Information-Flow Control Zoo

Julius Marozas **Nahusenay Yifter** **Erik Nilsson**

Group 51

Contents

1 Abstract

This project aims to explore language-based security as it pertains to information-flow control with the additional hope that the project could serve as a good educational resource for implementing IFC languages. In the report, existing research in the field has been explored such as the aspects of non-interference and checking these properties through static analysis techniques. As an artefact of the project, a small but extendable while language has been created with a security flow checker which provides a way to verify the integrity and confidentiality of an arbitrary well-typed program. Furthermore, the flow checker has been rigorously examined to satisfy termination insensitive non-interference via property-based testing. Additionally, we analyse the limitations of the implemented flow checker by showcasing that it fails termination sensitive non-interference property. Finally, we discuss further improvements.

2 Introduction

Security is of great importance in programming since deficiencies in security can have dire consequences such as hacking, theft of sensitive information or spreading of harmful code. By integrating security principles in the development process one can reduce the risk for such attacks and thus protect both users and organizations. Language-based security revolves around this very topic of developing and integrating features and techniques that will minimize risks and vulnerabilities into the programming language itself.

2.1 Motivation and relevance to language-based security

The aim of the project is to explore the current state-of-the-art research in language-based security through information-flow control (IFC). We believe that this work could serve as a good educational resource for anyone new to IFC.

2.2 Project description and goals

2.2.1 Overview of the work

We planned to investigate unique features offered by existing IFC programming languages, discuss their differences, and create a small programming language (or embedded DSL), offering some unique features found in existing IFC languages. Additionally, if there is enough time, we will develop a demo application, a password manager or something similar.

2.2.2 The goal of the project

- Discuss unique features offered by existing IFC languages.
- Create an extendable small programming language showcasing various IFC techniques.
- (Optional) Write a demo application in each language.

3 Background

Most of the information below, with the exception of Section 3.5, originates in the paper “A perspective on information-flow control" by D. Hedin, and A. Sabelfeld. [HS12]

3.1 Information-flow control

The control of information propagation in computing systems, also known as information-flow control, is essential for information security. While access control has traditionally been used to prevent unauthorized dissemination of information, it has limitations in terms of ensuring secure handling of accessed information.

Information-flow control, on the other hand, focuses on tracking how information moves through a program during execution to ensure its secure handling. Secure information flow encompasses confidentiality and integrity aspects, with non-interference being a fundamental concept.

Confidentiality pertains to reading while integrity pertains to writing. This is to say that confidentiality refers to the assurance that information is only accessible to authorized individuals or entities. Integrity on the other hand refers that information remains uncorrupted. It guarantees that data is not modified or tampered with in an unauthorized manner.

3.2 Non-interference

Non-interference, originally described by Goguen & Meseguer in 1982, is a semantic property that ensures that “low-security behavior of the program is not affected by any high-security data”. In other words, non-interference guarantees that secret information can not leak through the observable outputs of a computation and thus a system that satisfies non-interference will behave the same way regardless of the state of private information.

3.2.1 Termination-(in)sensitive non-interference

Termination-insensitive non-interference (TINI) means that the secret information cannot be inferred from the observable outputs of a program. However, if the program terminates or not could be dependent on the secret.

Termination-sensitive non-interference (TSNI), on the other hand, is a stronger requirement. It requires that the secret information cannot be inferred from the observable outputs nor the termination behaviour.

TSNI is usually considered to be a more realistic and practical requirement for real-world applications, as programs may not always terminate in practice, and therefore, it is essential to ensure that sensitive information is not leaked during an incomplete execution. On the other hand, TINI is sometimes used in theoretical contexts because it simplifies the analysis of information flow by disregarding the possibility of incomplete executions.

3.2.2 Progress-(in)sensitive non-interference

While termination NI is about non-interactive programs (also called batch-job programs), progress NI is about interactive programs. *Progress-insensitive non-interference* (PINI) and *progress-sensitive non-interference* (PSNI) are two variants of the non-interference property that differ in their level of tolerance for interference between sensitive and non-sensitive information. PINI allows for some degree of interference, while PSNI requires the absence of any interference. They revolve around addressing the issue of when an attacker has access to inspect intermediate steps in the execution.

PINI, similarly to TINI, is the weaker form of progress non-interference that allows for some degree of interference between sensitive and non-sensitive information. The presence of sensitive information might cause a program to terminate earlier or to produce a different output, but it would not leak the sensitive information itself to an attacker.

