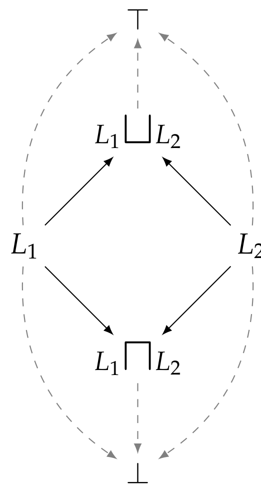
C Timed Information Flow

Master's Thesis

Mikael Elkiær Christensen
Mikkel Sandø Larsen



Aalborg University
Department of Computer Science

Copyright © Aalborg University 2016

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
http://cs.aau.dk

Title:
C Timed Information Flow

Theme:
IoT Security

Project Period:
Spring Semester 2016

Project Group:
DES105F16

Participant(s):
Mikael Elkiær Christensen
Mikkel Sandø Larsen

Supervisor(s):
René Rydhof Hansen
Mads Chr. Olesen

Copies: 2

Page Numbers: 74

Date of Completion:
May 31, 2016

Source code:
`https://github.com/des105f16/Editor/tree/d074ede4d55c6565b9d6eac651e24dff32d754cf`

Abstract:

This report describes C Timed Information Flow (CTIF). CTIF provides a tool which can take new and existing C source code, extending upon the syntax with the concept of security policy labels. The tool can then, based on the labeling, perform a check while providing information about any potential breaches of security as labeled information flows through the program.
These security policy labels are based on *The Decentralized Label Model* (DLM). The report takes important concepts of DLM and provides an extended description as well as formalization in regards to the inferrence of security policy labels. Additionally, an extension to the security policies is provided, by allowing the expression of time policies, which similarly will be checked by the tool.
The time policies were created with simplicity and practical applications in mind. As a first step in formalizing the time policies it will be shown how they can be translated into *timed automata*. In order to ascertain the practical applications, the time policies will be compared with *The Timed Decentralized Label Model* – which takes a more formal approach.

The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.

Contents

Summary **vii**

Preface **ix**

1 Introduction **1**
- 1.1 Security problems . . . 2
  - 1.1.1 Privacy . . . 2
  - 1.1.2 Time constraints . . . 3
- 1.2 Aiding development of secure software . . . 3
- 1.3 Related work . . . 3
- 1.4 Running examples . . . 4
  - 1.4.1 Smart meter bill calculation . . . 4
  - 1.4.2 Password checker . . . 5

2 The Decentralized Label Model **9**
- 2.1 Labels and policies . . . 9
- 2.2 Security class lattice . . . 11
  - 2.2.1 Composite labels . . . 12
  - 2.2.2 Label comparison . . . 12
- 2.3 Channels . . . 13
- 2.4 Implicit flows . . . 13
- 2.5 Authority and declassification . . . 14
- 2.6 Label constraints . . . 15
- 2.7 Inferring labels . . . 17
  - 2.7.1 Label types . . . 18
  - 2.7.2 Output channels . . . 19
  - 2.7.3 Inference algorithm . . . 21

3 Decentralized Label Model in C **25**
- 3.1 Scope . . . 25
- 3.2 Informal description . . . 26
  - 3.2.1 Function declarations . . . 27
  - 3.2.2 Variable declarations . . . 28
  - 3.2.3 If-acts-for and declassification . . . 28
  - 3.2.4 Inference . . . 29

3.3 Constraint extraction 31
3.3.1 Syntax 31
3.3.2 Semantic setup 36
3.3.3 Program and declarations 38
3.3.4 Label and policy 39
3.3.5 Function declarations 40
3.3.6 Variable declaration 41
3.3.7 Statements 42
3.3.8 Control structures 43
3.3.9 Assignment and return statements 44
3.3.10 Acts for statements 44
3.3.11 Expressions 45
3.3.12 Declassification 46
3.3.13 Function call 47

