept of security lattices. First introduced in Denning (1976), they are extensively utilized to precisely describe information flow properties. Next, an informal introduction to type systems and logical inference rules is provided. The notation introduced here is used to define the core language- a compact imperative language that forms the foundation for the proofs presented in this thesis, and the first of the two verification systems used - a type system, introduced in Volpano, Smith, and Irvine (2000), that ensures noninterference for well-typed programs. This is followed by an introduction to information flow verification using Java dynamic language and its verification in the KeY theorem prover. Notably, a second definition of noninterference is introduced at this point. These two noninterference guarantees, one from the Volpano-Smith type system, and a second one from the KeY theorem prover, form the basis for establishing the combined verification system.

2.1 Security Lattices

To facilitate the formal definition of information flow policies, we use the concept of*security lattices*, as defined by Denning (1976) and Bell (1973). These lattices are composed of *Labels* or *Security Classes* (SC) and a partial order on the classes, denoted as $\leq$. Every program variable is assigned a security class, represented by $\text{sec}(x)$. The information flow from a program variable x to another variable y is deemed permissible if $x \leq y$. Security classes can be combined using the associative and commutative binary operator $\oplus$, which forms the least upper bound, or by $\otimes$, which denotes the greatest lower bound. When security classes are represented as sets of labels, the operators $\oplus$ and $\otimes$ function as the union and intersection operators, respectively.

In the lattice model, secrecy might be defined by two security classes: low (L) and high (H), with $L \leq H$. Similarly, integrity could be defined by two security classes: trusted (T) and untrusted (U), with $T \leq U$.

To facilitate the illustration of these concepts, we use $x \rightarrow y$ to denote $x \leq y$, meaning that flow is allowed from x to y. Figure 2.1 displays a simple security lattice that combines both secrecy and integrity into a single lattice, using the labels L, H, T, and U. The security classes are formed from possible combinations of these labels, with a security label AB representing the security class $A \oplus B$.

Figure 2.2 illustrates a reduced approach to integrating the concepts of secrecy and integrity. In this model, we utilize only two labels: **safe** and **tainted**. These labels

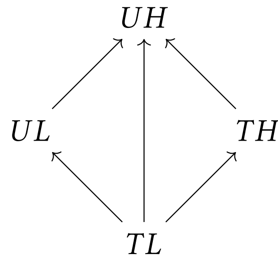

Figure 2.1: Security lattice combining secrecy and integrity

can be interpreted in two ways. For secrecy, **safe** = T and **tainted** = U, while for integrity, **safe** = L and **tainted** = H. Critically, despite its simplicity, this model does not compromise on theoretical expressiveness. It remains possible to examine a program for both integrity and secrecy by deploying two distinct lattices: one for gauging secrecy, and the other for assessing integrity. This approach, with its simplicity, is easier to manage and will be employed in Section 3.1 as the foundation for the combined verification system.

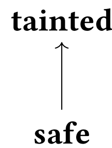

Figure 2.2: Simple security lattice combining secrecy and integrity

Explicit and Implicit Flows The most obvious way for an information flow property to be violated is through assignment between variables. Consider the assignment $t = u$, which is deemed unsound if $\mathrm{sec}(t) = T$ and $\mathrm{sec}(u) = U$ since $T \not\leq U$. This type of information transfer, where data is directly passed from one variable to another, is referred to as *explicit* information flow. However, there exist alternative mechanisms for information to flow between variables. In the subsequent example, the information flow is not explicit but *implicit*:

```
if (u == 0) {
  t = 0;
}
```

In this scenario, the trusted value in t is still reliant on the untrusted source u. Such information flows can become rather intricate and thus challenging to identify. Initial techniques for detecting such information flows were proposed by Denning (1976), and later Volpano, Smith, and Irvine (2000) who introduced a refined method based on type systems, accompanied by a proof of soundness.

Noninterference To encapsulate the concept of information flow in a mathematical framework, we employ the principle of *noninterference*. At an intuitive level, noninterference

proposes that, in terms of secrecy, the public output of a program should be exclusively determined by the public input to that program. This ensures that any private or confidential information does not influence or alter the public output.

Similarly, in the context of integrity, noninterference stipulates that the trusted output of a program should solely rely on the trusted input of that program. This prevents any untrusted or potentially harmful input from affecting the trusted output.

In essence, noninterference captures the intuitive notion of disallowing insecure information flows while being mathematically rigorous. This is achieved by disallowing any dependencies of the trusted output on untrusted input or public output on private input, thus maintaining the sanctity of the information flow within a program.

For an in-depth and formal definition of noninterference, please refer to the later sections: Section 2.5 and Section 2.4.

Attacker Model Information can be compromised through a variety of methods. For example, *side-channel* attacks infer information by observing power consumption, heat generation or other indirect aspects of program execution. These observable characteristics can inadvertently reveal sensitive information.

However, the lattice security model, which is the focus of this thesis, considers information flows that are inherent to the program itself. In this model, an attacker can only observe or modify information that has been marked as having low secrecy or low integrity according to the security labels assigned within the program. This implies that the attacker has limited yet direct access, confined solely to information that has been classified as less vital or sensitive. Information disclosed through indirect access, such as side-channel attacks, is not taken into account within the security framework that serves as the foundation for this thesis.

2.2 Type Systems

Type systems are formal systems that can be used to verify the absence of certain errors in computer programs. Essentially, a type system is composed of *inference rules* and associates types with program elements. The inference rules are employed to derive *judgments* in the form:

$$\Gamma \vdash p : \tau$$

Such a judgment indicates that under certain assumptions Γ, the expression p has a valid type τ.

Inference rules dictate what judgments can be derived. They comprise two sections divided by a horizontal line. The upper section contains a set of *premises* separated by spaces, and the lower section contains a *conclusion*. Intuitively, an inference rule states that the conclusion holds true if all the premises are true. In the context of type systems, this means that the judgment in the conclusion can be derived if all the judgments in the premise can be derived.

Take, for instance, the following example of a simple type system that deals with integers and addition:

$$\text{(ADD)}\ \frac{\Gamma \vdash e : \text{int} \qquad \Gamma \vdash e : \text{int}}{\Gamma \vdash e + e' : \text{int}}$$

This rule states that the addition of two expressions can be typed as int if both expressions are also typed as int.

To derive a type for the expression 1 + 2 using the rule above would not be sufficient. It necessitates the addition of a rule for integer literals:

$$\text{(INT)}\ \frac{}{\Gamma \vdash n : \text{int}}$$

This rule, having no premises, can always be derived. Such rules are referred to as *axioms.* With the application of this rule, we can now conclude that the expression 1 + 2 is of the type int:

$$\text{(ADD)}\ \frac{\text{(INT)}\ \frac{}{\Gamma \vdash 1 : \text{int}} \qquad \text{(INT)}\ \frac{}{\Gamma \vdash 2 : \text{int}}}{\Gamma \vdash 1 + 2 : \text{int}}$$

Now that we can derive a type for the expression 1 + 2, we categorize it as *well-typed.* The type system aids in ensuring program correctness by rejecting programs that are not well-typed, thereby preventing certain types of errors. For instance, we could expand our basic type system to accommodate floating point numbers, while rejecting expressions that combine integers and floating point numbers:

$$\text{(FLOAT)}\ \frac{}{\Gamma \vdash f : \text{float}} \qquad\qquad \text{(FADD)}\ \frac{\Gamma \vdash e : \text{float} \qquad \Gamma \vdash e : \text{float}}{\Gamma \vdash e + e' : \text{float}}$$

In this context, we interpret f as a real number that is not an integer. By applying the rules outlined above, we can determine that the expressions 1 + 2 and 1.2 + 2.2 are well-typed, whereas the expression 1 + 2.2 is not. Requiring programs to be well-typed would thereby exclude addition between floating point numbers and integers.

Definition 1. *(Synatx-directed type system) A type system is considered syntax-directed if, for every statement or expression in the language, there exists a unique inference rule that can be applied at any given stage of the derivation.*

Type Context Type judgments and inference rules are formulated with respect to a *type context* Γ. This context comprises a set of assumptions, often used to store the types of variables. The type context can be updated by inference rules, allowing the introduction of new variables, as demonstrated in the example below:

$$\text{(LETVAR)}\ \frac{\Gamma, e : \tau \qquad \Gamma[x : \tau] \vdash s : \tau'}{\Gamma \vdash \textbf{let}\ x := e\ \textbf{in}\ s : \tau'}$$

This rule allows us to derive a type τ' for a statement s under the additional assumption that the variable x holds the type τ of the expression assigned to it, denoted as $\Gamma[x : \tau]$.

To utilize this rule in type derivations, we also need to introduce a rule to consume the assumption that x holds the type tau:

$$(\textsc{var})\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

By applying these two rules, we can demonstrate that the program `let x := 1 in x + 2` is well-typed:

$$(\textsc{letvar})\ \frac{(\textsc{int})\ \dfrac{}{\Gamma \vdash 1 : \text{int}} \qquad (\textsc{add})\ \dfrac{(\textsc{int})\ \dfrac{}{\Gamma[x : \text{int}] \vdash 2 : \text{int}} \qquad (\textsc{var})\ \dfrac{\Gamma[x : \text{int}](x) = \text{int}}{\Gamma[x : \text{int}] \vdash x : \text{int}}}{\Gamma[x : \text{int}] \vdash x + 2 : \text{int}}}{\Gamma \vdash \textbf{let}\ x := 1\ \textbf{in}\ x + 2 : \text{int}}$$

2.3 The Core Language

To facilitate the theoretical exploration of the combined verification system, we will introduce the core language, as defined by Volpano, Smith, and Irvine (2000) and Volpano and Smith (1997). From this point forward, we will refer to it simply as the *core language*. The core language is an imperative language that includes integers, variables, internal memory, and procedures. One significant simplification, when compared to Java, is the absence of an object model. The process of adapting the verification system from the core language to Java is discussed in Chapter 4.

Syntax of the Core Language The syntax of the core language is delineated by the following grammar:

$$\begin{aligned}
&(\text{Phrase}) && p ::= e \mid c \\
&(\text{Expr}) && e ::= x \mid n \mid l \mid e + e' \mid e - e' \mid e == e' \mid e < e' \mid \textbf{proc}\ (\textbf{in}\ x_1, \textbf{intout}\ x_2)c \\
&(\text{Stmt}) && c ::= e := e' \mid c; c' \mid e(e_1, e_2) \mid \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' \mid \textbf{while}\ e\ \textbf{do}\ c \mid \\
& && \qquad \textbf{letproc}\ x(\textbf{in}\ x_1, \textbf{inout}\ x_2)c\ \textbf{in}\ c'
\end{aligned}$$

In this syntax, we introduce the meta-variables x, n, l, which range over (free) identifiers, integers, and *locations*, respectively. Additionally, p, e, c range over phrases, expressions, and statements (called *commands* in the original sources), respectively. These variables will act as placeholders for concrete instantiations in subsequent chapters.