PSNI is thus the stronger form of progress non-interference that requires that there is no interference between sensitive and non-sensitive information, even if the interference is unrelated to the value of the sensitive information.

Example: Consider a program that computes the maximum of two input values. In a progress-insensitive non-interference system, the presence of a sensitive input value might cause the program to terminate earlier than it would otherwise, but it would not allow the sensitive value to leak to an attacker. In a progress-sensitive non-interference system, the presence of a sensitive input would not affect the program's behavior in any way, except for determining the value of the maximum output.

3.2.3 Relationship between the properties

Progress non-interference can be seen as a more general concept of non-interference, encompassing interaction in addition to termination. By examining the definition of PINI and TINI, it becomes evident that if a program is progress-insensitive (PINI), it must also be termination-insensitive (TINI). Considering that PSNI is the most stringent form and that it satisfies PINI, a program meeting the requirements of PSNI will automatically meet the requirements of TINI as well. The relationship is visualized in Figure 1.

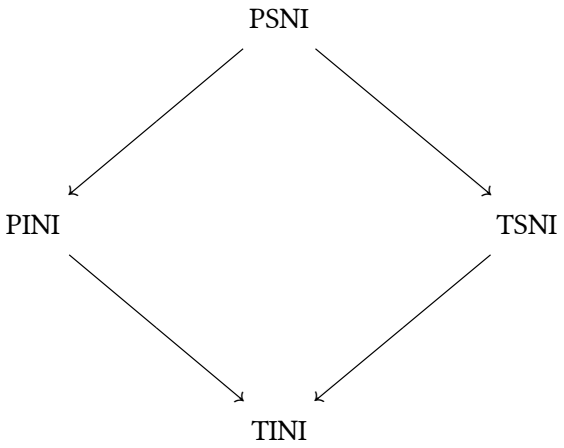

Figure 1: Lattice diagram of the four dominant non-interference properties

3.3 Concurrency

Concurrency is another phenomenon that influences security in the context of information flow. Concurrency poses a challenge because parallel composition of secure programs is not necessarily secure, and internal timing channels can lead to potential information leakage. An example is given by Hedin and Sabelfeld in "A perspective on information flow control" [HS12]:

Example: Consider the following program in which both sub-programs satisfy noninterference, but when run in parallel they leak the secret h to the public variable l.

```
if h {sleep(100)}; l = 1 | sleep(50); l = 0
```

To address this challenge, several proposed solutions have been proposed. One approach is possibilistic non-interference, which ensures that the possible low outputs of a program remain unaffected when high inputs are varied. However, this notion is only applicable to a limited range of probabilistic schedulers. Additionally, various concurrency-aware information flow control mechanisms have been developed, including probabilistic non-interference, timing-sensitive security, and bisimulation-based formulations of confidentiality. These mechanisms offer alternative forms of non-interference tailored to the specific requirements of concurrent systems.

3.4 Declassification

In general, any sensitive information should not be leaked. For instance, when attempting to log in to a system with the wrong password, there is a prompt suggesting that the password is incorrect [HS12]. Even though this is a partial leak, the system is still considered secure. Such scenarios give rise to the need for the intentional leaking of some secret information. This is what declassification is all about. In order to achieve this we can introduce a binary operator that takes an expression and the level it will be declassified to.

However, we need a way to avoid declassifying more than intended. To "securely" declassify an expression we can follow the "delimited" release principle. Which specifies that an expression can be declassified in some statement if making the value of the expression does not reveal information about secret input [JC21].

There are four main dimensions to consider for declassification.

- **what** is declassified

The "what" dimension of declassification pertains to specifying the information that is to be declassified, such as the average salary from a list of salaries that are secret. However, it is vital to ensure that the declassification policy includes a safeguard that imposes an upper limit on the released information. This ensures that the amount of information disclosed remains within acceptable boundaries and prevents unintended leaks.

- **who** is able to declassify

The "who" dimension represents another crucial aspect that focuses on determining who has the authority to control the release of information, thereby ensuring information integrity. If the attacker gains control over the declassification process, it can lead to potential vulnerabilities, such as laundry attacks, where unintended leaks are strategically hidden within the system's declassification policy [HS12]. It is imperative to establish mechanisms that prevent unauthorized entities from manipulating the declassification decisions to maintain the security of the information release process.

