Lycée Champollion

TIPE

Formal methods : implementing a secure-type system for an imperative programming language

Student :
Youssef Miled

Advisor :
Nicolas Pécheux

Abstract

This research investigates the development and integration of a type system that will enhance software security in an simple programming language. With OCaml as the basis of this study, we implement the type system to enforce the security measures.

The central interest lies in assessing how effective the type system is at preventing illegal data flow and in enhancing confidentiality and integrity restrictions. The main contributions are: the definition of a specialized type system based on Geoffrey Smith's grammar, validated through non-interference theorem and real-world testing.

In summary, this approach joins both theoretical and practical work to move us towards more secure coding practices and ultimately to developing stronger software security solutions.

Contents

1 Introduction

1.1 Motivation

The use of formal methods allows securing computer programs by ensuring that they adhere to their specifications. The goal is to leverage these formal methods to guarantee a certain level of safety, for instance the absence of unintended data transfer within the program.

The vulnerability of connected objects to cyberattacks in a connected city prompts the search for effective methods to address issues such as unauthorized system access or theft of confidential data. This involves securing the computer programs involved in these processes.

In this work, we focus on formal methods and their applications on security. A specific approach called Language-based security is examined. Intuitively, we study what happens in a program, how data is transmitted, and thus analyze and secure the information flow allowed during program execution.

1.2 Background

Formal methods play a significant role in securing computer systems. Beyond testing, they enable the assimilation of a program into a mathematical definition using certain semantics, thus employing logical deduction to establish true properties for all possible program inputs.

One can choose to reason about abstract models such as program graphs (PG). Program verification involves associating predicates with each node of the PG, which remains challenging to automate as predicates are difficult to express in machine. As for model-checking, it allows verifying that the system's abstraction model satisfies the formal model of the desired properties through logical calculation trees that formally express properties while being less expressive than the first method to achieve complete automation. Finally, the language-based security (LBS) approach ensures the security of a program in a given programming language.

Indeed, this approach is based notably on the study of the information flow allowed during program execution. Various security levels are then distinguished for each input or output involved in the program: this generally translates into sets {reliable, dubious} for integrity, {private, public} or also {unclassified, classified, secret, top secret} for confidentiality.

Controlling the information flow means determining whether a program can allow a sequence of information from a variable x, with low confidentiality level, to a variable y, with high confidentiality level, either through a direct (explicit) flow or through a conditional branch and other mechanisms (implicit flow). Information Flow Control (IFC) has already been applied to certain programming languages such as Caml (Flow Caml).

Besides, Nevin Heintze and Jon G. Riecke designed the SLam calculus (Secure Lambda Calculus), which is a typed lambda calculus containing, in addition to the usual types, security information such as reader, indirect reader (aiming to specify confidentiality), as well as creator and indirect creator (aiming to specify integrity). Using a programming language for security purposes allows statically verifying programs before their execution.

1.3 Objective

This research is centered around the exploration of a particular type system customized for a simple programming language, with a primary focus on elucidating its contributions to reinforcing security properties within software systems. In addition to examining the type system, this study also involves a detailed analysis of the target programming language, essential for parsing purposes. Furthermore, the ultimate aim is to translate theoretical insights into practical implementation by realizing the identified approach within the OCaml programming environment.

This approach combines theoretical research, hands-on experimentation, and practical deployment to generate actionable insights for secure coding practices and develop robust software security solutions.

1.4 Contributions

The main contributions of this paper are:

- We develop a certain type system for secure information flow by extending the use of functions and typing rules to Geoffrey Smith's grammar and type system.

- We establish the validity of the type system by proving the non-interference theorem for the added functionalities.
- We make the case for implementing a representation of trees for secure type proofs.
- We test the type system and its representation on a real-world case.

2 Modelisation

2.1 Language

The language used in this work is a simple imperative language that contains the most important aspects a programming language must have, i.e assignments, conditional branches, while loops and binary operations between expressions.

2.1.1 Definition

The language that we work on is defined by the following grammar:

$$\begin{aligned}
e &::= n \mid x \mid e_1 \odot e_2 \mid f(e) \\
c &::= x := e \,|\, \textbf{skip} \mid \;|\, \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \,| \\
&\quad\ \textbf{while } e \textbf{ do } c \,|\, c_1;\ c_2 \mid f(e) \mid \textbf{return } e \mid \\
&\quad\ f ::= \ fun\ (\lambda.x)\ \{c\} \\
\mathrm{p} &::= e \mid c
\end{aligned}$$

The grammar we present includes expressions (e), commands (c), and phrases (p). It outlines a straightforward language that contains basic expressions, while loops, conditional statements (if-else), and the ability to sequence commands. This framework provides the essential tools needed to build simple programs, allowing for both basic calculations and control flow structures.
Each category (expression, command, phrase) is essential to the language's structure, serving distinct roles within the language's syntax and semantics.