Definition 2 (Program). *A phrase is considered closed if it contains no free identifiers. A closed phrase that also qualifies as a statement is referred to as a program.*

Even though there is no special syntax for blocks, they are represented as a sequence of statements. This sequence of statements is again called a statement in the core language. This language may be misleading, but it is consistent with the original sources and allows us to make general affirmations about the core language without having to distinguish between blocks and statements.

Definition 3 (Substitution). *The notation $[e/x]c$ is used to denote the statement resulting from the capture-avoiding substitution of an identifier x with an expression e in a statement c.*

Semantic of the Core Language. The semantics of the core language are defined by the derivation rules in Figure 2.3. Utilizing inference rules to describe the semantics of a programming language provides the advantage of providing a precise and robust foundation for formal proofs.

In the core language, integers are the sole data type and are used to represent boolean values, with 1 interpreted as true and 0 as false. The core language also includes simple procedures that require two parameters. The first parameter is input-only, or read-only, denoted by **in**, and the second parameter can be both read and updated, denoted by **inout**. Memory is accessed and modified through locations, which can be considered as memory addresses. These locations are interpreted with respect to a memory μ:

Definition 4 (Memory). *A memory* $\mu : \mathrm{dom}(\mu) \rightarrow \mathbb{Z}$ *is a finite function that maps from a finite domain* $\mathrm{dom}(\mu)$ *of valid memory locations to integer values.*

Definition 5 (Memory update). *We denote* $\mu[l := n]$ *as the function that maps from* $\mathrm{dom}(\mu) \cap \{l\}$ *to* $\mathbb{Z}$*, modifying the value for the location* l*:*

$$\mu[l := n](l') = \begin{cases} n & , \mathit{if}\ l' = l \\ \mu(l') & , \mathit{if}\ l' \neq l \end{cases}$$

Definition 6 (Memory deletion). *We denote* $\mu' - l$ *as the function that maps from* $\mathrm{dom}(\mu) \setminus \{l\}$ *to* $\mathbb{Z}$*, effectively removing the location* l *from the domain of* μ'*.*

Evaluation of Programs The semantics provided in Figure 2.3 allow us to derive judgments in the form of $\mu \vdash e \Rightarrow n$ for expressions and $\mu \vdash c \Rightarrow \mu'$ for closed statements. In this context, expressions are evaluated as integers, and statements are evaluated as modified memory. We can use these evaluations to define the evaluation function $\mathrm{eval}_\mu(p)$, which represents the evaluation of a closed phrase p with respect to a specific memory μ.

Definition 7 (Evaluation of expressions). *For a closed expression* e*, the evaluation is denoted as* $\mathrm{eval}_\mu(e) = n$*, where* $\mu \vdash e \Rightarrow n$*.*

Definition 8 (Evaluation of statements). *For a closed statement* c*, the evaluation is denoted as* $\mathrm{eval}_\mu(c) = \mu'$*, where* $\mu \vdash c \Rightarrow \mu'$*.*

Input and Output In this framework, the *input* of a program is considered to be the initial memory configuration μ, and the *output* of a program is the modified memory location μ' after the program has been executed.

2.4 Volpano Smith Secure Flow Type System

We will now introduce a type system for the core language, aimed at proving the absence of insecure information flows. This type of system was initially proposed in Volpano, Smith, and Irvine (2000), and later expanded to incorporate procedures in Volpano and Smith (1997). We will use notation and terminology that aligns with the other systems discussed in this thesis.

This type of system's objective is not to identify typical program errors, such as assigning floating point values to integer variables. Instead, it aims to verify that a program does not violate a specified information flow policy. This policy is represented as a security lattice $(\mathrm{SC}, \leq)$, comprising security classes SC and a partial order $\leq$ on the security classes.

Security Types There exists a one-to-one correspondence between the security classes and the *security types*. Hereafter, these security types will be denoted as τ. The partial order of the security classes translates into a subtyping relation on the security types. The security type system uses three additional parameterized types. Variables can have the type $\tau\ var$, while statements can be classified as type $\tau\ cmd$. Procedures, on the other hand, can be designated as type $\tau\ proc(\tau_1, \tau_2)$. In this context, τ signifies the overall type of the procedure, while τ_1 and τ_2 represent the types of the **in** and **inout** parameters, respectively.

In addition to τ, we introduce the meta-variables π and ρ. Here, ρ represents any type, and π denotes any non-variable type.

Type Rules The type inference rules in the Volpano-Smith type system aim to eliminate both explicit and implicit insecure information flows. To achieve this, the type system is designed with a particular interpretation of the security types: For an expression to have type τ, every value read in the expression must have a type $\tau' \leq \tau$. When using integrity types, where $T \leq U$ this implies that a trusted expression only contains trusted values, whereas an untrusted expression may contain both trusted and untrusted values. For statement and procedure types, the interpretation is that every variable updated in a statement or procedure of type τ must have a type $\tau \leq \tau'$.

A straightforward rule for valid assignments would then be:

$$\frac{\lambda; \gamma \vdash e : \tau\ var \qquad \lambda; \gamma \vdash e' : \tau}{\lambda; \gamma \vdash e := e' : \tau\ cmd}$$

This rule states that the value and the variable in an assignment must agree on their security type τ. However, assignments between variables of different security levels can still occur by coercing the type of the value using the subtyping relation. For instance, if e is of type $U\ var$ and e' is of type T, the type of e' can be coerced to U, thereby facilitating the assignment.

The syntax-directed type rules are depicted in Figure 2.4. Any rule that concludes with a hyphen, such as IDENT′, incorporates a subtyping relationship. These rules were originally derived from a set of rules that excluded subtyping relationships, in conjunction with a set of explicit subtyping rules. The advantage of the syntax-directed set of typing rules is that for every statement in the core language, only one rule is applicable. This aspect is extensively utilized in the induction proofs provided later.

The original subtyping rules that were used to derive the syntax-directed set of rules are provided in Figure 2.5 for reference. It's important to note that the types of statements are contravariant. Variables do not have any direct subtyping relationship since they can be both read and updated.

Type Context The type context consists of an *identifier typing* γ and a *location typing* λ.

Definition 9. *(Identifier typing) An identifier typing is a finite function that assigns types of the form* τ *or* τ *var to identifiers.*

Definition 10. *(Location typing) A location typing is a finite function that assigns types of the form* τ *or* τ *var to identifiers.*

The notations $\gamma[x : \rho]$ and $\lambda[l : \rho]$ represent a modified type context, where the type ρ is assigned to the variable x or the location l, respectively. The detailed definition of these notations aligns with Definition 5, and for the sake of brevity, it is not reiterated here.

Volpano Smith Noninterference We are now prepared to recapitulate the definition of noninterference as presented by Volpano and Smith.

Definition 11. *(Volpano-Smith noninterference) Let* c *be a program and* λ *a location typing with* $\mathrm{dom}(\lambda) = D$ *for its locations. The program* c *is noninterferent if and only if for all memory configurations* μ, ν, μ', ν' *such that* $eval_\mu(c) = \{\mu'\}$ *and* $eval_\nu(c) = \{\nu'\}$*, and* $\mathrm{dom}(\mu) = \mathrm{dom}(\nu) = D$*:*

$$\forall l\ \lambda(l) \leq \tau : \nu(l) = \mu(l) \implies \forall l\ \lambda(l) \leq \tau : \nu'(l) = \mu'(l) \tag{2.1}$$

When interpreting Definition 11 in terms of integrity, it implies that the state of trusted memory after a program's execution is solely determined by the state of trusted memory prior to the program's execution. In other words, the trusted memory of a program, post-execution, is not influenced by untrusted input.

Type Soundness To formalize the goal of excluding programs with insecure information flow by application of the Volpano-Smith type system a type soundness theorem is proven in Volpano, Smith, and Irvine (2000).

Theorem 1. *(Type Soundness) Let* c *be a program and* λ *a location typing for* c*.*
If $\lambda \vdash c : \rho$*, then* c *is noninterferent according to Definition 11.*

Theorem 1 states that any program for which a typing can be derived in the Volpano-Smith type system is noninterferent, thereby giving us a method to formally prove the absence of insecure information flows for a given security lattice simply by providing a valid typing for the program.

The proof of the type soundness theorem relies on structural induction and two lemmas, namely *simple security* and *confinement.* These lemmas reflect the earlier presented intuition about the significance of expression and statement types respectively. As these lemmas form the foundation for the combined verification system, they are reiterated here:

Lemma 1 (Simple Security). *If* $\lambda; \gamma \vdash e : \tau$*, then for every* l *in* e*:* $\lambda(l) \leq \tau$*, and for every* x *free in* e*,* $\gamma(x) \leq \tau$*.*

Lemma 2 (Confinement). *If* $\lambda \vdash c : \tau\ cmd$, $\mathrm{eval}_\mu(c) = \mu'$, $\mathrm{dom}(\lambda) = \mathrm{dom}(\mu)$, *and* l *is a location updated in c, then* $\lambda(l) \geq \tau$.

Here, Lemma 1 specifies that expressions of type τ only read locations with a subtype of τ, and Lemma 2 states that statements of type $\tau\ cmd$ only update locations with a supertype of τ.

2.5 Information Flow Verification in KeY

KeY is a theorem prover that utilizes higher-order dynamic logic, specifically designed for Java, known as JavaDL. It allows the formulation of a provable version of noninterference using JavaDL. This version of noninterference, akin to the Volpano-Smith version of noninterference, stipulates that two program executions, starting with the same low secrecy input, yield the same high secrecy output.

This concept of noninterference, formulated in JavaDL, can be verified by the KeY solver using *symbolic execution.* This method evaluates the given Java program's representation in JavaDL to formally prove the equivalence of the low secrecy output.

In this section, we recapitulate the fundamental definitions that shape the concept of noninterference within the KeY system as presented in Ahrendt et al. (2016). This exploration lays the groundwork for the combined system discussed in this thesis.

Please note that we will make minor modifications to the definitions presented in this chapter to fit the core language.

Observation Expressions Observation expressions are used to define the attacker model by accurately specifying the information an attacker can access. In the simplest form, an observation expression is a sequence of program variables that an attacker can read or modify. In Ahrendt et al. (2016), observation expressions can be composed of any JavaDL terms, providing a more expressive representation of the disclosed information. However, for the purpose of our work, we will be using a subset, focused on the memory locations in our program, only:

Definition 12. *(Observation expression) An observation expression R is a sequence of program locations* $\langle l_1, l_2, \ldots \rangle$.