4 Time Policies **49**
4.1 Extending the security model 49
4.1.1 Expressiveness 50
4.1.2 Applying time policies to the examples 51
4.1.3 Inference of time policies 52
4.1.4 Time constructs 55
4.1.5 Selecting a policy 56
4.1.6 Applying time constructs to the examples 57
4.2 Timed automata 59
4.2.1 Time policies and timed automata 62
4.3 The Timed Decentralized Label Model 65
4.3.1 Usage 65
4.3.2 Comparison 66

5 Conclusion **69**
5.1 Runtime model 70
5.2 Code generation 70

Bibliography **73**

Summary

This report presents C Timed Information Flow (CTIF), which resulted in a tool of the same name. The CTIF tool allows for writing C programs, with the ability to express security policies in order to ensure information flow. These security policies are based upon those of *The Decentralized Label Model* (DLM). Further, these security policies are extended with the ability to express simple and powerful time policies.

The project provides two main contributions. Firstly, formalizations of important DLM concepts are provided. The concepts related to label inferrence, and constraint checking – the main driver of inferrence, have been formalized by providing an extended description, as well as a denotational semantics. Secondly, an extension to the DLM security policies has been provided in the form of time policies. These time policies were created as to have a focus on their practical applications. However, it is also shown how they can be formalized by being translated into *timed automata*, as well as a comparison with a related model *The Timed Decentralized Label Model*.

The tool itself was written in C#, employing the SablePP toolbox[1] to generate the compiler, which allows for easy extensibility. The use of SablePP allowed us to incrementally extend our C language scope, increasing the practical applications of CTIF. It also allowed for great flexibility in defining both syntax and semantics, providing means for fast prototyping and experimenting.

CTIF gives programmers a practical and highly usable way of annotating new and existing C source code, allowing for easy analysis of information flow while requiring only minor extensions. Very few constructs are added to the C syntax, as to not over-complicate its potential uses, while keeping it both highly useful and powerful. This is true for the security policies, as well as time policies.

[1] `https://github.com/deaddog/SablePP`

Preface

This report has been prepared by 10th semester Software Engineering students at Aalborg University, during the spring semester of 2016. It is a master's thesis project within the area of distributed and embedded systems.

It is expected of the reader to have a background in IT/software, due to the technical content.

References and citations are done by the use of numeric notation, e.g. [1], which refers to the first item in the bibliography. Footnotes will be used for referring to related internet articles.

We would like to thank our supervisor René Rydhof Hansen and co-supervisor Mads Chr. Olesen for their excellent supervision throughout the project period.

Aalborg University, May 31, 2016

Chapter 1

Introduction

The Internet-of-Things (IoT) is an ever-growing phenomena, becoming apparent in more and more aspects of our everyday lives.[1] However, security in IoT is not widely standardized, and therefore the quality of any such device will depend on the individual manufacturer (as well as any hobbyist making his own devices). There is still much that can be done in order to expand this area.

In this chapter we will present the needed background information which serves as the base for our project. We will, based on our pre-specialization [1], introduce some of the security problems found in the IoT. Specifically, we will look into privacy issues associated with IoT. We will also discuss the possibility of handling some of these privacy issues with time-based constraints. Additionally, we will discuss some problems related to the development of secure software, which will be related to our solution. Following the description of our solution, we will present some related work. Finally, we will present two code examples, which will be used throughout the report, to serve as IoT use-cases.

The remainder of the report is structured as follows. In Chapter 2 is provided the necessary background information for understanding the first part of our solution. Then we will start presenting our solution in Chapter 3, by providing an informal description, as well as a formal definition, of how labels are extracted and checked. Then in Chapter 4 this solution is extended, by introducing time policies, which we will compare to existing theory. Finally, the project is concluded in Chapter 5 along with a description of possible future works.