Expressions:
Expressions (e) constitute the fundamental computational constructs of the language. They are defined recursively as follows:
Numerals (n): Represent integer constants.
Variables (x): Symbols that store values.
Binary Operations ($e1 \odot e2$): Combinatorial constructs enabling the composition of expressions. $\odot \in \{+, -, \times, \div, =, \neq, <, \geq, \wedge, \vee\}$.

Commands:
Commands (c) embody the executable actions and control structures within the language:
Assignment ($x := e$): Modifies the value stored in the variable x to the value evaluated from expression e.
No-operation (**skip**): Represents a command that performs no action, useful for expressing semantic no-ops in complex command sequences. We will see later how this command is crucial so that the type system offers secure information flow.
Conditional execution (**if** e **then** c_1 **else** c_2): Directs the flow of execution based on the evaluation of expression e. If e evaluates to a true value, command $c1$ is executed; otherwise, c_2 is executed. We will see later how it is important to have the else branch in certain contexts.
Iteration (**while** e **do** c): Implements looping behavior where command c is repeatedly executed as long as the expression e evaluates to a true value.
Sequence ($c_1; c_2$): Denotes the sequential execution of two commands, c_1 followed by c_2.
Functions ($fun(\lambda.x)\{c\}$) are named blocks of code designed to perform specific tasks. They promote code reusability and modularity by allowing you to write a code block once and call it multiple times throughout your program. Functions typically take arguments (inputs) of specific types and return a value (output) of another type. We reserve the notation $\lambda.x$ for an internal variable within a function.

Phrases:
Phrases (p) are the top-level syntactic units of the language, which can be either an expression(or a command. This dual nature allows the language to be versatile in contexts such as interactive shells or script commands, where expressions can be evaluated for their effects or values, and commands can alter states or control flows. A code is therefore defined, in this system, by a certain phrase p.

An example of a code in this grammar can be a phrase defined by :

$$a := 5;\ b := 2;\ \textbf{while}\ a\ \textbf{do}\ a := a - b$$

The design of this grammar supports the development of a structured programming language capable of expressing a wide range of computational logic through a compact and coherent syntactic framework. This approach not only simplifies parsing and interpretation but also helps in clearly defining computational semantics, which is essential for understanding the theoretical foundations and practical applications of the language.

2.2 Semantics

In this language, programs are executed based on a specific memory denoted as μ, which consists of a set of pairs $\{(e_i, n_i)\}$, where e_i represents expressions and n_i represents integers.
The semantics of commands are defined through a sequential transition relation denoted as $\longrightarrow$ on configurations.

Definition 2.1. *A configuration C is defined by a couple (c, μ) or μ where c is a command and μ is a memory.*

We extend Geoffrey Smith's semantics to include functions here ((FUNC) semantic).

(UPDATE)	$\dfrac{x\ \in\ dom(\mu)}{(x\ :=\ e,\ \mu)\longrightarrow\mu[x\ :=\ \mu(e)]}$
(NO-OP)	$(\textbf{skip},\ \mu) \longrightarrow \mu$
(BRANCH)	$\dfrac{\mu(e)\ \neq\ 0}{(\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2,\ \mu)\longrightarrow(c_1,\ \mu)}$
	$\dfrac{\mu(e)\ =\ 0}{(\textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2,\ \mu)\longrightarrow(c_2,\ \mu)}$
(LOOP)	$\dfrac{\mu(e)\ =\ 0}{(\textbf{while}\ e\ \textbf{do}\ c,\ \mu)\longrightarrow\mu}$
	$\dfrac{\mu(e)\ \neq\ 0}{(\textbf{while}\ e\ \textbf{do}\ c,\ \mu)\longrightarrow(c;\ \textbf{while}\ e\ \textbf{do}\ c,\ \mu)}$
(SEQUENCE)	$\dfrac{(c_1,\ \mu)\longrightarrow\mu'}{(c_1;c_2,\ \mu)\longrightarrow(c_2,\ \mu')}$
	$\dfrac{(c_1,\ \mu)\longrightarrow(c_1',\mu')}{(c_1;c_2,\ \mu)\longrightarrow(c_1';c_2,\ \mu')}$
(RET)	$(\textbf{return}\ e,\ \mu) \longrightarrow \mu$
(FUNC)	$(f\ ::= fun\ (\lambda.x)\ \{c\},\ \mu) \to \mu$
	$\dfrac{\begin{array}{c} dom(\mu'') = dom(\mu') \setminus \{\lambda.x\} \\ \mu''(e) = \mu'(\lambda.x) \\ (c,\ \mu\ \cup\ \{(\lambda.x,\ \mu(e))\}) \longrightarrow^* \mu' \end{array}}{(f(e)\{c\},\ \mu)\longrightarrow\mu''}$

Table 1: Extended semantics

Here, we assume for simplicity that function calls are evaluated atomically. To assure this we can use a mutex.
We explain the semantic for functions as follows: in order for a function call $f(e)\{c\}$ to modify a memory μ into μ'', executing the command c with the memory μ extended with the internal variable $\lambda.x$ should result in a memory μ' and we also require $\mu''(e) = \mu'(\lambda.x)$ since $e \in dom(\mu'')$ which is the memory oustide the function. Naturally, the final memory μ'' does not include the internal variable $\lambda.x$.