The evaluation of closed phrases can be extended to observation expressions, done in a component-wise manner:

Definition 13. *(Evaluation of an observation expression)*

$$eval_\mu(R) = \{\langle eval_\mu(l_1), eval_\mu(l_2), \ldots \rangle\} = \{\langle \mu(l_1), \mu(l_2), \ldots \rangle\}$$

Definition 14. *(Equality of observation expressions) Two observation expressions are equal if and only if they consist of the same number of elements and their corresponding components are equal.*

To define KeY's concept of noninterference , it is necessary to establish what it means for two memory configurations to agree on an observation expression. This is defined in the same way as it is in Ahrendt et al. (2016):

Definition 15. *(Agreement of States) Let R be an observation expression. Two states μ and ν agree on R, abbreviated by* $\text{agree}(R, \mu, \nu)$, *if and only if* $eval_\mu(R) = eval_\nu(R)$.

KeY Unconditional Noninterference Using the previously outlined definitions, we can now define KeY's concept of unconditional noninterference for the core language:

Definition 16 (KeY Unconditional Noninterference)**.** *Let c be a program and R_1, R_2 be observation expression. A program c is non-interferent if and only if for all states μ, ν, μ', ν' such that $eval_\mu(c) = \{\mu'\}$ and $eval_\nu(c) = \{\nu'\}$, we have:*

$$\textit{if } \text{agree}(R_1, \mu, \nu) \textit{ then } \text{agree}(R_2, \mu', \nu')$$

In essence, this implies that if the public information in the prestate of an execution remains unchanged, then the public information in the poststate will also remain unchanged. Here, the public information in pre- and poststate is denoted by the observation expressions R_1 and R_2 respectively.

Importantly, Definition 16 can be proven using the KeY theorem prover. The mechanisms behind this proof are not within the scope of this thesis but are detailed in Section 13.5 of Ahrendt et al. (2016).

$$(\textsc{val}) \quad \frac{}{\mu \vdash n \Rightarrow n}$$

$$(\textsc{contents}) \quad \frac{l \in \mathrm{dom}(\mu)}{\mu \vdash l \Rightarrow \mu(l)}$$

$$(\textsc{add}) \quad \frac{\mu \vdash e \Rightarrow n \qquad \mu \vdash e' \Rightarrow n'}{\mu \vdash e + e' \Rightarrow n + n'}$$

$$(\textsc{sequence}) \quad \frac{\mu \vdash c \Rightarrow \mu' \qquad \mu' \vdash c' \Rightarrow \mu''}{\mu \vdash c;\ c' \Rightarrow \mu''}$$

$$(\textsc{branch}) \quad \frac{\mu \vdash e \Rightarrow 1 \qquad \mu \vdash c \Rightarrow \mu'}{\mu \vdash \textbf{if}\ e\ \textbf{then}\ c\ \ \textbf{else}\ \ c' \Rightarrow \mu'}$$

$$\frac{\mu \vdash e \Rightarrow 0 \qquad \mu \vdash c' \Rightarrow \mu'}{\mu \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' \Rightarrow \mu'}$$

$$(\textsc{call}) \quad \frac{\mu \vdash e \Rightarrow n \qquad \mu \vdash [n, l, l'/x_1, x_2, x_3]c \Rightarrow \mu'}{\mu \vdash \textbf{proc}\ (\textbf{in}\ x_1, \textbf{inout}\ x_2)\ c)(e, l, l') \Rightarrow \mu'}$$

$$(\textsc{update}) \quad \frac{\mu \vdash e \Rightarrow n \qquad l \in \mathrm{dom}(\mu)}{\mu \vdash l := e \Rightarrow \mu'[l := n]}$$

$$(\textsc{bindvar}) \quad \frac{\mu \vdash e \Rightarrow n \qquad l \notin \mathrm{dom}(\mu) \qquad \mu[l := n] \vdash [l/x]c \Rightarrow \mu'}{\mu \vdash \textbf{letvar}\ x := e\ \textbf{in}\ c \Rightarrow \mu' - l}$$

$$(\textsc{loop}) \quad \frac{\mu \vdash e \Rightarrow 0}{\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow \mu}$$

$$\frac{\mu \vdash e \Rightarrow 1 \qquad \mu \vdash c \Rightarrow \mu' \qquad \mu' \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow \mu''}{\mu \vdash \textbf{while}\ e\ \textbf{do}\ c \Rightarrow \mu''}$$

$$(\textsc{bindproc}) \quad \frac{\mu \vdash [\textbf{proc}\ (\textbf{in}\ x_1, \textbf{inout}\ x_2)\ c/x]c' \Rightarrow \mu'}{\mu \vdash \textbf{letproc}\ x(\textbf{in}\ x_1, \textbf{inout}\ x_2)\ c\ \textbf{in}\ c' \Rightarrow \mu'}$$

Figure 2.3: Semantics of the core language

$$(\textsc{ident}')\qquad \frac{\gamma(x) = \tau \qquad \tau \leq \tau'}{\lambda;\gamma \vdash x : \tau'}$$

$$(\textsc{var})\qquad \frac{\gamma(x) = \tau\ var}{\lambda;\gamma \vdash x : \tau\ var}$$

$$(\textsc{varloc})\qquad \frac{\lambda(l) = \tau\ var}{\lambda;\gamma \vdash l : \tau\ var}$$

$$(\textsc{int})\qquad \frac{}{\lambda;\gamma \vdash n : \tau}$$

$$(\textsc{r-val}')\qquad \frac{\lambda;\gamma \vdash e : \tau\ var \qquad \tau \leq \tau'}{\lambda;\gamma \vdash e : \tau'}$$

$$(\textsc{sum})\qquad \frac{\lambda;\gamma \vdash e : \tau \qquad \lambda;\gamma \vdash e' : \tau}{\lambda;\gamma \vdash e + e' : \tau}$$

$$(\textsc{compose})\qquad \frac{\lambda;\gamma \vdash c : \tau\ cmd \qquad \lambda;\gamma \vdash c' : \tau\ cmd}{\lambda;\gamma \vdash c; c' : \tau\ cmd}$$

$$(\textsc{letvar})\qquad \frac{\lambda;\gamma \vdash e : \tau \qquad \lambda;\gamma[x : \tau\ var] \vdash c : \tau'\ cmd}{\lambda;\gamma \vdash \textbf{letvar}\ x := e\ \textbf{in}\ c : \tau'\ cmd}$$

$$(\textsc{assign}')\qquad \frac{\lambda;\gamma \vdash e : \tau\ var \qquad \lambda;\gamma \vdash e' : \tau \qquad \tau' \leq \tau}{\lambda;\gamma \vdash e := e' : \tau'\ cmd}$$

$$(\textsc{if}')\qquad \frac{\lambda;\gamma \vdash e : \tau \qquad \lambda;\gamma \vdash c : \tau\ cmd \qquad \lambda;\gamma \vdash c' : \tau\ cmd \qquad \tau' \leq \tau}{\lambda;\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \tau'\ cmd}$$

$$(\textsc{while}')\qquad \frac{\lambda;\gamma \vdash e : \tau \qquad \lambda;\gamma \vdash c : \tau\ cmd \qquad \tau' \leq \tau}{\lambda;\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau'\ cmd}$$

$$(\textsc{procedure})\qquad \frac{\lambda;\gamma[x_1 : \tau_1, x_2 : \tau_2\ var] \vdash c : \tau\ cmd}{\lambda;\gamma \vdash \textbf{proc}\ (\textbf{in}\ x_1, \textbf{inout}\ x_2)\ c : \tau\ proc(\tau_1, \tau_2\ var)}$$

$$(\textsc{apply})\qquad \frac{\lambda;\gamma \vdash e : \tau\ proc(\tau_1, \tau_2\ var) \qquad \lambda;\gamma \vdash e_1 : \tau_1 \qquad \lambda;\gamma \vdash e_2 : \tau_2\ var}{\lambda;\gamma \vdash e(e_1, e_2) : \tau_1\ cmd}$$

$$(\textsc{letproc})\qquad \frac{\begin{array}{l}\lambda;\gamma \vdash \textbf{proc}\ (\textbf{in}\ x_1, \textbf{inout}\ x_2) c : \pi, \\ \lambda;\gamma \vdash [\textbf{proc}\ (\textbf{in}\ x_1, \textbf{inout}\ x_2) c / x] c' : tau\ cmd\end{array}}{\lambda;\gamma \vdash \textbf{letproc}\ x(\textbf{in}\ x_1, \textbf{inout}\ x_2) c\ \textbf{in}\ c' : \tau\ cmd}$$

Figure 2.4: Syntax-directed typing rules for the core language

$$(\textsc{cmd}^-) \qquad \frac{\vdash \tau \leq \tau'}{\tau\ cmd \leq \tau\ cmd}$$

$$(\textsc{proc}^-) \qquad \frac{\vdash \tau' \leq \tau \qquad \vdash \tau_1' \leq \tau_1}{\tau\ proc(\tau_1, \tau_2\ var) \leq \tau'\ proc(\tau_1', \tau_2\ var)}$$

$$(\textsc{subtype}) \qquad \frac{\lambda; \gamma \vdash p : \rho \qquad \vdash \rho \leq \rho'}{\lambda; \gamma \vdash p : \rho'}$$

Figure 2.5: Subtyping rules

3 Mathematical Formulation of the Translation Layer

This chapter lays the theoretical groundwork for the combined verification system. We begin by presenting the security lattice that forms the foundation of this system. Subsequently, we adapt the noninterference definitions (Definition 11, Definition 16) specifically to align with this security lattice. We then demonstrate that these adapted definitions are equivalent. This equivalence establishes the theoretical underpinning for the combined verification system.

3.1 The Basic Lattice

We proceed to define the security lattice (Figure 3.1) utilized in the translation layer. The adoption of this concrete security lattice has two main benefits. Firstly, it enables succinct theoretical discussions while maintaining practical translation. Secondly, it is in line with the security lattices frequently used in real-world systems such as Huang, Dong, and Milanova (2014).

Definition 17. *(Basic lattice) The basic lattice is a security lattice* $(SC, \leq)$ *with security classes* $SC = \{\textbf{\textit{tainted}}, \textbf{\textit{safe}}\}$ *and the partial order relation* $\textbf{\textit{safe}} \leq \textbf{\textit{tainted}}$.

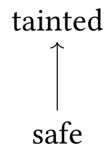

Figure 3.1: The basic lattice

This lattice is essentially the condensed security lattice discussed in Section 2.1. It facilitates the checking of a program for either secrecy or integrity. A program can be checked for both secrecy and integrity by requiring two validation passes.