[1] https://www.theguardian.com/technology/2015/may/06/what-is-the-internet-of-things-google

1.1 Security problems

In [1] we investigated problems involving the imminent implementation of smart meters throughout Europe.[2] Here we discovered a wide array of problems, ranging from ethical to practical. One of the more interesting discoveries that we made was the newly acquired problems associated with increasing the availability of a previously physically isolated device. This is exactly what is going to happen when the current electrical meters are going to be substituted with remotely accessed and controlled smart meters. It will open up for problems already found in similar devices, which in general is any IoT device.

One of the focuses we had in [1] was on the privacy issues related to such an exposed device. We would like to explore this problem further, while coming up with a solution that will aid the programmer in the securing IoT devices. Many of the security problems associated with privacy could possibly also be minimized by adding the concept of time constraints, which we will also discuss here.

1.1.1 Privacy

With the emergence of IoT, more and more of our devices are being globally exposed, for both good and for worse. There will be endless possibilities in easing our everyday lives, both at a personal level and at a communal level. At the personal level, we will be able to access devices that monitor and control our homes. On the communal level, many of the tasks that previously had to be done by people can now be pushed to remote devices, such as monitoring the health of patients or changing traffic signs on a freeway.[3]

All these things are associated with major potential risks, should they be compromised. At the personal level, we do not wish to give up any information about our personal lives, and we don't want anyone else controlling our devices. At the community level the risks can be even larger, especially those devices linked to health care, such as a "smart" pacemaker.

Some of these risks are associated with faulty protocols or erroneous implementations of protocols. However, privacy can still be an issue even with a "good" protocol. The problem here is that it is not easy ensuring that the information managed by an IoT device is distributed as intended. This is especially true if different policies apply to different users, as is more than likely with IoT devices. This could for instance be power consumption data which have different policies depending on who is accessing the data, e.g.: the consumer itself, the electrical company, or the hardware manufacturer. It is also possible to have *passive cheaters*, users of the system who follow the rules, but try to acquire more information than is intended possible.

[2] `http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2009:211:0055:0093:EN:PDF`

[3] `http://techcrunch.com/2015/10/24/why-iot-security-is-so-critical/`

1.1.2 Time constraints

Properly handling timing within a system is an already-existing and quite common problem.[4] It is something that is hard to both model and ensure that any such modeling holds. This is especially true for IoT devices which operate in the real world where any potential mistakes can result in great consequence.

To further the concept of protecting privacy, it would make sense to limit functionality of a device based on time constraints. By doing this, it could be further ensured that any leak of information from a system is minimal. Examples of this could be to limit the amount of authentication requests within a certain time interval, or how often personal data can be read by outside parties.

1.2 Aiding development of secure software

When developing any kind of software, developers must take many things into account. Even when developing software using high abstraction programming languages and/or frameworks, a lot can go wrong when having to take into account the multitudes of aspects related to a problem domain. If we also add security, specifically the modeling of security policies, even more can go wrong if not properly handled.

Applying more tools and techniques to ensure security can be cumbersome and time-consuming, and could ultimately end up in complete omission if it requires too much effort. This is why we would like to create a tool which could assist developers in gaining more insights to the security aspects of the software to be written. We imagine the use of the tool being as seamless as possible, as to not diminish its usefulness.

We will develop a tool, which will give the user the ability to actively create and modify security policies while implementing software, and receive feedback about breaches of these, in similar style of a modern IDE displaying compiler errors. We will take a bottom-up approach, in order to maintain high usability for the programmer, such as ourselves. This tool will be based on the security policies of the Decentralized Label Model (DLM), with a focus on inferring labels. Additionally, we will introduce our simple time policies, as an extension to the DLM policies, which will be similarly usable and provide similar feedback about time-breaches. The tool will allow for applying these security policies to C programs, as C is a widely used language for low level implementations, such as an IoT device.

1.3 Related work

Despite DLM having been around for near two decades now [2], not much work has been put into putting the theory to use. One exception is the *Java Information Flow* (JIF)[5] project,

[4] `http://www.nist.gov/pml/div688/timing-031915.cfm`

[5] `https://www.cs.cornell.edu/jif/`

a project related to the original authors. However, by using JIF, and thereby Java and a Java Virtual Machine (JVM), one is somewhat limited in actual applications, especially in regards to IoT devices. This is why we explore the possibility of applying DLM to C.