- **where** the declassification occurs

The "where" dimension encompasses two distinct forms of release locality, each associated with different aspects of information flow control. The first form is closely related to the "what" dimension, focusing on the specific code locality where information is released [HS12]. The second form addresses where information may flow in relation to the security levels established within the system [HS12].

This dual perspective ensures that the precise locations within the code and the security level hierarchy are carefully considered when determining the appropriate points for information release.

- **when** the declassification takes place

According to the works of Sabelfeld and Sands, the "When" dimension of declassification can be categorized into three distinct classes of temporal release: time-complexity-based, probabilistic, and relative policies [SS09]. In time-complexity-based policies, information remains undisclosed until a specific time frame has elapsed, often determined as an asymptotic measure relative to the size of the secret. On the other hand, probabilistic considerations focus on the likelihood of a leak occurring, aiming to minimize the probability of such events. This policy is regarded as temporal declassification since they reflect the infrequent disclosure of secrets [SS09]. Lastly, relative temporal policies establish temporal relationships with abstract systems, dictating when declassification events can transpire based on other events within the system. For instance, Sabelfeld and Sands discuss a scenario where the downgrading of a software key is allowed only after confirmation of payment has been received [SS09].

3.5 Existing languages

3.5.1 Jif

Jif is a programming language that provides support for information-flow control (IFC). It is designed to enforce strong security guarantees by tracking the flow of sensitive information within a program. This enforcement takes place both at compile time and run time. Jif performs static analysis by incorporating language-level constructs and a type system that allows users to annotate variables and expressions with security labels. These labels are used to enforce information flow policies, ensuring that sensitive data is not unintentionally leaked.[Ste16]

Example:

```
int {Alice→Bob} x;
```

In this scenario, the security policy specifies that the information contained in variable x is under the control of Alice, who authorizes its visibility to Bob. Expressed as {Alice←Bob}, the policy signifies that Alice is the owner of the information and grants permission for Bob to interact with it. Leveraging such label annotations, the Jif compiler conducts an analysis of information flows within programs. It's objective; ascertain whether the programs effectively uphold the principles of confidentiality and integrity, safeguarding the sensitive data involved.[Ste16]

3.5.2 FlowCAML

FlowCAML is another extension of a programming language, specifically, OCaml. It incorporates a type system for tracing information flow and ensures that a given program is in compliance with confidentiality and integrity policies. In FlowCAML, security levels are assigned as annotations to standard ML types, forming a custom lattice. With full type inference, the system verifies, without requiring explicit source code annotations, that all information flows within the analyzed program align with the specified security policy set by the programmer.

The FlowCAML compiler generates valid Objective Caml code from FlowCAML code, allowing for the compilation of FlowCAML programs using the byte-code or native compiler and executing them like any other Objective Caml program[Sim03]. Additionally, it is straightforward to interface FlowCAML

programs with existing Objective Caml code, leveraging a wide range of available libraries. However, FlowCAML lacks support for object-oriented features, polymorphic variants, and labels present in Objective Caml.

3.5.3 FlowLock

FlowLock is a research project focused on expressive, dynamic information-flow policies. It introduces a type system for a language similar to ML (meta-language), incorporating various information-flow control (IFC) techniques. FlowLock allows you to control the conditions under which data can be read by certain entities. Through static analysis of the program's source code, FlowLock is capable of identifying possible information leaks and guaranteeing the protection of sensitive data. [BS06]

The FlowLock type system allows programmers to specify information-flow policies at a fine-grained level, enabling precise control over how information propagates and is accessed within a program. The Flow Locks paper presents a type and effect system inspired by Almeida Matos and Boudol. In this system, each expression is assigned a reading effect and a writing effect. The reading effect specifies who is allowed to access the expression's result and under which lock states. Similarly, the writing effect defines which actors can observe the memory effect of the expression's execution and in what lock states. [BS06]

Type judgments then have the form

$$\Gamma; \Sigma M : \tau, (r, w) \Rightarrow \Sigma'$$

where:

- Γ is a typing environment for variables giving a type and policy for each variable.
- Σ represents the state of the program, i.e. the set of locks currently open.
- M represents an expression in the program that is being typed.
- τ represents the type of the expression M.
- (r, w) represents the reading and writing effects of the expression M, both in the form of policies.
- Σ' represents the state that the program will be in after evaluating the expression M.