3 Type system

3.1 Overview

We revisit Georges Smith type system for our language and add other features such as functions. This revisited system will be proven to verify information flow security (non-interference property, see 1). The types used in this system are:

$$\begin{aligned}\tau &::= L \mid H \\ \rho &::= \tau \mid \tau\ cmd\ n \mid \tau_1\ cmd\ \tau_2 \mid \tau_1 \rightarrow \tau_2\end{aligned}$$

Indeed, there are two main classes un in Geoffrey Smith's (GS) type system: L (for low) which characterizes a public information, that does not need to be protected, and H (for high) which is made for private information that should be secured. Besides, commands c which are typed $\tau_1\ cmd\ \tau_2$ assign only to variables of type τ_1 and whose execution time depends on variables of type τ_2. The same thing goes for commands of type $\tau\ cmd\ n$, which execute in n steps.
We introduce a new type constructor for functions, that GS's type system does not include. A function of type $\tau_1 \rightarrow \tau_2$) is a function that has an argument of type τ_1 and return a value of type τ_2.

Therefore, programs are constructed following the revisited rules from Geoffrey Smith typing rules (TABLE ??). We added to these some rules involving functions.

(R-VAL) $$\frac{\gamma(x) = \tau\ var}{\gamma \vdash x : \tau}$$

(INT) $$\gamma \vdash n : L$$

(BINOP) $$\frac{\gamma \vdash e_1 : \tau_1,\ \gamma \vdash e_2 : \tau_2}{\gamma \vdash e_1 \odot e_2 : \tau}$$

(ASSIGN) $$\frac{\gamma(x) = e : \tau\ var,\ \gamma \vdash e : \tau}{\gamma \vdash x := e : \tau\ \mathrm{cmd}\ 1}$$

(SKIP) $$\gamma \vdash \textbf{skip} : H\ cmd\ 1$$

(IF) $$\frac{\begin{array}{c}\gamma \vdash e : \tau \\ \gamma \vdash c_1 : \tau\ cmd\ n \\ \gamma \vdash c_2 : \tau\ cmd\ n\end{array}}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau\ cmd\ n{+}1}$$

$$\frac{\begin{array}{c}\gamma \vdash e : L \\ \gamma \vdash c_1 : \tau_1\ cmd\ \tau_1 \\ \gamma \vdash c_2 : \tau_1\ cmd\ \tau_2\end{array}}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : \tau_1\ cmd\ \tau_2}$$

$$\frac{\begin{array}{c}\gamma \vdash e : H \\ \gamma \vdash c_1 : H\ cmd\ H \\ \gamma \vdash c_2 : H\ cmd\ H\end{array}}{\gamma \vdash \textbf{if}\ e\ \textbf{then}\ c_1\ \textbf{else}\ c_2 : H\ cmd\ H}$$

$$(\text{WHILE})\quad \frac{\begin{array}{c}\gamma \vdash e : L\\ \tau_2 \subseteq \tau_1\\ \gamma \vdash c : \tau_1\ cmd\ \tau_2\end{array}}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau_1\ cmd\ \tau_2}$$

$$\frac{\begin{array}{c}\gamma \vdash e : H\\ \gamma \vdash c : H\ cmd\ H\end{array}}{\gamma \vdash \textbf{while}\ e\ \textbf{do}\ c : \tau_1\ cmd\ \tau_2}$$

$$(\text{COMPOSE})\quad \frac{\begin{array}{c}\gamma \vdash c_1 : \tau_1\ cmd\ L\\ \gamma \vdash c_2 : \tau_1\ cmd\ \tau_2\end{array}}{\gamma \vdash c_1;\ c_2\text{:}\ \tau_1\ cmd\ \tau_2}$$

$$\frac{\begin{array}{c}\gamma \vdash c_1 : \tau\ cmd\ H\\ \gamma \vdash c_2 : H\ cmd\ H\end{array}}{\gamma \vdash c_1;\ c_2\text{:}\ \tau\ cmd\ H}$$

Table 2: Geoffrey Smith typing rules

We add a rule for the **return** command.

$$(\text{RET})\quad \frac{\gamma \vdash e\ :\ \tau_1}{\gamma \vdash \textbf{return}\ e\text{:}\ \tau_1\ \ cmd\ 1}$$