Two recent projects have also explored this; *C Information Flow* (CIF) [3] and *Content-Based Information Flow Control* (CBIF) [4]. These projects are very similar and demonstrate some of the properties that we seek to develop as well. All three extend C with DLM constructs, making it possible to annotate code with security policies, and statically check that policies are not violated throughout program flow. None of the three discuss the run-time concepts of DLM.

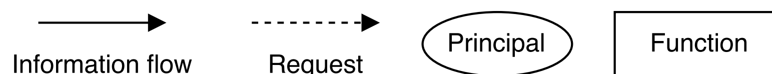
In CIF, a simple approach has been chosen, having only the bare essentials needed for static checking. Besides providing static checking of DLM policies, CIF also outputs information flow graphs, providing a visual overview of the flow for a program.

In CBIF, a very different approach is used, focusing on extending the syntax with new constructs, so that model checking can be applied. CBIF is also very domain-specific, having a focus on avionics-related use.

Not much is said about inferring labels in neither CIF nor CBIF, which is why CTIF will explore this further. Additionally, CTIF will explore the possibility of extending the security policies with time constraints, adding further possibilities for modeling security of programs.

1.4 Running examples

We have created two examples which will work as IoT use-cases, and will be the base for our following discussions and implementation. Attached to each example is also a graph visualizing the program information flow, with the following legend:



In the graphs we have two abstractions that are not apparent when looking at the source code. The main function has been left out, and the *request* flows represent calls to the program through some means (e.g. a call from another program or through a web-request). In the code source, the logic in the main function will also not represent an actual implementation, as we have abstracted away from how the program will actually be called.

1.4.1 Smart meter bill calculation

Related to the protection of smart meter data, we have created a simple example (see Listing 1.1 and Figure 1.1) which uses data from both consumer and electrical company in order to calculate a bill. Here we make use of 4 declared auxiliary functions: `get_latest_usage`, `get_latest_prices`, `send_to_consumer`, and `send_to_electrical_company`. The actual

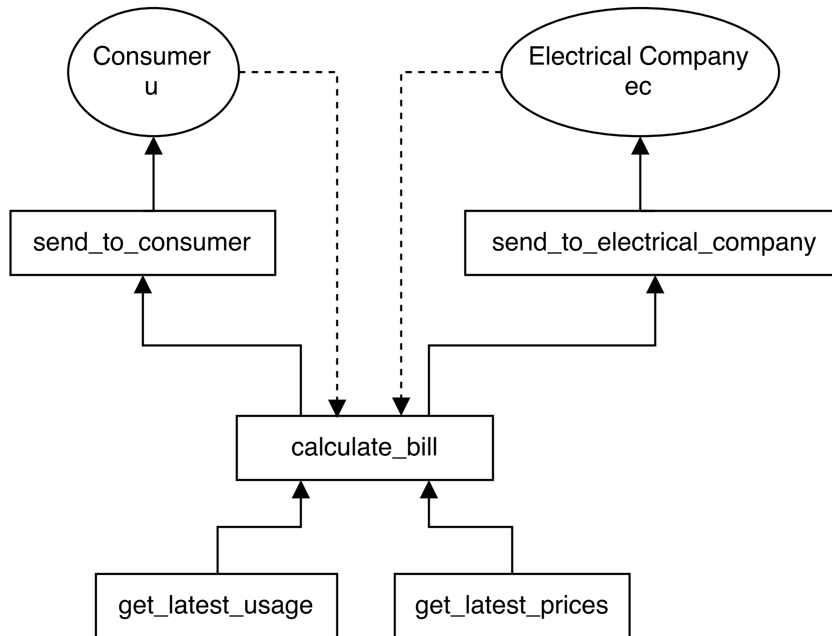


Figure 1.1: Information flow graph for `calculate_bill`

implementation of these is not important, they are only seen to represent ways of either obtaining data from outside the program, or sending data to outside of the program. Similarly, the implemenation of the `calculate_bill` function is not important, it is however worth noting that in even such a small example it can easily become difficult to assert how information flows.