  This type system allows for completely static verification of flow lock policies in an ML-like language with references.[BS06]

3.5.4 Spark Examiner

SPARK Examiner is a tool that is part of the SPARK technology and ecosystem. SPARK is a programming language (annotated) and a set of verification tools used for developing high-assurance software systems. SPARK Examiner is specifically designed to analyze SPARK programs and perform static analysis and verification of their properties.

SPARK Examiner performs analysis on SPARK programs for compliance with the SPARK language restrictions, such as type safety, absence of runtime errors, and adherence to the specified contracts and annotations[Hil16]. It does this by analyzing each variable in the program and assigning it a security level based on its annotations. It then tracks how data flows between variables with different security levels and flags any potential violations.

4 While language

4.1 Syntax

For the syntax of the language we chose the most simple option, which is also often used in the existing literature: the batch-job while language. The same language is also presented in [HS12].

$$e ::= b \mid n \mid x \mid !e \mid e_1 \oplus e_2 \mid (e) \text{ where } \oplus \in \{+, -, \times, =, <, >\}$$
$$c ::= \text{skip} \mid c_1; c_2 \mid \text{if } e \text{ then } c_1 \text{ else } c_2 \mid \text{while } e \text{ do } c \text{ done} \mid x := e$$

4.2 Semantics

We first give the big-step operational semantics for the expressions of the language Figure 2. Going from left to right, we have that:

- (first 2 rules) integer and boolean literals evaluate to themselves regardless of the environment,
- (rule 3) the value of a variable is looked up in the environment,
- (rule 4) a binary expression is evaluated by first evaluating the operands to their values and then applying the operation.

$$\langle n, E\rangle \Rightarrow n \quad \langle b, E\rangle \Rightarrow b \quad \frac{E[x] = v}{\langle x, E\rangle \Rightarrow v} \quad \frac{\langle e_1, E\rangle \Rightarrow v_1 \quad \langle e_2, E\rangle \Rightarrow v_2}{\langle e_1 \oplus e_2, E\rangle \Rightarrow v_1 \oplus v_2}$$

Figure 2: Big-step semantics for the expressions

Next, we have the big-step semantics for the commands Figure 2:

$$\frac{}{\langle \text{skip}, E\rangle \Rightarrow E} \quad \frac{\langle c_1, E_1\rangle \Rightarrow E_2 \quad \langle c_2, E_2\rangle \Rightarrow E_3}{\langle c_1; c_2, E_1\rangle \Rightarrow E_3}$$

$$\frac{\langle e, E_1\rangle \Rightarrow \text{true} \quad \langle c_1, E_1\rangle \Rightarrow E_2}{\langle \text{if } e \text{ then } c_1 \text{ else } c_2, E_1\rangle \Rightarrow E_2} \quad \frac{\langle e, E_1\rangle \Rightarrow \text{false} \quad \langle c_2, E_1\rangle \Rightarrow E_2}{\langle \text{if } e \text{ then } c_1 \text{ else } c_2, E_1\rangle \Rightarrow E_2}$$

$$\frac{\langle e, E_1\rangle \Rightarrow \text{true} \quad \langle c, E_1\rangle \Rightarrow E_2 \quad \langle \text{while } e \text{ do } c \text{ done}, E_2\rangle \Rightarrow E_3}{\langle \text{while } e \text{ do } c \text{ done}, E_1\rangle \Rightarrow E_3}$$

$$\frac{\langle e, E\rangle \Rightarrow \text{false}}{\langle \text{while } e \text{ do } c \text{ done}, E\rangle \Rightarrow E} \quad \frac{\langle e, E\rangle \Rightarrow v}{\langle x := e, E\rangle \Rightarrow E[x \mapsto v]}$$

Figure 1: Big-step semantics for the commands

4.3 Security checks

The program is checked for any TINI as well as TSNI violations, explained in detail in Section 5.3.