Considering a function f defined by $f ::= fun(\lambda.x)\{c;\ \text{return}\ e\}$:

$$(\text{FUNC})\quad \frac{\begin{array}{c}\gamma,\ \lambda.x : \tau_1\ \vdash\ y : \tau_2\\ \gamma,\ \lambda.x : \tau_1\ \vdash\ c : \tau_1\ cmd\ n\end{array}}{\gamma \vdash f\ \ (\lambda.x)\{c;\ \text{return}\ y\}\ :\ \tau_1 \rightarrow \tau_2}$$

If f is defined by $f ::= fun(\lambda.x)\{c\}$:

$$(\text{VOID-FUNC})\quad \frac{\gamma,\ \lambda.x : \tau_1\ \vdash\ c : \tau_1\ cmd\ n}{\gamma \vdash f\ \ (\lambda.x)\{c\}\ :\ \tau_1 \rightarrow H}$$

In both cases, the rule to call a function is as follows :

$$(\text{CALL})\quad \frac{\begin{array}{c}\gamma \vdash e\ :\ \tau_1\\ \gamma \vdash f(\lambda.x)\{c\}\ : \tau_1 \rightarrow \tau_2\end{array}}{\gamma \vdash f(e)\{c\}\ :\ \tau_2}$$

We presume that upon calling a function f with a given expression e, the command $\lambda.x := e$ is automatically executed.

For a definition of a function, nothing should impact the code and the information flow, therefore we consider the following rule:

$$(\text{FUNC-DEF})\quad \gamma\ \vdash\ f ::= fun\ (\lambda.x)\{c\} : L\ cmd\ H$$

Indeed, for any command type $\tau_1\ cmd\ \tau_2$, we have $L\ cmd\ H\ \subseteq \tau_1\ cmd\ \tau_2$ (see subtyping rules 3).

Note that γ, the typing environment, contains the types of the variables that are manipulated for a certain code.

These new typing rules involve two categories of functions :

- functions that return an expression. For instance,

  $$f(\lambda.x)\{\lambda.y := x + 1;\ \text{return}\ \lambda.y\}$$

  is in this category.

- functions that do not return anything and simply apply certain commands. For instance,

  $$f(\lambda.x)\{\lambda.x := \lambda.x + 1\}$$

  is a void function.

In order to use these rules in formal proofs, we need to define subtyping rules which define an order for the types that we defined. For this matter, we extend Geoffrey Smith's subtyping rules by introducing rules that establish an order among the new added types.

$$(\textsc{base}) \qquad L \subseteq H$$

$$(\textsc{cmd}) \qquad \frac{\tau_1 \ \subseteq \ \tau_1', \quad \tau_2' \ \subseteq \ \tau_2}{\tau_1 \ cmd \ \tau_2 \ \subseteq \ \tau_1' \ cmd \ \tau_2'}$$

$$\frac{\tau' \ \subseteq \ \tau}{\tau \ cmd \ n \ \subseteq \ \tau' \ cmd \ n}$$

$$\tau \ cmd \ n \subseteq \tau \ cmd \ L$$

$$(\textsc{reflex}) \qquad \rho \ \subseteq \ \rho$$

$$(\textsc{trans}) \qquad \frac{\rho_1 \ \subseteq \ \rho_2, \quad \rho_2 \ \subseteq \ \rho_3}{\rho_1 \ \subseteq \ \rho_3}$$

$$(\textsc{subsump}) \qquad \frac{\gamma \vdash p \ : \ \rho_1, \quad \rho_1 \ \subseteq \ \rho_2}{\gamma \vdash p \ : \ \rho_2}$$

Table 3: Geoffrey Smith subtyping rules

Here are the additional subtyping rules:

$$(\textsc{func}) \qquad \tau_1 \to \tau_2 \ \subseteq \ \tau_1 \ cmd \ 1$$

$$\frac{\tau_1' \ \subseteq \ \tau_1, \ \tau_2 \ \subseteq \ \tau_2'}{\tau_1 {\to} \tau_2 \ \subseteq \ \tau_1' {\to} \tau_2'}$$

Table 4: New subtyping rules

The last rule ((FUN)) can be justified as follows: A function that takes as an argument $x : \tau_1$ can also be viewed as a general case of a function that takes an argument of type $\tau_1' \subseteq \tau_1$ and returns an expression of type τ_2' since it is designed to return an expression of type $\tau_2 \subseteq \tau_2'$.

3.2 Properties

Before delving into the property, let's recall some definitions.