Updated versions of the example, appearing later:

- With inferred labels – Listing 3.3, page 34
- Added use of time constructs – Listing 4.2, page 61

1.4.2 Password checker

This more general example represents a means of authentication, which could be found in many different IoT devices. The password checker (see Listing 1.2 and Figure 1.2) takes a username and password combination and checks it against the user database, giving a response to the user whether it was correct or not. Here we have 3 auxiliary functions: `get_login`, `get_users`, and `send_response` – they represent the two inputs needed, as well as the response to be given.

Updated versions of the example, appearing later:

- Fully labeled – Listing 3.1, page 32

```c
typedef struct usage {
  int start_time;
  int usage_in_Wh;
} usage;

typedef struct price {
  int start_time;
  int price_in_cents;
} price;

usage *get_latest_usage();
price *get_latest_prices();
void send_to_consumer(int bill_total);
void send_to_electrical_company(int bill_total);

int calculate_bill() {
  int usage_count = 100;
  int prices_count = 100;
  usage *latest_usage = get_latest_usage();
  price *latest_prices = get_latest_prices();
  int result = 0;

  int i = 0;
  int j = 0;
  while (i < usage_count) {
    while ((j < prices_count-1) && (latest_prices[j+1].start_time <=
    latest_usage[i].start_time)) {
      j = j + 1;
    }
    result = result + latest_usage[i].usage_in_Wh * latest_prices[j].
    price_in_cents;
    i = i + 1;
  }
  return result;
}

int main(int argc, char **argv) {
  int bill_total = calculate_bill();
  send_to_consumer(bill);
  send_to_electrical_company(bill);
}
```

Listing 1.1: Smart meter bill calculation example

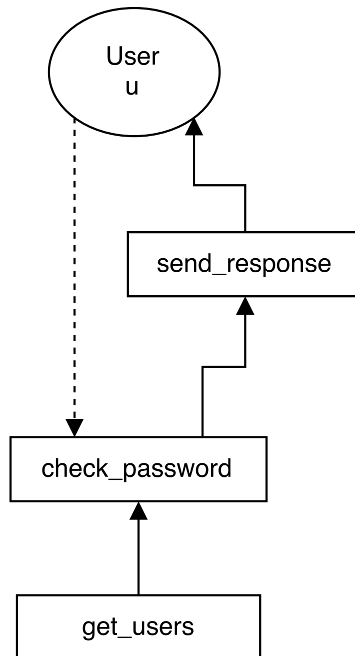


Figure 1.2: Information flow graph for `check_password`

- With inferred labels – Listing 3.2, page 33
- Added use of time constructs – Listing 4.1, page 60

```c
#include <stdbool.h>
#include <string.h>

typedef struct user_info {
  char username[20];
  char password[20];
} user_info;

user_info get_login(){}
user_info *get_users(){}
void send_response(bool is_match){}

bool check_password(char *username, char *password) {
  int user_count = 100;
  user_info *users = get_users();
  int i = 0;
  bool match = false;

  while (i < user_count) {
    if (!strcmp(users[i].username, username) && !strcmp(users[i].
   password, password)) {
      match = true;
    }
    i = i + 1;
  }

  return match;
}

int main(int argc, char **argv) {
  user_info login = get_login();
  bool is_match = check_password(login.username, login.password);
  send_response(is_match);
}
```

Listing 1.2: Password checker example

Chapter 2

The Decentralized Label Model

The Decentralized Label Model [2, 5, 6] is a model for ensuring information flow control in a system. This is done by annotating source code with security policies, in the form of labels attached to data-holding constructs. This chapter will present the necessary information needed to understand the implementation presented in Chapter 3. The descriptions and definitions in this section are based on [2, 5, 6]. Throughout the examples, we will use the same grammar as that used by our implementation.