$$\begin{aligned}\text{TINI}_\Gamma(c) = \forall E_{11}, E_{12}, E_{21}, E_{22}.\\ E_{11} \sim_\Gamma E_{12}\\ \wedge \langle c, E_{11}\rangle \Rightarrow E_{21}\\ \wedge \langle c, E_{12}\rangle \Rightarrow E_{22}\\ \Longrightarrow E_{21} \sim_\Gamma E_{22}\end{aligned}$$

$$\begin{aligned}\text{TSNI}_\Gamma(c) = \forall E_{11}, E_{12}, E_{21}.\\ E_{11} \sim_\Gamma E_{12}\\ \wedge \langle c, E_{11}\rangle \Rightarrow E_{21}\\ \Longrightarrow \exists E_{22}.\\ \langle c, E_{12}\rangle \Rightarrow E_{22}\\ \Longrightarrow E_{21} \sim_\Gamma E_{22}\end{aligned}$$

5 Implementation

5.1 Tagless Final

The syntax tree of the language is represented as a typeclass instead of a sum data type. The advantage is that this representation allows for extending the language in separate modules.

Example:

With the initial encoding, the syntax tree for a language supporting addition might look like this.

```
data Expr
  = Lit Int
  | Add Expr Expr
```

For comparison, the final encoding would be the following:

```
class Expr e where
  lit :: Int -> e
  add :: e -> e -> e
```

Now, if we want to add subtraction to the language, we have to modify the original data type in the initial encoding. However, in the final encoding we can just add a new typeclass.

5.2 Passes

5.2.1 Abstract

The abstract syntax tree represents the surface language that corresponds to what the user programs in. The tree is represented as two typeclasses `While` for commands and `Expr` for expressions:

```
class (Expr (WhileExpr c)) => While c where
  type WhileExpr c
  skip_     :: c
  semicolon :: c -> c -> c
  if_       :: WhileExpr c -> Then c -> Else c -> c
  while_    :: WhileExpr c -> c -> c
  (.=)      :: Name -> WhileExpr c -> c

class Expr e where
  bool_ :: Bool -> e
  int_  :: Int -> e
  var_  :: Name -> e
  not_  :: Expr e => e -> e
  (+.), (-.), (*.), (==.), (<.), (>.)
        :: Expr e => e -> e -> e
```

5.2.2 Typed

The typed syntax tree is generated by the type-checker from the abstract syntax tree. It is represented below as two typeclasses once again:

```
class (Expr (WhileExpr c)) => While c where
  type WhileExpr c :: Type -> Kind.Type
  skip_     :: c
  semicolon :: c -> c -> c
  if_       :: WhileExpr c 'TBool -> c -> c -> c
```

```
  while_    :: WhileExpr c 'TBool -> c -> c
  ass_      :: Name t -> WhileExpr c t -> c

class Expr e where
  bool_   :: Bool -> e 'TBool
  int_    :: Int -> e 'TInt
  var_    :: Name t -> e t
  not_    :: e 'TBool -> e 'TBool
  arithOp :: ArithOp -> e 'TInt -> e 'TInt -> e 'TInt
  relOp   :: RelOp -> e t -> e t -> e 'TBool
```

This time the `Expr` class has an additional type parameter for the type of the expression. The expression in `while` and `if` conditions are ensured to be of type `Bool`, and the expression in the assignment is checked to be of the same type as the variable. `ArithOp` and `RelOp` are arithmetic $(+, -, *)$ and relation $(=, <, >)$ operators, respectively.

5.2.3 Secure Flow

After the type-checker, the secure flow is checked according to the functions below.

The flow checker carries an implicit context `SecContext`:

```
data SecContext = SecContext
  { secMap :: SecurityMap
  , secLevel :: Level
  }
```

Here:

- `SecurityMap` is a map from variables to security levels
- `Level` is a type with two cases: `High` and `Low`

The context is propagated to all the functions below via the `ReaderT` monad transformer.

For `if` commands, the local security-level for the branches is inherited from the level of the expression in the condition. In other words, this means that if the condition is "high", assignments to "low" variables is disallowed.

```
  if_ e c1 c2 = SecFlow $ do
    eLevel <- checkExpr e
    local (updateLevel eLevel) $ do
      secCheck c1
      secCheck c2
```

Analogously `while` commands inherit the level from the expression in the condition as well. Meaning that a "low" variable cannot be assigned if the condition was "high".

```
  while_ e c = SecFlow $ do
    eLevel <- checkExpr e
    local (updateLevel eLevel) $ do
      secCheck c
```

Finally, the assignment is checked to disallow sensitive (high) information to flow into the low variables. More specifically, an error is raised if the upper bound of the level of the expression with local level is high and the variable level is low.

```
  ass_ x e = SecFlow $ do
    xLevel <- lookupLevel' x
```

```
    eLevel <- checkExpr e
    pLevel <- asks secLevel
    unless (eLevel <> pLevel <= xLevel) $ do
      throwError $
        AssignmentOfHighToLow
          { varName = untypedName x
          , varLevel = xLevel
          , expLevel = eLevel
          }
```

5.3 Testing methodology

The tests, in short, work by looking for deviation between two low-equivalent environments in which the same program is being executed. For two environments to be low-equivalent means that they have the same low variables with the same values, the high-level variables, though, can vary.