Definition 3.1. *A command c is well typed with respect to a typing environment γ iff $\exists \ \rho, \ \gamma \ \vdash \ c \ : \ \rho$.*

Definition 3.2. *2 memories μ and ν are equivalent with respect to γ iff μ and ν have the same domain and agree on all L variables.*

Definition 3.3. *2 commands c and d are equivalent with respect to γ iff : c and d are well typed with respect to γ and one of the following conditions is verified :*

- $c = d$
- $c : H \ cmd \ \tau$ *and* $d : H \ cmd \ \tau$
- $c = c_1; c_2; ...; c_k$ *and* $d = d_1; c_2; ...; c_k$ *with* c_1 *and* $d_1 \ : \ H \ cmd \ n$

This relation is denoted by $\sim_\gamma$.

Definition 3.4. *2 Configurations C and D are equivalent with respect to γ iff*

- *$C = (c, \mu)$, $D = (d, \nu)$ with $c \sim_\gamma d$ and $\mu \sim_\gamma \nu$*
- *$C = (c, \mu)$, $D = \nu$ with $c : H\ cmd\ \tau$ and $\mu \sim_\gamma \nu$.*
- *$C = \mu$ and $D = \nu$ with $\mu \sim_\gamma \nu$.*

Lemma 1. *Given two memories μ and ν and a typing environment γ, if $\mu \sim_\gamma \nu$ then*

$$\mu \cup \{(\lambda.x, \mu(e))\} \sim_{\gamma'} \nu \cup \{(\lambda.x, \nu(e))\}$$

where $\gamma' = \gamma \cup \{(\lambda.x, \tau\ var)\}$.

Proof. Let's consider two memories μ and ν and γ a typing environment. We suppose $\mu \sim_\gamma \nu$. Let $\gamma' = \gamma \cup \{(\lambda.x, \tau\ var)\}$. To prove that $\mu \sim_{\gamma'} \nu$ we should prove that they agree on all L variables in γ'. Let $(y, L\ var) \in \gamma'$. If $(y, L\ var) \in \gamma$ then $\mu(y) = \nu(y)$ because $\mu \sim_\gamma \nu$. Otherwise, $(y, L\ var) = (\lambda.x, \tau\ var)$ so $\tau = L$. Thus, $\gamma\ \vdash\ e\ :\ L$ according to the rule ASSIGN in **??**. So $\mu(e) = \nu(e)$ and $\mu(y) = \nu(y)$ because we have $\lambda.x := e$ when calling $f(e)$. In both cases $\mu(y) = \nu(y)$. Therefore, $\mu \sim_{\gamma'} \nu$. □

Lemma 2. *Given two memories μ and ν and a typing environment γ, if $\mu \sim_\gamma \nu$ and $\gamma' \subseteq \gamma$ then $\mu \sim_{\gamma'} \nu$.*

Proof. If μ and ν agree on all L variables in γ, they also agree on all L variables in γ' since $\gamma' \subseteq \gamma$. □

The type system that we developed here, based on Geoffrey Smith's and some added functionalities verifies the property of non-interference.

Theorem 1. *(non-interference).*

$$\forall \gamma \in F(I, \{L\ var, H\ var\}), \forall c \in commands, \forall \mu, \nu \in F(I, \mathbb{N}),$$
$$(\exists \rho, \gamma \vdash c : \rho\ \wedge\ \mu \sim_\gamma \nu\ \wedge\ (c, \mu) \longrightarrow C'\ \wedge\ (c, \nu) \longrightarrow D')\ \Longrightarrow\ (C' \sim_\gamma D')$$

Proof. Let c be a well typed command with respect to γ, i.e $\exists \rho,\ \gamma \vdash c : \rho$. Suppose also $\mu \sim_\gamma \nu$ and $(c, \mu) \longrightarrow C'$ and $(c, \nu) \longrightarrow D'$. For the general case, we consider $c = c_1; c_2; ...; c_k$ and we consider in turn each of the possible forms of c_1. The theorem is proven in Geoffrey Smith's paper if c_1 is of one of these forms :

- $x\ :=\ e$
- **skip**
- **if** e' **then** $c_{1,1}$ **else** $c_{1,2}$
- **while**e **do** $c_{1,1}$

If c_1 is of the form $f ::= fun\ (\lambda.x)\ \{d\}$, i.e c defines a function, then according to the first semantic rule (FUNC), we can deduce that $C' = (c_2; ...; c_k, \mu)$ and $D' = (c_2; ...; c_k, \nu)$ and $C' \sim_\gamma D'$ holds true because of the equivalence between μ and ν.

We now consider c_1 as $f(e)$ for some expression e and some function $f(\lambda.x)\{d\}$. We suppose that d is a command without any void function call, therefore $d = d_1; d_2; ...; d_k$ with d_i defined as **if** e' **then** $d_{i,1}$ **else** $d_{i,2}$ **while** e do d_{i1}, or **skip**. We can always reduce to this case since when demonstrating the validity of this specific case, we can then build our argument for all other possible cases iteratively.