In this presentation of DLM some details have been left out. These details are related to run-time concepts, such as *principal hierarchy* and *authority*. Our focus will be on the concepts needed for compile-time checking. The impliciations of this limitation will be further discussed throughout.

Finally, in Sections 2.6 and 2.7 we will present our own formalizations related to the concept of *label inference*, which is briefly introduced in [2]. We will use these formalizations as a base in order to define the label inference algorithm, which is an essential part of our implementation.

2.1 Labels and policies

Throughout a program, values are declared, initialized, and assigned to variables and other value-holders. Value-holders are collectively known as *slots*, which cover constructs such as variables, structs, and other storage locations. In order to ensure that a certain assignment is legal, such that no information is unintentionally leaked, we assign *labels* to slots so that their "security" can be compared. This way we can ensure that an assignment is only legal in the cases where higher-security values aren't assigned to lower-security slots. In [2] labels are seen as an extension to types, and would therefore be applicable anywhere a type normally would be applicable.

Each slot is associated with a label, that describes how the data in the slot can be handled. We denote the label of slot s as $\underline{s}$. Referring to labels in this fashion serves the same

purpose as letters in algebra; it allows for handling labels without knowledge about their actual value.

For a given label it is possible to define both read (*privacy*) and write (*integrity*) policies. The first represents information flow out of a system and the latter information flow into a system. In this report only read policies will be considered. Due to the relation between the two types of policies, we note that write policies can be ensured in a fashion similar to read policies. Because of this we choose to simplify the scope of the report such that it only discusses read policies.

Principals

In the following, we employ the concept of *principals*. A principal (or *subject*, *agent*) is an entity in a system that represents some interest. In short, principals represent real-world users or the authority under which a program/system runs.

In the following we make use of the special set of principals; $*$. This set contains all the principals in a system.

Labels

A label can be described as a set of policies, where a policy consists of an owner principal o and a set of reader principals $r_1, r_2, \ldots, r_n$. Formally we provide the following, equivalent definition:

> **Definition 2.1 (Labels)**
> A label L is a set of owners $owners(L)$, and a function $readers(L, o)$ that retrieves the set of reader principals that o allows to read. Note that given this definition we have that
>
> $$o \notin owners(L) \Rightarrow readers(L, o) = *$$

Owners are allowed to change their own policies within a label. We describe how this is done in Section 2.5. If an owner is not part of its own policies reader set, that owner is not allowed to read from the slot associated with the label. He is however still allowed to change his policy.

The effective reader set

To ensure that the policies of all owners in a label are enforced, only readers that all owners "agree" can read from the slot associated with the label. This is known as the *effective reader set*, which is the intersection of *all* reader sets of a label:

> **Definition 2.2 (The effective reader set)**
> The reader set of a label L is the set of readers that are in the reader set of all owners:
>
> $$readers(L) = \bigcap_{o \in *} readers(L, o)$$

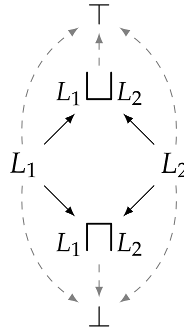


Figure 2.1: Abstraction of security lattice

Example 2.1 (A label with two policies)
Below is an example of a label with two policies using the notation from above;

$$L_1 = \{o1 \rightarrow r1, r2; o2 \rightarrow r2, r3\}$$

The owner set of the label is $owners(L_1) = \{o1, o2\}$, and its reader sets are $readers(L_1, o1) = \{r1, r2\}, readers(L_1, o2) = \{r2, r3\}$. By intersecting the reader sets of the label we get the effective reader set. In practice we can disregard the reader set of labels that are not owners. Thus the effective reader set is

$$readers(L_1) = \{r1, r2\} \cap \{r2, r3\} = \{r2\}$$

2.2 Security class lattice

A set of DLM labels can be seen as a *security class lattice* [2, pp. 6-7]. A lattice is a partially ordered set, for which each pair of elements in the set have a unique least upper bound (the join/$\sqcup$ of the labels) and a greatest lower bound (the meet/$\sqcap$ of the labels). In the case of DLM, the lattice is the set of all labels and it is ordered by how restrictive the labels are.