Thus, if executing the same program in two initial low-equivalent environments, results in environments that are no longer low-equivalent, there has been some leak from the high variable to the low one. For testing, the input environments are randomly generated with the constraints that they have to be low equivalent.

For testing TSNI the program is further tested for different termination behaviour, exempli gratia: execution in one of the low-equivalent environments terminates while the execution in the other environment does not

Checking whether a program terminates or not is quite an onerous task, videlicet it is proven that no method exists whereby one can with absolute certainty determine if a program will terminate or not (The Halting Problem). As such, in this implementation, a program is determined as non-terminating if it exceeds a time boundary like `timeout 1000`:

```
runOrDiverge :: (Env -> c -> Env) -> Env -> c -> Maybe Env
runOrDiverge evalIn env c = do
  unsafePerformIO $ timeout microseconds $ pure $! evalIn env c
 where
  microseconds = 1_000
```

6 Results

We test our implementation of the language and security checks with multiple programs and randomly generated input environments (via property based testing). In the test-suite, each program, unless specified otherwise, goes through the following passes:

1. Type-checking the program
2. Checking the secure flow of the typed program
3. Finally, if the program passes the aforementioned checks, it is tested with random initial environments by non-interference properties (id est, TINI, TSNI, PINI, PSNI).

The final step is implemented in Haskell's `falsify` property-based testing library. It is similar to QuickCheck, but among other things has better shrinking by default which makes it better at finding smaller counterexample and thus debugging is easier.

6.1 Examples

In the below examples, variables `l` and `h` are of low and high security-level, respectively.

6.1.1 TINI checking (explicit & implicit flow)

An example program with the explicit flow is the simplest test case employed for testing this language:

```
l := h
```

Because the information flows from the high-level variable to the low-level one, the sensitive information is leaked, and thus the program is insecure.

The TINI checker, given this program directly after type-checking, will notice the explicit flow and will subsequently fail the program for violating the termination-insensitive non-interference property.

Running the test case as described above (without any security checks) produces the following output:

```
IFC-Zoo
  TINI
    explicit flow:   FAIL (0.01s)
      failed after 2 shrinks
      Execution in initially low-equivalent environments:

        env11 = {h |-> true; l |-> false}
        env12 = {h |-> false; l |-> false}

      resulted in low-inequivalent environments:

        env21 = {h |-> true; l |-> true} /= {h |-> false; l |-> false} = env22

      Evaluation:

        <c, env11 = {h |-> true; l |-> false}> =>
            env21 = {h |-> true; l |-> true}

        <c, env12 = {h |-> false; l |-> false}> =>
            env22 = {h |-> false; l |-> false}
```

As we can see above, the `falsify` library found two low-equivalent environments that when fed to the program outputs environment that are no longer low-equivalent.

If the same program were to first be checked with the implemented security flow checker it would give the following error message, showing that there is an explicit flow leak.

```
ERROR: Flow from low expression to low variable "l" violates security policy
```

The property tests are implemented so that they discard the test programs if they fail to typecheck (or flow check). The remaining ones are tested to satisfy the TINI property.

```
IFC-Zoo
  TINI
    explicit flow:   OK
      100 successful tests
    implicit flow:   OK
      100 successful tests
```

Here, the program is also checked for implicit flow, where information leaks indirectly. Below is a simple example of a program showcasing implicit flow and it can be seen that, although the information from the high-level variable is not assigned directly to the low-level variable, the result is that the low-level variable will acquire the same value as the high-level one and thus the sensitive information is leaked.

```
if h then l := true
```

6.1.2 TSNI checking (non-termination)

Checking for TSNI-compliant termination behaviour is also done. If a program fails the TSNI checking then the user would be met with the following error message.

```
  TSNI
    non-terminating: FAIL (0.10s)
      failed after 2 shrinks
      ◊ /= {h |-> true; l |-> false}
      Logs for failed test run:
      generated ({h |-> false; l |-> false}, {h |-> true; l |-> false})
```

The non-terminating program is implemented as follows:

```
if l == h then
  while true do
    skip
  done
```

The above program is insecure since the attacker could leak the secret inside the `h` variable by checking if the program terminates or not.