Since $(c, \mu) \longrightarrow C'$, by the semantic (SEQUENCE) we have $C' = (c_2; ...; c_k, \mu')$ and similarly, D' is $(c_2; ...; c_k, \nu')$, where μ' (respectively ν') is the modified memory μ (respectively ν) after applying the void function f according to the following semantic:

$$\frac{\begin{array}{c} dom(\mu') = dom(\mu'') \setminus \{\lambda.x\} \\ \mu'(e) = \mu''(\lambda.x) \\ (d,\ \mu\ \cup\ \{(\lambda.x,\ \mu(e))\}) \longrightarrow^* \mu'' \end{array}}{(f(e)\{d\},\ \mu) \longrightarrow \mu'}$$

Thus, it follows that $(d,\ \mu \cup \{(\lambda.x,\ \mu(e))\}) \longrightarrow^* \mu''$, which means that executing d with the memory $\mu \cup \{(\lambda.x,\ \mu(e))\}$ leads to the memory μ''. Similarly, $(d, \nu \cup \{(\lambda.x,\ \nu(e))\}) \longrightarrow^* \nu''$. Besides, note that the typing environment corresponding to $\mu \cup \{(\lambda.x,\ \mu(e))\}$ and $\nu \cup \{(\lambda.x,\ \nu(e))\}$ is $\alpha = \gamma \cup \{(\lambda.x, \tau\ var)\}$. Since by assumption $\mu \sim_\gamma \nu$, LEMMA 1 gives us $\mu \cup \{(\lambda.x, \mu(e))\} \sim_\alpha \nu \cup \{(\lambda.x, \nu(e))\}$. Moreover, as the theorem is proven for well-typed commands that do not include functions, we can assert that $\mu'' \sim_\alpha \nu''$. Additionally, since we have $dom(\mu') = dom(\mu'') \setminus \{\lambda.x\}$ and $\mu'(e) = \mu''(\lambda.x)$, LEMMA 2 implies $\mu' \sim_\gamma \nu'$. Consequently, we conclude that $C' \sim_\gamma D'$. □

4 Implementation

The implementation begins by defining data types to represent the syntax of expressions, commands, and other language constructs. These data types capture the structure of the language and provide a foundation for type inference.

```
type binop =
  Plus | Minus | Mult | Div | Eq | Neq | Lt | Gt | Leq | Geq | And | Or

type exp =
  | Int of int
  | Var of string
  | Binop of binop * exp * exp
  | FuncCall of string * exp

type command =
  | Assign of string * exp
  | Skip
  | If of exp * command * command
  | While of exp * command
  | Seq of command * command
  | Return of exp
  | FuncDef of string * string * command

type phrase =
  | Exp of exp
  | Command of command

type data_types = L | H

type phrase_type =
  | T of data_types
  | Var of data_types
  | Func of data_types * data_types
  | Cmd of data_types * data_types
  | Ncmd of data_types * int

type var_type = string * phrase_type
```