- In the context of secrecy, **tainted** corresponds to H while **safe** corresponds to L.
- In the context of secrecy, **tainted** corresponds to U while **safe** corresponds to T.

3.2 Noninterference in the Context of the Basic Lattice

We proceed to adapt the noninterference definitions from Volpano-Smith and the KeY theorem prover to the basic lattice. These specialized definitions lay the groundwork for the equivalence proof.

Secure Flow Types The secure flow types utilized in the simplified definition are derived from the security classes of the basic lattice, as explicated in Section 2.4. Henceforth, τ will represent these simplified security types, specifically: $\tau \in \{\mathbf{safe}, \mathbf{tainted}\}$. The subtype relationship is also derived from the lattice, defined as: $\mathbf{safe} \leq \mathbf{tainted}$.

Volpano Smith Noninterference for the Basic Lattice Let c be a program and λ a location typing with $\mathrm{dom}(\lambda) = D$ for its locations. Furthermore μ, ν, μ', ν' are arbitrary but fixed memory locations such that $eval_\mu(c) = \{\mu'\}$ and $eval_\nu(c) = \{\nu'\}$, and $\mathrm{dom}(\mu) = \mathrm{dom}(\nu) = D$.

Theorem 2. *The following definition is equivalent to Definition 11 when applied to the basic lattice:*

$$\forall l\ \lambda(l) = safe : \nu(l) = \mu(l) \implies \forall l\ \lambda(l) = safe : \nu'(l) = \mu'(l) \tag{3.1}$$

Proof. The Volpano Smith noninterference (Definition 11), when formulated for the basic lattice, describes two cases $\tau \in \{\mathbf{safe}, \mathbf{tainted}\}$.

The case where $\tau = \mathbf{tainted}$ holds trivially true for a deterministic programming language. Since all locations are either **safe** or **tainted**, it follows that all locations $l \in D$ have a type $\leq$ **tainted**. Using this, one can restate Equation (2.1) as

$$\forall l : \nu(l) = \mu(l) \implies \forall l \leq \tau : \nu'(l) = \mu'(l) \tag{3.2}$$

□

This property affirms that the output of a program is solely dependent on its input, which is a defining characteristic of a deterministic programming language. Consequently, the core language, being deterministic, ensures noninterference for every program under consideration in the case that $\tau = \mathbf{tainted}$.

Therefore, noninterference according to Volpano Smith can be simplified to the second case only, where $\tau = \mathbf{safe}$, as stated in Equation (3.1).

KeY Unconditional Noninterference for the Basic Lattice To formulate KeY noninterference in the context of the basic lattice, we will derive observation expressions from a security class specification for a given program in the core language. The specification input is simply a location typing λ, with $\lambda(l) \in \{\mathbf{safe}, \mathbf{tainted}\}$ for all locations in $\mathrm{dom}\,\lambda = D$. This location typing assigns every location in the input of a program a security type, and by extension, an equivalent security class.

By using a location typing as the specification of security classes for the initial memory configuration, we simplify the composition of the two verification systems.

Safe Observations We define the set of all safe locations as $L_s(\lambda) = \{l \in \mathrm{dom}(\lambda) \mid \lambda(l) = \mathrm{safe}\}$. With this, we introduce the concept of a *safe observation*, denoted as S_λ:

Definition 18 (Safe observation)**.** *The safe observation S_λ for a given location typing lambda is the ordered sequence of $L_s(\lambda)$, according to an arbitrary but fixed order.*

With this definition, we can formulate KeY unconditional noninterference in the context of the basic lattice and for location typings.

Definition 19 (KeY unconditional noninterference, with respect to location typings, in the basic lattice)**.** *Let c be a program and λ a location typing with $\lambda(l) \in \{\textbf{safe}, \textbf{tainted}\}$ for all locations $l \in \mathrm{dom}(\lambda)$. The program c is non-interferent if and only if for all states μ, ν, μ', ν' such that $eval_\mu(c) = \{\mu'\}$ and $eval_\nu(c) = \{\nu'\}$, we have:*

$$\textit{if } \mathrm{agree}(S_{\lambda_c}, \mu, \nu) \textit{ then } \mathrm{agree}(S_{\lambda_c}, \mu', \nu') \tag{3.3}$$

Definition 19 is a direct instantiation of Definition 16. As such, it can be proven by the KeY theorem prover in JavaDL.

3.3 Equivalence of the Noninterference Formulations with Respect to the Basic Lattice

Both noninterference definitions now depend on an initial specification of the security labels given by a location typing λ. We can now show that their notions of noninterference are equivalent, with respect to this location typing.

Lemma 3. *For a program c and location typing λ with $\lambda(l) \in \{\textbf{safe}, \textbf{tainted}\}$, and states μ, ν with $D := \mathrm{dom}(\mu) = \mathrm{dom}(\nu) = \mathrm{dom}(\lambda)$ it holds that:*

$$\mathrm{agree}(S_\lambda, \mu, \nu) \iff \forall l \in D\ \lambda(l) = safe : \mu(l) = \nu(l) \tag{3.4}$$

Proof.

$$\begin{aligned}
\mathrm{agree}(S_\lambda, \mu, \nu) &\iff eval_\mu(S_\lambda) = eval_\nu(S_\lambda) \\
&\iff \forall l \in S_\lambda : eval_\mu(l) = eval_\nu(l) \\
&\iff \forall l \in D\ \lambda(l) = \mathrm{safe} : eval_\mu(l) = eval_\nu(l) \\
&\iff \forall l \in D\ \lambda(l) = \mathrm{safe} : \mu(l) = \nu(l) \qquad \text{according to (CONTENTS)}
\end{aligned}$$

□

Equivalence of Noninterference We are now ready to show the equivalence of the noninterference formulations concerning the core language semantics and the basic lattice information flow model. The equivalence follows almost directly from Lemma 3 and the definitions made in Section 3.2.

Theorem 3 (Equivalence of noninterference)**.** *For a program c and location typing λ with $\lambda(l) \in \{\textbf{safe}, \textbf{tainted}\}$. The noninterference definitions of Volpano-Smith and the KeY theorem solver are equivalent.*

Proof. Let c be a program and λ a location typing with $\lambda(l) \in \{\textbf{safe}, \textbf{tainted}\}$ for all locations $l \in \text{dom}(\lambda)$. Furthermore μ, ν, μ', ν' are arbitrary but fixed memory locations, such that $eval_\mu(c) = \{\mu'\}$ and $eval_\nu(c) = \{\nu'\}$.

By Theorem 2 Volpano-Smith noninterference is, under these conditions, equivalent to:

$$\forall l \; \lambda(l) = \text{safe} : \nu(l) = \mu(l) \implies \forall l \; \lambda(l) = \text{safe} : \nu'(l) = \mu'(l)$$

By substituting the agree predicate according to Lemma 3 we get:

$$\text{if agree}(S_{\lambda_c}, \mu, \nu) \text{ then agree}(S_{\lambda_c}, \mu', \nu')$$

Which is equal to Equation (3.3), thereby ending the proof. □

Due to Theorem 3 no more distinction shall be made between KeYs and Volpano-Smiths notion of noninterference. We will define $\text{secflow}_\lambda(c)$ to be a predicate that is true if and only if a program c with location typing λ is noninterferent.

Corollary 3.1. *If a valid typing ρ exists such that $\lambda \vdash c : \rho$ then the program c is noninterferent with respect to a location typing λ with $\lambda(l) \in \{\textbf{safe}, \textbf{tainted}\}$ according to Volpano-Smith and the KeY theorem solver.*

Corollary 3.1 directly follows from the equivalence of noninterference and Theorem 1 (type soundness).

3.4 Combined Verification System

Leveraging the equivalence established in the previous section, we are now able to construct a theoretical model for a combined verification system.

The security specification for both the Volpano-Smith type system and the KeY theorem prover, within the context of the basic lattice, is predicated on a location typing λ. This location typing will also serve as the security specification for the combined system. Verification is delegated either to the type system or to the KeY solver on a method-by-method basis. In practical application, this is achieved by initially running the type-system and, in the event of failure, resorting to the theorem prover. Theoretically, the sequence of these operations is irrelevant, and either verification method would suffice.

To streamline the proof, we will introduce one additional predicate:

Definition 20. *Let* $\text{taintedassign}_\lambda(c)$ *hold true if and only if, only locations of type τ = **tainted** are updated when executing c.*

The validity of this predicate can be confirmed by the KeY solver utilizing JavaDL. For further details, refer to Ahrendt et al. (2016) for more details.

Extended Type System We will formulate the combined system as an extension to the Volpano-Smith type system presented in Section 2.4. The extended system will incorporate all inference rules of the original type system, albeit the security types τ are confined to the security classes of the basic lattice $\tau \in \{\textbf{safe}, \textbf{tainted}\}$.

Furthermore, we extend the Volpano-Smith type system with an additional inference rule for procedure definitions:

$$(\textsc{KeY})\ \frac{\text{secflow}_{\lambda[l_1:\tau_1,l_2:\tau_2]}(c[l_1,l_2/x_1,x_2]) \qquad \text{taintedassign}_{\lambda[l_1:\tau_1,l_2:\tau_2]}(c[l_1,l_2/x_1,x_2])}{\lambda;\gamma \vdash \textbf{proc}\ (\textbf{in}\ x_1, \textbf{inout}\ x_2)\ c : \tau\ proc(\tau_1, \tau_2\ var)}$$

This inference rule, as opposed to relying on the type system proof of noninterference, accepts any proof of noninterference, including those delivered by the KeY theorem prover. The taintedassign predicate is crucial to support the existing type soundness proofs as outlined in Volpano and Smith (1997). Without this predicate, information flow could still occur, even if the method is noninterferent. This could occur if the method, which updates global variables with type **safe**, is executed conditionally within another method. In this inference rule, l_1 and l_2 are arbitrary locations not included in λ. They are utilized because KeY noninterference has been defined in terms of locations rather than variables to align more closely with the Volpano-Smith concept of noninterference.

Type Soundness for the Combined Volpano Smith Type System To demonstrate the efficacy of the combined verification system, we need to prove that it can indeed verify noninterference. This is achieved by establishing the following theorem:

Theorem 4. *(Extended type soundness) Let c be a program and λ a location typing where $\lambda(l) \in \{\textbf{safe}, \textbf{tainted}\}$.*

If $\lambda \vdash c : \rho$ in the combined verification system, then c is noninterferent.

The proof of Theorem 4 extends the proofs of Theorem 1, Lemma 1, and Lemma 2 as provided in Volpano, Smith, and Irvine (2000) and Volpano and Smith (1997). These proofs are structural inductions over the syntax of the core language. Since the proofs by Volpano and Smith are valid for any lattice, they also hold for the basic lattice exclusively. The only significant difference in the extended system is the inclusion of the (KeY) inference rules.