The flow checker doesn't fully satisfy the TSNI property since the termination of the program is not statically analysed. Some potential ways of solving this problem are given in the next section, Section 7.2.

7 Discussion & Conclusion

7.1 The language's features and usefulness

The developed while language implements many an aspect of non-interference and subsequently thus also explores the use of information-flow control. The language checks for both implicit and explicit flow leaks at the typechecking stage and a user will thus be warned about the vulnerability and can take care of it before it has a chance to be exploited by a potential attacker. Additionally, testing for leakage by the way of termination behaviour is also possible which is needed to satisfy TSNI. Furthermore, the language was made in a way that is extendable and which facilitates adding new functionality.

7.2 Limitations and future research

One of the limitations is that the developed while language is quite small and could be further expanded and developed to include more operations and the like for it to be usable for a larger scale program. Some examples are implementing: declarations, PINI, PSNI, arrays as well as input and output.

Currently, when a program fails to typecheck or flow check the error message is not informative as to where the origin of the user's faux pas lies. And as such the checker's errors might infuriate the user even though they are themselves in wrength. Additionally, the errors could include more information about the state of the security levels in the program, for instance in the implicit flow program, the error message could indicate that the security level is inherited from the if statement condition.

Another possible expansion that could be made is to better estimate the termination behaviour of a program. As stated it is now implemented by means of checking if the executions exceed a time boundary, but other methods of termination analysis could be tested that can better gauge if a program will terminate or not. However, it must be noted that there is yet no algorithm that can solve the problem completely where the problem of termination analysis also ties in with the famous halting problem.

7.3 Summary & review of goals

In summary, existing research in information-flow control as it relates to language-based security has been studied and the feautures offered by IFC have been discussed. Furthermore, a while language has been created that implements various IFC-techniques and aspects of non-interference. The results have then been discussed and further research and extentsion opportunities have been pointed out, like implementing more operations in the while language as well as more advanced termination analysis.

By going over information-flow control and implementing several of its techniques in the while language, it is also hoped that the project can serve as a good educational resource for people new to IFC.

8 Appendix

Specific contributions of each group member are as follows:

- **Julius:** I implemented the artefact of the project (the while language in Haskell) and wrote some parts of the report including Section 3.2.3, Section 4, Section 5, Section 6.
- **Erik:** In the report I wrote various sections. I wrote most of the beginning of the "Background" (Section 3), until (Section 3.2.3). Furthermore, I wrote most, but not all, of the conclusion and discussion section, Section 7 as well as a lot of the "Testing methodology" section, Section 5.3.
- **Nahusenay:** I researched relevant papers, manuals, and articles. In this report, I contributed to Section 2, Section 3, Section 3.5.

References

[Sim03] V. Simonet, "The flow caml system (version 1.00): documentation and user's manual," 2003. [Online]. Available: https://www.normalesup.org/~simonet/soft/flowcaml/manual/index.html

[BS06] N. Broberg, and D. Sands, "Flow locks: towards a core calculus for dynamic flow policies," in *Program. Languages Syst.*, Berlin, Heidelberg, 2006, pp. 180–196.

[SS09] A. Sabelfeld, and D. Sands, "Declassification: dimensions and principles," *J. Comput. Secur.*, vol. 17, no. 5, p. 517, Oct. 2009.

[HS12] D. Hedin, and A. Sabelfeld, "A perspective on information-flow control," in *Softw. Saf. Secur.*, 2012.

[Ste16] K. V. L. Z. Stephen Chong Andrew C. Myers, "Jif," 2016. [Online]. Available: https://www.cs.cornell.edu/jif/

[Hil16] R. C. A. Hilton, "Enforcing security and safety models with an information flow analysis tool," 2016. [Online]. Available: https://www.cs.cornell.edu/jif/

[JC21] T. Jensen, and S. Castellan, "Software analysis and security (sos)," 2021. [Online]. Available: https://www.irisa.fr/celtique/teaching/SOS/