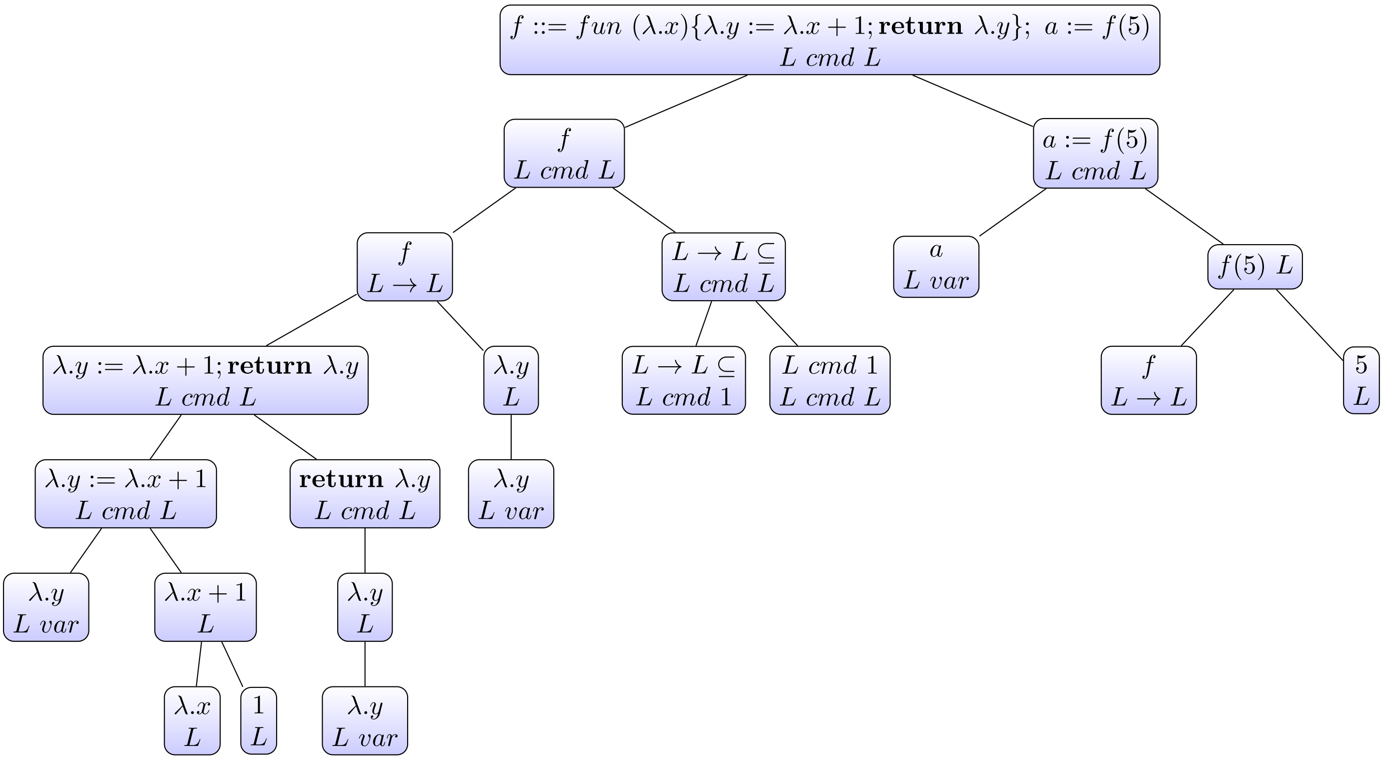
Besides, we represent a formal proof tree using an OCaml datatype. Each node in the tree corresponds to a step in the proof, and the tree structure captures the logical flow of the proof process. Below is the OCaml datatype definition for the tree:

```
type tree =
  | Empty
  | VarDeriv of string * phrase_type
  | ConstDeriv of int
  | BinopDeriv of phrase * phrase_type * tree * tree
  | SkipDeriv of phrase_type
  | AssignDeriv of string * phrase * phrase_type * tree * tree
  | IfDeriv of phrase * phrase_type * tree * tree * tree
  | WhileDeriv of phrase * phrase_type * tree * tree
  | SeqDeriv of phrase * phrase_type * tree * tree
  | ReturnDeriv of exp * phrase_type * tree
  | FuncDefDeriv of string * string * command * phrase_type * tree
  | FuncNonVoidCallDeriv of string * exp * phrase_type * tree
  | FuncVoidCallDeriv of string * string * command * phrase_type
  (* For the subtyping rules : *)
  | SubDeriv of phrase * phrase_type * phrase_type * tree * tree
| SubRules of phrase_type * phrase_type * tree * tree
```

This datatype captures various derivations and subderivations within the proof tree, allowing us to construct a structured representation of the proof process. To understand more this dataset, consider the following code:

$$f ::= fun\ (\lambda.x)\{\lambda.y := \lambda.x + 1; \textbf{return}\ \lambda.y\};\ a := f(5)$$

Here is an example of a proof tree that demonstrates that this code is secure and therefore there cannot be any leak of private information into public variables:



$f ::= fun\ (\lambda.x)\{\lambda.y := \lambda.x + 1; \textbf{return}\ \lambda.y\};\ a := f(5)$ $L\ cmd\ L$
f $L\ cmd\ L$
$a := f(5)$ $L\ cmd\ L$
f $L \to L$
$L \to L \subseteq$ $L\ cmd\ L$
a $L\ var$
$f(5)\ L$
$\lambda.y := \lambda.x + 1; \textbf{return}\ \lambda.y$ $L\ cmd\ L$
$\lambda.y$ L
$L \to L \subseteq$ $L\ cmd\ 1$
$L\ cmd\ 1$ $L\ cmd\ L$
f $L \to L$
5 L
$\lambda.y := \lambda.x + 1$ $L\ cmd\ L$
$\textbf{return}\ \lambda.y$ $L\ cmd\ L$
$\lambda.y$ $L\ var$
$\lambda.y$ $L\ var$
$\lambda.x + 1$ L
$\lambda.y$ L
$\lambda.x$ L
1 L
$\lambda.y$ $L\ var$