As methods cannot be part of expressions (refer to Section 2.3), the proof for Lemma 1 remains valid and does not require modification.

The inclusion of the (KeY) type rule means that the combined verification system is no longer syntax-directed, necessitating slight modifications to the original proofs.

It is worth noting that it is not surprising how slight the required modifications are. Essentially, they translate the remaining noninterference or confinement proof into a format that KeY can verify using JavaDL.

Proof: Extended Confinement. During the induction step for method definitions for confinement, we now need to distinguish between two cases. In the first case, the method was typed according to (PROCEDURE). The fact that confinement holds in this case has already been proven by Volpano and Smith.

In the second case, the method declaration was typed by (KeY). This implies that $\text{taintedassign}_\lambda(c)$ holds, directly establishing confinement. □

For type soundness, we only need to modify the part concerning procedure definitions.

Proof: Extended Type Soundness. If the type derivation, in the extended type system, concludes with the typing rule (KeY) we know that the program is noninterferent due to $\text{secflow}_\lambda(c)$. All other cases have already been proven by Volpano and Smith. □

We have now established the mathematical foundations for a verification system that integrates a type system akin to the Volpano-Smith type system for information security and a theorem prover, like KeY, capable of demonstrating the absence of information flow. The combined system is formulated as an extension of the Volpano-Smith type system. If the combined system allows for a type judgment $\lambda \vdash c : \tau$ to be made, the program is noninterferent according to Theorem 4. This noninterference complies with both the Volpano-Smith and the KeY definitions of noninterference, as indicated by Theorem 3.

In this system, the noninterference checks for methods can be performed by both the original type system and the KeY theorem solver.

4 Implementation

This chapter provides a comprehensive overview of the combined verification system's implementation. The goal of this implementation is to create a verification system that seamlessly integrates the ease of use of a pluggable type system with the expressive power of a logic-based theorem prover, such as KeY. This combined system can be employed to verify the absence of information flows in Java programs, requiring only a minimal number of user-specified annotations, thereby providing a foundation for further research into the practicality of information flow type systems.

Throughout the implementation process, special emphasis will be placed on preserving the soundness of the underlying mathematical model. Considering that Java is a more complex language than the core language used in the mathematical model, it necessitates support for a wider array of language features. Consequently, any implementation of information flow verification must accommodate fundamental Java features, including its object model and inheritance.

4.1 Architecture

The complete verification system, as illustrated in Figure 4.1, takes in Java source files annotated with security types. From this, it either generates a soundness proof, if possible or aids the user in pinpointing potential information flow violations.

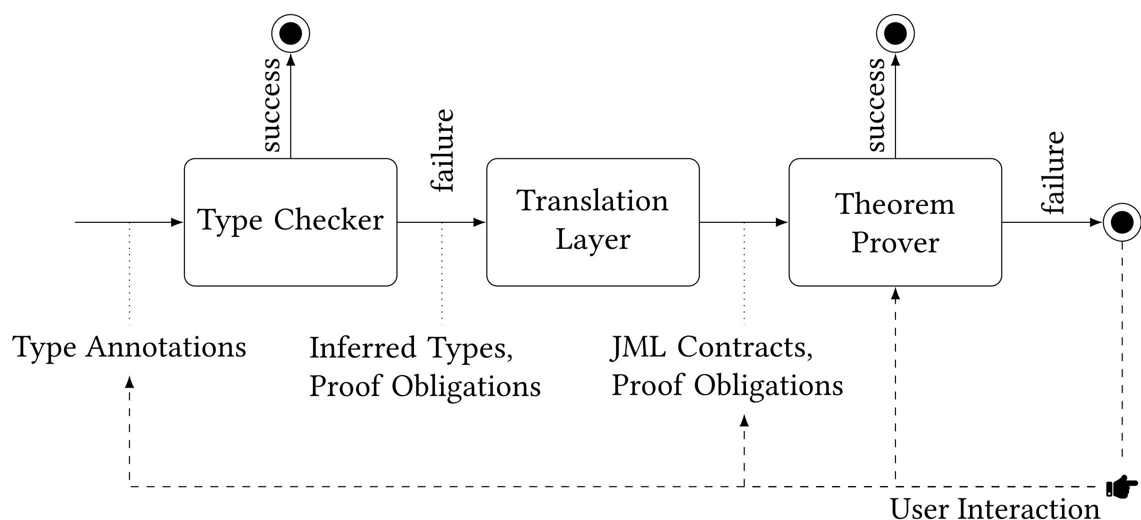

Figure 4.1: Architecture of the complete verification system

The verification system consists of three components. The first is a type checker, which is utilized to infer types that lack annotations and to verify the absence of information flow for the majority of methods. The second component is a translation layer that generates JML-annotated source code based on the types provided. The third component is a theorem prover that uses the generated source code to verify the remaining more complicated methods.

The absence of information flow is established on a method-by-method basis. For most methods, a proof is generated by the type system. However, methods with more complex information flow require additional verification by the KeY theorem prover. These methods are marked with special comments, known as proof obligations, in the generated source code. It is important to note that JML annotations must be generated for all methods, as these annotations serve as axioms for the theorem prover when analyzing method invocations.

Generating the proof can involve user interaction at various stages of the verification process. Most notably, the KeY deductive verification system is designed to be an interactive system empowering the user to prove the absence of information flow even for complex functions. This might require the user to interactively choose rules to apply to the proof or add additional JML specifications, such as loop invariants, to facilitate the proof. Moreover, both the original type annotations and the generated JML contracts can be modified by the user. This provides the user with fine-grained control at every step, including multiple options to declassify information at different levels of granularity. This topic will be discussed in more detail in Section 4.7. Lastly, proofs can also fail due to bugs present either in the source code or the type specification, necessitating user interaction to rectify the issue.

4.2 Requirements

Given that the main goal of this thesis is to provide a platform for further academic research an important requirement is that all components used should be freely available and proven to be sound. Starting from the architecture outlined in Figure 4.1 we will now derive additional requirements for the individual components of the verification system.

Requirements for the Type System The type system employed needs to meet several key requirements to ensure its effectiveness and reliability. Primarily, it must be sound, adhering to the same security notions as outlined in the model proposed by Volpano, Smith, and Irvine (2000). The soundness of the system should be validated through peer-reviewed publications, providing a solid foundation of credibility. Furthermore, the type system should incorporate implicit flows, a feature that is crucial for a comprehensive security model but often missing from practical type system implementations. Lastly, the system should be capable of performing type inference, as the goal is to develop a practical system that requires minimal annotations.

In the search for a suitable type system, two serious contenders emerge: SFlow, a secure type system initially developed for android applications and Java information flow (Jif). Both systems are proven to be sound and capable and appear in peer-reviewed academic

publications. SFlow, while not incorporating implicit flows, is based on the widely used and well-supported Checker Framework. On the other hand, Jif does not offer type inference. After careful consideration, SFlow is chosen as the preferred system. The rationale behind this decision is that extending SFlow to include implicit flows is a simpler task than incorporating type inference into Jif. This choice balances the need for a robust, reliable system with the practical considerations of system development and extension.

4.3 The Checker Framework

The Checker Framework has been utilized for a broad range of type systems and has been featured in numerous publications (developers, 2023). It is implemented as an annotation processor for Java, providing "pluggable types" (Bracha, 2004). A pluggable type systems do not influence the run-time behavior of a program and they can be used with other pluggable type systems without conflict. One consequence of these rules is the absence of custom syntax. Types are implemented by relying on standard Java annotations. This feature allows for easy incremental addition to existing projects. The framework integrates well with tooling and can be optionally enabled or disabled according to the needs of the project. Importantly, the use of the Checker Framework ensures that annotated code remains compatible with existing tool chains and build systems. The Checker Framework also offers type inference only within methods.

An example from the manual (developers, 2023) illustrates its functionality:

```
import org.checkerframework.checker.nullness.qual.*;

public class GetStarted {
    void sample() {
        @NonNull Object ref = new Object();
    }
}
```

The above code compiles without error. However, when an error is introduced:

```
@NonNull Object ref = null;
```

The framework produces the following error:

```
GetStarted.java:5: incompatible types.
found    : @Nullable <nulltype>
required: @NonNull Object
        @NonNull Object ref = null;
                              ^
1 error
```

This demonstrates the Checker Framework's robust error-reporting capabilities. It also provides good defaults, such as the subtyping checker and other checkers, to start from.

Annotation Locations The Checker Framework allows you to add annotations to the following locations:

```
@Interned String intern() { ... }                  // return value
int compareTo(@NonNull String other) { ... }       // parameter
String toString(@Tainted MyClass this) { ... }     // receiver
@NonNull List<@Interned String> messages;          // generics
@Interned String @NonNull [] messages;             // arrays
myDate = (@Initialized Date) beingConstructed;     // casts
```

It's crucial to note that annotations added to method declarations are interpreted as type annotations for the method's return value. This interpretation poses a significant restriction as it inhibits the use of method annotations to specify the statement type of a method. This issue will be addressed and resolved in Section 4.6.

In most cases, types within method bodies are inferred, eliminating the need for explicit specification.

4.4 The SFlow Type System

SFlow is a context-sensitive type system that ensures secure information flow. This system is detailed in the research paper "Type-based Taint Analysis for Java Web Applications" by Huang, Dong, and Milanova (2014) and is built upon a type inference framework for context-sensitive pluggable types, as referenced in (Milanova and Huang, 2012). SFlow and SFlowInfer are implemented as checkers inside the Checker Framework and are written in Java 6. In SFlow typing rules are implemented that closely resemble the rules of the Volpano Smith Type system. However, a significant difference is the lack of typing rules for implicit information flow, specifically (WHILE′) and (IF′). Most effective type systems forgo implicit flows since these can only be disallowed by very restrictive typing rules (Huang, Dong, and Milanova, 2014). It is important to note that noninterference cannot be guaranteed in the absence of such rules. To use SFlow as a basis for the translation layer, it must therefore be extended to include these.

Security Lattice In SFlow there are two basic type qualifiers **tainted** and **safe** with the subtyping relation **safe** ≤ **tainted**.