Figure 1: A fromal proof tree

This tree is represented in our implementation by the example_proof_tree :

```
let assign_tree =
  SubDeriv(
    Command (Assign("a", FuncCall("f", Int 5))),
    Cmd(L, L),
    Ncmd(L, 1),
    AssignDeriv(
      "a",
      Exp (FuncCall("f", Int 5)),
      Ncmd(L, 1),
      VarDeriv("a", T L),
      FuncNonVoidCallDeriv("f", Int 5, T L, ConstDeriv(5))
    ),
    SubRules(
      Ncmd(L, 1),
      Cmd(L, L),
      Empty,
      Empty
    )
  )
```

```
let function_command_tree =
  AssignDeriv(
    "y",
    Exp (Binop(Plus, Var "x", Int 1)),
    Ncmd(L, 1),
    VarDeriv("y", T L),
    BinopDeriv(
      Exp( Binop(Plus, Var "x", Int 1)),
      T L,
      VarDeriv("x", T L),
      ConstDeriv(1)
    )
  )


let func_tree =
  SubDeriv(
    Command (FuncDef("f", "x", Seq( Assign("y", Binop(Plus, Var "x", Int 1)), (Return (V
    Cmd(L, L),
    Func(L, L),
    FuncDefDeriv(
      "f",
      "x",
      Seq( Assign("y", Binop(Plus, Var "x", Int 1)), (Return (Var "y")) ),
      Func(L, L),
      function_command_tree
    ),
    SubRules(
      Func(L, L),
      Cmd(L, L),
      SubRules(
        Func(L, L),
        Ncmd(L, 1),
        Empty,
        Empty
      ),
      SubRules(
        Ncmd(L, 1),
        Cmd(L, L),
        Empty,
        Empty
      )
    )
  )


let example_proof_tree =
  SeqDeriv(
    Command (
      Seq (
        FuncDef("f", "x", Seq( Assign("y", Binop(Plus, Var "x", Int 1)), (Return (Var "y
        (Assign("a", FuncCall("f", Int 5)))
      )
    ),
    Cmd(L, L),
    func_tree,
    assign_tree
```

)

We also implement a type_checker function (check_type) that verifies if a proof (represented in an OCaml formal proof tree as defined previously) is correct.

The ‘check_type‘ function is the central piece of the implementation. It takes an inference tree, along with a context (gamma) which is a list describing the typing environment, and recursively traverses the tree to verify that types inferred at each node align with the language's type system. The function also handles subtyping relationships and function calls, verifying that argument types match parameter types and that the returned type of a function matches the expected type.

When we test if the proof is correct with respect to the typing rules, i.e when we call

```
let verification = check_type example_proof_tree [("a", Var L)]
```

we get *true*. This means that the proof is correct (with respect to the typing rules), therefore by the non-interference theorem (1) the code $f ::= fun\ (\lambda.x)\{\lambda.y := \lambda.x + 1; \textbf{return}\ \lambda.y\};\ a := f(5)$ is guarantied to verify information flow security.

5 Conclusion

5.1 Summary

This paper explored a formal method to secure the information flow in a program, by using a specific type-based approach.

We extended existing typing rules and semantics for simple commands to handle the integration of functions, a very useful tool in a code. Through a theoretical framework, we were able to build a type system made to enforce security-centric typing regulations. We proved that the typing rules uphold the non-interference property, which asserts that when two configurations differ only on public variables, any code, well typed according to these rules, applied to these configurations yields configurations differing only in public variables. This ensures that modifying public variables, which could be manipulated by malicious actors, divulges no information about private data. There's a complete absence of leakage from private to public variables.

We then developped a more practical approach by implementing and choosing certain representations to foramlly verify the well-typed nature of code. This involvbed the development of a type checker tasked with ensuring the correctness of a proof in our model.

5.2 Future directions

Our work presents potential development in the following areas :

- Automatic type inference: We are thinking about this approach, where types are automatically inferred, as a security information flow mechanism. A first step could be to develop a tool that help building a formal proof tree, by suggesting to the user, in a manner somewhat resembling the Coq language, how to proceed to prove that a phrase has a certain type.
- Machine Learning for Type Checking: One possible line of research is whether machine learning can help with the process of type checking. We may train machine learning models on secure programs in order to recognize patterns on formal proof trees and how to proceed.

References

FLEMMING NIELSON, HANNE RIIS NIELSON: Formal methods, an appetizer. Available at: `https://link.springer.com`

DAVID BASIN: Formal methods for security. Available at: `www.cybok.org`.

XAVIER LEROY: Le logiciel, entre l'esprit et la matière: Leçon inaugurale prononcée au Collège de France le jeudi 15 novembre 2018. Available at: www.college-de-france.fr

VINCENT SIMONET: Flow Caml in a Nutshell: http://cristal.inria.fr

NEVIN HEINTZE, JON G. RIECKE: The Slam Calculus: Programming with Secrecy and Integrity. Available at: https://www.cs.cornell.edu/andru/cs711/2003fa/reading/heintze98slam.pdf

GEOFFREY SMITH: A New Type System for Secure Information Flow. Available at: https://fpl.cs.depaul.edu/jriely/547/extras/smith-csfw01.pdf

DENNIS VOLPANO, GEOFFREY SMITH: A Type-Based Approach to Program Security. Available at: https://www.researchgate.net/publication/220917257_A_TypeBased_Approach_to_Program_Security