Additionally, a polymorphic qualifier exists **poly** which allows for context-sensitive typing. The **poly** qualifier can be interpreted either as **tainted** or as **safe** depending on the invocation context. The final subtyping hierarchy therefore becomes:

$$\textbf{safe} \leq \textbf{poly} \leq \textbf{tainted}$$

Type Inference To complement SFlow, Huang, Dong, and Milanova (2014) also introduces SFlowInfer, a worst-case cubic inference analysis for SFlow. If the inference succeeds, a *valid typing* is derived according to SFlows typing rules. A valid typing guarantees that there is no explicit flow from a source to a sink and gives types for all methods and variables not explicitly specified.

```
public static void validatePassword(
  @Tainted String pass, @Safe List<String> database) {
    @SuppressWarnings("sflow") // declassify password
    database.add(password);
}
```

Figure 4.2: Selective declassification in SFlow

Declassification in SFlow Declassification in SFlow is achieved by utilizing the checker framework's capability to suppress warnings. As shown in Figure 4.2 annotations can be added to statements to suppress checking for information flows and thereby selectively declassifying variables in certain contexts.

It is important to note that declassification using this approach is coarse. Working on a statement level does not allow us to further specify the extent of the information declassified.

4.5 JML Information Flow Contracts in KeY

Information flow contracts for KeY are defined using the Java Modeling Language (JML). In this context, two keywords are particularly relevant. The first is the `assignable` clause, which outlines the variables that can be updated during the execution of a method. This clause can be used to implement the confinement property. The second keyword is the `determines` clause, which serves as the JML equivalent of an observation expression. This clause comprises two JavaDL expressions: a precondition and a postcondition. The precondition specifies the information that can be observed, by a potential attacker, before the execution of a method and the postcondition the information that can be observed after the method's execution. It's important to emphasize that the return value of a method can only be referenced in the precondition, where it's indicated by the `\result` keyword. An example of a `determines` clause is provided in Figure 4.3.

```
//@ determines \result, l \by l;
int m();
```

Figure 4.3: Example of a `determines` clause

Declassification in KeY By using different observation expressions in the pre- and postcondition of a method information can be precisely declassified. Figure 4.4 shows an example from the Ahrendt et al. (2016) where only the sum of a list is declassified.

```
class C {
  private int[] values;

  /*@ determines \result \by
    @     (\sum int i; 0 <= i && i < values.length; values[i]);
    @*/
  int sum() {
    int s = 0;
    for (int value : values) {
      s += value;
    }
    return s;
  }
}
```

Figure 4.4: Program declassifying the sum of an array

4.6 Extending SFlow

The complete translation layer is implemented as a modification of the SFlow type system. Before we can discuss the implementation of the actual translation layer we will first need to discuss the modification made to adapt SFlow to our use case, especially the extension of SFlow to include the type checking rules for implicit information flow.

Adapting SFlow To implement the translation layer as a modification of SFlow some initial adaptations need to be made. Minor adjustments include the cleanup of the SFlow compilation and execution process. This involves the modification of build files to utilize the javac compiler from JAVA_HOME, eliminating the use of arbitrary personal paths. Additionally, the installation of versions of Java 6 and an outdated Apache Ant version is required.

One major modification involves the removal of polymorphism and the **poly** qualifier. This step is crucial to align the implementation with the underlying mathematical theory, which was formulated without polymorphism for simplicity.

Implicit Flow Typing Rules The second significant modification involves incorporating typing rules for implicit information flow. These rules are analogous to (IF′) and (WHILE′) for conditional statements and loops but also cover rules pertaining to statements that alter control flow, such as `break` and `return`. It's worth noting that the implicit flow in ternary operators has already been addressed within the Checker Framework through straightforward subtyping checks, thus eliminating the need for further attention.

Implementing these rules in the checker framework is however nontrivial. This is mainly due to the lack of support for statement types such as τ *cmd*. Furthermore, these types cannot be conveniently annotated on method declarations since annotations on methods are interpreted as the type of the return value. A more comprehensive discussion

of methods will be provided in a later section. The verification of implicit flows is therefore conducted subsequent to the execution of the base SFlow checker. By this point, the most general types for all expressions have been determined, and the absence of explicit flow, that is, the absence of assignments of tainted values to safe variables, has been proven. For notational convenience, we introduce the SFlow(λ, γ, e, τ) predicate. This predicate holds true when SFlow determines that an expression e has a most general type τ given the typing contexts λ and γ.

The next step is to demonstrate the absence of implicit flows. Intuitively this necessitates proving that no safe variables are updated if the statement being executed is conditionally dependent on a tainted expression. From a theoretical viewpoint, the goal is to find an instantiation of the typing rules for a given method. If such an instantiation is possible it proves the absence of implicit flows by Lemma 2 (Confinement).

In our implementation, called Extended-SFlow, we employ a top-down approach to check types by recursively examining the nodes of the Abstract Syntax Tree (AST). At each stage, we infer the types required by the child nodes assuming that the most general type is known for all expressions. The logic behind this process is based on the instantiation of the typing rule for a respective statement type. This close adherence to the typing rules guarantees a correct implementation. We aim to demonstrate that this instantiation is always deterministic, effectively eliminating the need for backtracking. Initially, we will focus on the implementation without considering methods, which we will discuss later. The actual implementation uses a visitor design pattern for traversing the AST, but for clarity, we will illustrate a recursive implementation.

As an entry point to our verification system, we will try to prove that a program section has type **safe** *cmd*. After introducing methods this type will depend on the statement type of the method verified. Figure 4.5 provides a schematic representation of the recursive type inference algorithm. Forthcoming examples will predominantly illustrate a single case of the statement kind switch statement.

```
public Void verifyNoImplicitFlow(Statement method) {
    return visitStatement(method, SmtType.SAFE);
}

public Void visitStatement(Statement smt, SmtType smtType) {
    switch(smt.getKind()) {
        ...
    }
}
```

Figure 4.5: Recursive type inference

Block Statements For a block to have type τ it simply means that every statement in the block needs to have type τ by the transitivity of (COMPOSE). This can be checked by recursively checking the statements in the block (Figure 4.6).

```
case BLOCK:
    for (Statement s : smt.getBlockStatements()) {
        visitStatement(s, smtType);
    }
    break;
```

Figure 4.6: Recursion for blocks

Conditional Statements If the inference algorithm needs to prove that a conditional statement has type **tainted** *cmd*, this corresponds to the following instantiation of the (IF′) typing rule:

$$(\textsc{if'})\quad \frac{\mathbf{tainted} \leq \tau \qquad \lambda; \gamma \vdash e : \tau \qquad \lambda; \gamma \vdash c : \tau\ cmd \qquad \lambda; \gamma \vdash c' : \tau\ cmd}{\lambda; \gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' : \mathbf{tainted}\ cmd}$$

Since in the basic lattice only **tainted** ≤ **tainted** it follows that τ must be **tainted**. This gives us a single deterministic way to instantiate the typing rule. The requirement for c and c' to be typed *tainted* will be recursively checked when visiting the blocks of the conditional branches.

Note that is does not matter if SFlow determines e to have type **tainted** or **safe** since we can derive $\lambda; \gamma \vdash e : \mathbf{tainted}$ even if e has most general type **safe** by subtyping:

$$(\textsc{subtype})\quad \frac{\mathbf{safe} \leq \mathbf{tainted} \qquad \dfrac{\text{SFlow}(\lambda, \gamma, e, \mathbf{safe})}{\lambda; \gamma \vdash e : \mathbf{safe}}}{\lambda; \gamma \vdash e : \mathbf{tainted}}$$

This instantiation can only fail if SFlow finds no type derivation for e. The error will be reported by SFlow and does not need to be checked again here. The intuition behind this rule is that the requirement to type the conditional as **tainted** *cmd* already specifies that no **safe** variables may be updated. Since this is already maximally restrictive there is no need to consider the type of the condition expression.

If the inference algorithm needs to prove that a conditional statement has type **safe** *cmd*, this intuitively corresponds to no additional requirements from previous inference steps. The types of the branches therefore only need to be restricted if the condition expression reads a tainted variable i.e. if the expression has type **tainted**. We will now demonstrate that this behavior follows from the typing rules.

Assuming that the conditional needs to have type **safe** *cmd* this corresponds to the following instantiation of the typing rule for conditionals:

$$(\textsc{if'})\quad \frac{\mathbf{safe} \leq \tau \qquad \lambda; \gamma \vdash e : \tau \qquad \lambda; \gamma \vdash c : \tau\ cmd \qquad \lambda; \gamma \vdash c' : \tau\ cmd}{\lambda; \gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ c\ \mathbf{else}\ c' : \mathbf{safe}\ cmd}$$

Since in the basic lattice it holds that **safe** ≤ **safe** and **safe** ≤ **tainted**, it follows that $\tau \in \{\mathbf{tainted}, \mathbf{safe}\}$ giving us two options to instantiate the proof tree and therefore two choices for the required type of the branches. If the most general type of e is *tainted*

then we need to choose $\tau = \textbf{tainted}$, since there we cannot derive $\lambda; \gamma \vdash e : \textbf{safe}$ from $\text{SFlow}(\lambda, \gamma, e, \textbf{tainted})$. When however the most general type of e is **safe**, we have two possible instantiations:

$$\text{(IF')}\quad \frac{\textbf{safe} \leq \textbf{safe} \qquad \dfrac{\text{SFlow}(\lambda, \gamma, e, \textbf{safe})}{\lambda; \gamma \vdash e : \textbf{safe}} \qquad \lambda; \gamma \vdash c : \textbf{safe}\ \mathit{cmd} \qquad \ldots}{\lambda; \gamma \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \textbf{safe}\ \mathit{cmd}}$$

or

$$\text{(IF')}\quad \frac{\textbf{safe} \leq \textbf{tainted} \qquad \dfrac{\textbf{safe} \leq \textbf{tainted} \qquad \dfrac{\text{SFlow}(\lambda, \gamma, e, \textbf{safe})}{\lambda; \gamma \vdash e : \textbf{safe}}}{\lambda; \gamma \vdash e : \textbf{tainted}} \qquad \ldots}{\lambda; \gamma \vdash \textbf{if}\ e\ \textbf{then}\ c\ \textbf{else}\ c' : \textbf{safe}\ \mathit{cmd}}$$

The first instantiation will be chosen in our implementation. This is because the second instantiation is more restrictive than necessary. The second instantiation requires the branches to be typed **tainted** *cmd* even if the condition expression does not read a **tainted** variable. By inspecting the other syntax-directed typing rules it can be observed that any rule that can be typed **safe** *cmd* can also be typed **tainted** *cmd* due to the contravariance of the statement types.

This instantiation is therefore deterministic and requires both branches to have type **safe** *cmd*. This is implemented by recursively checking the branches.

By combining the derivations described we can derive the algorithm shown in Figure 4.7 for inferring the types required by the branches of conditional statements.

```
case IF:
    if (type == SmtType.TAINTED) {
        visitStatement(smt.getThenStatement(),
            SmtType.TAINTED);
        visitStatement(smt.getElseStatement(),
            SmtType.TAINTED);
    } else { // type == SmtType.SAFE
        visitStatement(smt.getThenStatement(),
            SmtType(smt.conditional.type));
        visitStatement(smt.getElseStatement(),
            SmtType(smt.conditional.type));
    }
    break;
```

Figure 4.7: Recursion for conditionals

Conditional Loops Conditional loops have similar typing rules to conditional statements:

$$(\text{WHILE}') \quad \frac{\lambda;\gamma \vdash e : \tau \qquad \lambda;\gamma \vdash c : \tau\ cmd \qquad \tau' \leq \tau}{\lambda;\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau'\ cmd}$$

The inference step is therefore derived analogously to conditional statements. For a loop to have type **safe** the body needs to have the same type as the condition. For a loop to have type **tainted** *cmd* the body needs to have type **tainted** *cmd* and the condition can have either type, which is guaranteed by SFlow and does not need to be checked. This is again implemented recursively (Figure 4.8).

```
case WHILE_LOOP:
    if (type == SmtType.SAFE) {
        visitStatement(smt.getBody(),
            SmtType(smt.conditional.type));
    } else {
        visitStatement(smt.getBody(),
            SmtType.TAINTED);
    }
    break;
```

Figure 4.8: Recursion for loops

Base Cases Finally, we need to check assignments. Note that the subtyping checks required to show the absence of explicit information flow have already been performed by SFlow. We now need to additionally check the part of the assignment rule concerned with its statement type:

$$(\text{ASSIGN}') \quad \frac{\tau' \leq \tau \qquad \lambda;\gamma \vdash e : \tau\ var \qquad \lambda;\gamma \vdash e' : \tau}{\lambda;\gamma \vdash e := e' : \tau'\ cmd}$$

If the statement needs to be typed as $\tau' = safe\ cmd$ then τ can be both **safe** and **tainted** and the check is the same as for explicit flow. If however, the statement needs to be typed as $\tau' =$ **tainted** *cmd* then τ needs to be **tainted** and therefore the receiver and the value need to be typed **tainted** *var*. The type of the right-hand side does not need to be checked due to the covariant subtyping of expressions. However, the type of the left-hand side must be **tainted** *var*. This determination can be made by simply examining the result of the SFlow checker, as demonstrated in Figure 4.9.

Application to Java Most of the other language constructs that exist in Java but not in the core language are implemented similarly. The implementations for do while loops, for loops and enhanced for loops are analogous to the implementation of while loops combined with checking for explicit information flow since those language features. This can easily be seen since these language features can be reformulated into a program consisting only

```
case ASSIGNMENT:
    if (type == SmtType.TAINTED) {
        if (smt.getExpression().type == VarType.SAFE) {
            checker.report(Result.failure("implicit flow"));
        }
    }
    break;
```

Figure 4.9: Assignment base case

of conditionals while loops and other unconditional syntax elements. Switch statements are implemented analogously to conditionals for the same reason.

Special care needs to be taken however for statements that change control flow Figure 4.10 shows implicit flow due to early termination of a loop. This implicit flow is not forbidden under the rules presented so far.

```
@Tainted int x = 5;
@Safe int y = 0;
for (@Safe int i = 0; i < 10; i++) {
    if (x >= 5) {
        break; // continue, return, throw
    }
    y = i;
}
```

Figure 4.10: Example of implicit flow due to early termination of a loop

Code like this can be disallowed by simply assuming that every statement changing control flow is typed akin to statements writing a safe variable. For the type system this means that statements that modify control flow always have type **safe** *cmd*:

$$(\textsc{break}) \quad \vdash \texttt{break} : \textbf{safe}\ \textit{cmd}$$

This can be implemented by simply checking the expected statement type (Figure 4.11). Throwing exceptions continuing in loops and returning from methods are handled analogously.

Implicit Flow with Methods In theory, we could infer a method's most general statement type if we had access to the source code. However, this inference is not as straightforward as the previously described algorithms. The presence of cyclic calling graphs, as depicted in Figure 4.12, complicates the inference process. This is because the interred type of one method may be contingent on the type of another method. For instance, in the example provided, `foo` might call `bar` if `bar(x, y)` ⊢ **tainted** *cmd*, but the type of `bar` is dependent on the type of `foo`. Furthermore, inference becomes impossible when we lack access to

```
case BREAK:
    if (type == SmtType.TAINTED) {
        checker.report(Result.failure("implicit␣flow"));
    }
    break;
```

Figure 4.11: Control flow base case

```
public static void foo(@Safe int x, @Tainted int y) {
    if (y == 1) {
        bar(x, y);
    }
    x = 1;
}

public static void bar(@Safe int x, @Tainted int y) {
    foo(x, y);
}
```

Figure 4.12: Example of cyclic method calls

the source code. In the absence of explicit annotations, we will default to assuming the method has the type **safe** *cmd*.

By requiring the method's statement types to be specified we avoid having to infer types across method boundaries. When typing a method call we will assume that a correct type derivation exists for the annotated method type. The existence of such a derivation will be shown by requiring the type checker to prove the specified method type rather than **safe** *cmd*. See Figure 4.13 for an example implementation of this modified entry point.

```
public Void verifyNoImplicitFlow(
    Statement method, SmtType annotatedMethodType) {
    return visitStatement(method, annotatedMethodType);
}
```

Figure 4.13: Modified typecheking entry point

Method Annotations These annotated statement types cannot be the `@Safe` and `@Tainted` annotations of the SFlow Checker. Annotations on methods are interpreted as return type annotations by the Checker Framework. This interpretation can potentially result in annotation conflicts, or in the worst-case scenario, unreported errors. Figure 4.14 illustrates a scenario where the declared method type and the return type disagree. This leads to a

specification exception within the Checker Framework. Conversely, Figure 4.15 presents a case where the return type is accidentally specified as **tainted**, instead of being inferred.

```
@Tainted
public @Safe foo(@Safe) {
}
```

Figure 4.14: Example of a method declaration with conflicting annotations

```
@Tainted
public foo(@Safe) {
}
```

Figure 4.15: Example of a method with accidental return type declaration

We therefore introduce two additional annotations. `@TaintedMethod` corresponds ot the statement type **tainted** *cmd* and specifies that only variables of type **tainted** are updated during method execution. `SafeMethod` corresponds to the statement type **safe** *cmd* and specifies that variables of type **safe** may be updated during method execution.

Method Subtyping Subtype checks for arguments and return types of overridden methods are implemented by the Checker Framework. However, these subtype checks need to be extended to include the new method types.

For static and private methods, no subtyping checks are required because they are bound statically. All other methods are, by default, bound dynamically. Figure 4.16 illustrates an example of a dynamic binding causing implicit flow. Class A is an implementation without implicit flow. This is because method `foo` has type **tainted** *cmd* and therefore does not update any **safe** variables. When A is type-checked, no errors are found. Class B extends class A and overrides `foo` with type **safe** *cmd*. This is problematic because `foo` is called in `error` which expects `foo` to have type **tainted** *cmd*. When class B is type-checked, this implicit flow is not detected since the error is in class A.

Intuitively, this problem arises because class B violates the type contract of class A. Subtypes need to have method types that are at least as restrictive as the method types of their parent classes. In other words, the methods' statement types need to be contravariant under subtyping. This is implemented by simply checking the contravariance of the method types when type-checking the method declarations of child classes.

Extended-SFlow, a fusion of the original SFlow and the additional typing rules outlined in this section, now has the ability to ensure the absence of information flow in Java programs. To be more precise, if Extended-SFlow can infer types for all methods in a program, it proves that the program is noninterferent. To give these guarantees the Extended-SFlow system is conservative, potentially overestimating information flow and thus, it may be overly restrictive.

```
public static class A {
  public @Safe int safe;
  public @Tainted int tainted;

  @TaintedMethod
  public void foo() {}

  public void error() {
    if (tainted == 0) {
      foo(); // Requires foo to be @TaintedMethod
    }
  }
}

public static class B extends A {
  @SafeMethod // Forbidden by contravariant subtyping
  public void foo() {
    safe = 0;
  }
}
```

Figure 4.16: Example for dynamic dispatch causing implicit flow

4.7 Translation Layer

The translation layer is designed to perform three primary tasks: generating proof obligations by tracking type system violations at a method level, translating method types to JML contracts, and outputting source code that includes both JML contracts and proof obligations. The methods in the generated source code that include proof obligations can then be verified by the KeY solver to either prove the absence of information flow or further pinpoint the location of information flow violations.

The most efficient strategy for implementing the translation layer is to further adapt the Extended-SFlow system. This approach allows direct access to the inferred types and makes it possible to inject code to track potential information flow violations robustly and accurately.

The implementation of the translation layer is facilitated through three key components: a method information repository, a modified SFlow checker, and a source code printer. The method information repository serves as a storage for additional data related to method AST nodes. It not only records type system violations but is also used to append JML annotations to the AST. The modified SFlow checker is responsible for identifying and recording type system violations at the method level. It stores these violations along with the method types in the information repository. Lastly, the source code printer generates the source code from the AST, incorporating the supplementary data from the method information repository.

Tracking Method Type System Violations To track type system violations the SFlow checker is extended to store whenever type errors are logged during type checking. After visiting a method and finishing the type checking, if this is the case the method might have insecure information flow and the proof obligation is noted in the method information store. The user is also notified of the error by the checker framework via console output.

Generating JML Method Contracts The process of generating JML contracts for method types comprises two considerations. Firstly, we need to generate observation expressions from the method types. Secondly, for methods characterized by the **tainted** *cmd* type, we must ensure that no safe variables are updated. This safeguard is enacted by generating assignable clauses for these specific methods.

The JML comments encapsulate two primary components. For all methods, a safe observation expression is generated in the form of a JML `determines` clause. This involves the collection of all safe variables that may be accessed by the method, which includes return values, method arguments, and object fields. For Java, inherited fields must also be included. It is important to note that while return values are treated as normal variables in the underlying theory, they are managed via the JML `\result` keyword in practice. These return values can only appear in the segment of the observation expression that describes variables readable in the postcondition. Otherwise, the `determines` clause is symmetric as described in the theory.

For methods with type **tainted** *cmd*, an additional measure must be taken to ensure that no safe variables are written to. This is achieved by accumulating all **tainted** variables that can be accessed by the method and incorporating them into an `assignable` clause. This comprehensive approach to generating JML contracts provides a robust safeguard against potential information flow violations.

Generating Annotated Source Code The KeY theorem prover imports JML annotated source code, which must be complete and valid Java code. The source code is output on a file-by-file basis, and the output directory structure mirrors the structure of the input sources. To determine relative paths to be determined two command line arguments, `-Abasepath` and `-Atemppath`, are necessary. These arguments define the base directory of the input sources and the base directory of the generated sources, respectively. The code that generates the output is invoked once type checking is completed and the required information has been stored in the method information store. To integrate the output generation code into SFLow, it was necessary to upgrade the main `SourceChecker` class of the checker framework to a newer version, which includes a specific entry point method that is called for each compilation unit.

To generate source code from the AST and method store the internal OpenJDK pretty printer is extended. The pretty printer is already capable of outputting valid Java source code from a given AST. When visiting a method that has additional information stored in the method information store the extended pretty printer will output JML annotations and proof obligations in the form of Java comments.

Declassification in the Combined Verification System Information in SFlow can be declassified in two ways: either within the type system or within the theorem prover. Please refer to Section 4.4 and Section 4.5 for more details.

Declassification within the type system is accomplished by suppressing errors. As a result, no proof obligations are recorded in the method information store, no user output is produced, and no proof obligations are noted in the generated sources. In this scenario, KeY assumes that the methods contain no information flow and verifies the contracts for other methods under this assumption.

On the other hand, when declassifying information within KeY, the type system will initially raise errors and generate proof obligations. The user can then modify the generated observation expressions to declassify information precisely and continue with the proof using KeY in the usual manner.

The choice of which declassification method to use is dependent on the level of expressiveness required. Declassification within the type system is more convenient but less precise. However, with KeY, it is possible to specify precisely what information is to be declassified.

5 Evaluation

The Evaluation of the combined approach is divided into two parts. Initially, we will demonstrate that the adaptations made to SFlow accurately prohibit implicit flows. Subsequently, we will evaluate the combined verification system.

Evaluating Extended SFlow To ensure the accuracy of our implementation of the implicit flow typing rules, we utilized a test-driven development approach. During the development process, we created a comprehensive unit-test test suite that encompasses all the rules as defined by the Volpano Smith type system, as well as additional rules required to handle Java constructs not covered in the original type system. We initiated our testing with the previously implemented SFlow rules, which are designed to check for explicit flows. As we introduced new rules to the type system, we concurrently added corresponding test cases. These test cases were constructed to cover all potential instantiations of the typing rules. For instance, the rule for conditionals (IF′) necessitated the creation of test cases for the following scenarios:

- IF′ needing to be typed **safe** *cmd*
  - with a safe condition and safe branches
  - with a safe condition and tainted branches
  - with a tainted condition and safe branches
  - with a tainted condition and tainted branches
- IF′ needing to be typed **tainted** *cmd*
  - with a safe condition and safe branches
  - ...

Figure 5.1 and Figure 5.2 illustrate the test cases for a conditional that requires a **safe** *cmd* type with both a safe condition and safe branches, and a conditional that requires a **tainted** *cmd* type with a safe condition and safe branches, respectively. It's important to note that, in Figure 5.2, the type system is anticipated to log an error. This expectation is denoted by the `_fail` suffix in the test name.

A total of 34 unit tests were developed to validate the Extended-SFlow implementation. Although these tests may not encompass every possible scenario, they do thoroughly cover the type system rules and the most frequently used Java constructs. This comprehensive coverage provides a robust indication of the implementation's correctness.

```
@SafeMethod
public void testIfTypedTaintedSafeSafeSafe() {
  @Safe int safeValue = 0;
  @Safe int safeCondition = 0;

  if (safeCondition == 0) {
    safeValue = 0;
  }
}
```

Figure 5.1: Test case for IF′ needing to be typed **safe** *cmd* with a safe condition and safe branches

```
@TaintedMethod
public void testIfTypedTaintedSafeSafe_fail() {
  @Safe int safeValue = 0;
  @Safe int safeCondition = 0;

  if (safeCondition == 0) {
    safeValue = 0;
  }
}
```

Figure 5.2: Test case for IF′ needing to be typed **tainted** *cmd* with a safe condition and safe branches

```java
class TestProgram {
  public @Safe int safeValue;
  public @Tainted boolean logDeletion;

  @SafeMethod
  public void noImplicitFlow() {
    try {
      if (logDeletion) {
        safeValue = safeValue * 100; // we want to log in percent
        safelyLog();
      }
    } finally {
      safeValue = 0; // reset the value
    }
  }

  @TaintedMethod
  public void safelyLog() {}
}
```

Figure 5.3: False positive

The Combined Approach There is a variety of false positives that can be reduced by the combined approach. In Figure 5.3 a simple example is shown. The method `destruct` temporarily changes a safe variable conditioned on a tainted value. This is forbidden by the type system but does not facilitate insecure information flow. The absence of such flow can be proven with KeY in the combined verification system.

Besides testing examples where the combined system can demonstrate its ability to validate a broader range of programs, numerous unit tests, initially developed for the extended SFlow, were also employed to evaluate the combined system. The advantage of using the combined system in scenarios where invalid information flow is present lies in its capacity to identify errors that contribute to insecure information flow, even when the proof in the KeY theorem prover is incomplete.

Applying the combined system to a range of simple test cases offers insight into its efficacy. Crucially, the majority of these test cases necessitate minimal type annotations or interactions with the theorem prover.

6 Conclusion

The conclusion of this bachelor thesis marks the culmination of an extensive journey exploring the intersection between type systems and theorem provers within the realm of information flow control systems. The primary objective was to marry the user-friendliness of a type system-based approach with the expressive power of a theorem prover such as KeY. This goal was accomplished, both theoretically by developing a model for a combined approach as an extension to the Volpano-Smith type system, grounded on mathematical proofs of correctness and practically through an implementation, which was rigorously validated using diverse test cases.

During the pursuit of this objective, a thorough investigation was conducted into the disparities between information flow type systems, like SFlow, and other information flow control methodologies. The implementation of the translation layer effectively capitalized on the strengths and weaknesses of both the type-system and theorem prover approaches, offering valuable insights for future research in this domain.

The performance and capabilities of the combined SFlow-KeY system were evaluated, demonstrating its effectiveness in verifying simple but valuable information flow properties across a range of use cases. Notably, the most significant advancement was enabling the effective verification of the absence of implicit information flow using types as a specification language. Earlier efficient type systems did not verify implicit information flow to maintain practicality for large code bases (Huang, Dong, and Milanova, 2014). However, the combined approach presented in this thesis can verify implicit information flow without sacrificing practicality by delegating this verification to a theorem prover.

An intriguing aspect examined in this thesis was the impact of incorporating information flow specifications directly into the source code, as seen in SFlow and JML, compared to using external tools like JOANA. This direct integration into the source code already proved beneficial during the development of the tests used to evaluate the combined system. On one hand, it facilitated the use of version control, which proved invaluable when developing an extensive test suite. On the other hand, it enabled the quick and automatic execution of test cases. Furthermore, the author found that he frequently resorted to using the type system to generate valid JML annotations in preparation for using the KeY solver. This reliance on type inference to fill in the gaps underscored the practicality of using the type system as a foundation for the information flow specification.

A more extensive case study comparing the effectiveness and annotation overhead presents an interesting avenue for future work. The feasibility of such a case study has been established by providing the implementation of the combined approach.

In conclusion, this thesis successfully met its initial objectives and tasks, offering valuable insights and paving the way for future research. The challenges encountered along the way not only enriched the learning experience but also provided direction for future studies in this fascinating intersection of type systems and theorem provers.

Bibliography

Ahrendt, Wolfgang et al. (Jan. 2016). *Deductive Software Verification – The KeY Book: From Theory to Practice*. Vol. 10001. ISBN: 978-3-319-49811-9. DOI: 10.1007/978-3-319-49812-6.

Beckert, Bernhard, Simon Bischof, et al. (2018). "Using Theorem Provers to Increase the Precision of Dependence Analysis for Information Flow Control". In: *Formal Methods and Software Engineering*. Ed. by Jing Sun and Meng Sun. Cham: Springer International Publishing, pp. 284–300. ISBN: 978-3-030-02450-5.

Beckert, Bernhard, Daniel Bruns, et al. (2014). "Information Flow in Object-Oriented Software". In: *Logic-Based Program Synthesis and Transformation*. Ed. by Gopal Gupta and Ricardo Peña. Cham: Springer International Publishing, pp. 19–37. ISBN: 978-3-319-14125-1.

Bell, David E (1973). "Secure computer systems: Mathematical foundations and model". In: *Technical Report ESD-TR-73-278-1*.

Bracha, Gilad (2004). *Pluggable Type Systems*. URL: https://bracha.org/pluggableTypesPosition.pdf.

Denning, Dorothy E. (May 1976). "A Lattice Model of Secure Information Flow". In: *Commun. ACM* 19.5, pp. 236–243. ISSN: 0001-0782. DOI: 10.1145/360051.360056. URL: https://doi.org/10.1145/360051.360056.

developers, Checker Framework (2023). *Checker Framework Manual*. Version 3.37.0. URL: https://checkerframework.org/manual/.

Graf, Jürgen, Martin Hecker, and Martin Mohr (2013). "Using JOANA for information flow control in Java programs - A practical guide". In: *Software Engineering 2013 - Workshopband*. Bonn: Gesellschaft für Informatik e.V., pp. 123–138. ISBN: 978-3-88579-609-1.

Huang, Wei, Yao Dong, and Ana Milanova (2014). "Type-Based Taint Analysis for Java Web Applications". In: *Fundamental Approaches to Software Engineering*. Ed. by Stefania Gnesi and Arend Rensink. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 140–154. ISBN: 978-3-642-54804-8.

Kozyri, Elisavet, Stephen Chong, and Andrew C. Myers (2022). "Expressing Information Flow Properties". In: *Foundations and Trends in Privacy and Security* 3.1, pp. 1–102.

Milanova, Ana and Wei Huang (2012). "Inference and Checking of Context-Sensitive Pluggable Types". In: *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. FSE '12. Cary, North Carolina: Association for Computing Machinery. ISBN: 9781450316149. DOI: 10.1145/2393596.2393626. URL: https://doi.org/10.1145/2393596.2393626.

Myers, Andrew C. and Barbara Liskov (Oct. 2000). "Protecting Privacy Using the Decentralized Label Model". In: *ACM Trans. Softw. Eng. Methodol.* 9.4, pp. 410–442. ISSN: 1049-331X. DOI: 10.1145/363516.363526. URL: https://doi.org/10.1145/363516.363526.

Volpano, Dennis and Geoffrey Smith (1997). "A type-based approach to program security". In: *TAPSOFT '97: Theory and Practice of Software Development*. Ed. by Michel Bidoit and Max Dauchet. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 607–621. ISBN: 978-3-540-68517-3.

Volpano, Dennis, Geoffrey Smith, and Cynthia Irvine (Aug. 2000). "A Sound Type System For Secure Flow Analysis". In: *Journal of Computer Security* 4. DOI: 10.3233/JCS-1996-42-304.