A label without owners effectively has no restrictions on the data it protects. As such it is the lower bound $\bot$ of the lattice. Likewise the label that has all owners in a system (as defined in Definition 2.1), but no readers allowed by either of the owners, is the most restrictive label. This is the upper bound $\top$ of the lattice.

With these we have a bounded lattice as illustrated by Figure 2.1. In the following sections we define the properties of this security lattice. These will match the description of the lattice bounds given above.

2.2.1 Composite labels

We can construct composite labels by either joining or meeting two labels. A composite label $L_1 \sqcup L_2$ or $L_1 \sqcap L_2$ represents nodes in a composite label structure. To employ these labels similarly to the previously defined labels, we define the owner and reader sets of the composite labels:

Definition 2.3 (Label Join)
The join operation is denoted by the operator $\sqcup$ and represents the least restrictive label that is as restrictive as either operand:

$$owners(L_1 \sqcup L_2) = owners(L_1) \cup owners(L_2)$$
$$readers(L_1 \sqcup L_2, o) = readers(L_1, o) \cap readers(L_2, o)$$

Definition 2.4 (Label Meet)
The meet operation is denoted by the operator $\sqcap$ and represents the most restrictive label that is no more restrictive than either operand:

$$owners(L_1 \sqcap L_2) = owners(L_1) \cap owners(L_2)$$
$$readers(L_1 \sqcap L_2, o) = readers(L_1, o) \cup readers(L_2, o)$$

Label composition is an integral part of DLM. Label joins allows for label evaluation of expressions, as described in Section 3.3 and provides the means for representing implicit flows via scope blocks, as described in Section 2.4. Additionally when inferring labels, the meet operation allows us to infer the most restrictive label that agrees with the preexisting labels.

2.2.2 Label comparison

As mentioned above, labels provide restrictions on how data can flow to and from slots that the labels "protect". In order to check that security policies are enforced throughout a program, we need to be able to compare how restrictive labels are. To do this we introduce the "at most as restrictive as"-relation ($\sqsubseteq$) from [2].

Definition 2.5 (Label Restrictiveness)
Label L_1 is at most as restrictive as label L_2 ($L_1 \sqsubseteq L_2$) if it (2.1) does not have more owners and (2.2) each owner does not have a smaller reader set:

$$owners(L_1) \subseteq owners(L_2) \text{ and} \tag{2.1}$$
$$\forall o \in owners(L_1).readers(L_1, o) \supseteq readers(L_2, o) \tag{2.2}$$

We can employ this definition to determine if two labels are equal: If $L_1 \sqsubseteq L_2$ and $L_2 \sqsubseteq L_1$ then we must have that $L_1 = L_2$. Additionally we note that given Definitions 2.3 and 2.5

we note that $L_1 \sqcup L_2 \sqsubseteq L_3$ is equivalent to $L_1 \sqsubseteq L_3 \wedge L_2 \sqsubseteq L_3$. This provides a simplified means for comparing composite label-structures.

2.3 Channels

Throughout a system data can enter and leave, before- and afterwhich labels cannot be enforced. The entry points for data entering and leaving can therefore be seen as crucial endpoints, where we want to do additional checking to ensure we are not unintentionally leaking. This is done through *channels*, specifically *input channels* and *output channels*. However, input channels are not special and simply apply the label of the channel to the inputted data.

Output channels
On the other hand, output channels are especially crucial in that it is an endpoint where we will intentionally "leak" data, and afterwhich we are not able to control the further flow of this data. Intuitively, an output channel represents data going outside the system, e.g. through a physical printer, console output, or web request.

Output channels therefore differ in the way that the "security" is compared. Output channels need only declare a list of readers, that represent the principals that have access to the endpoint. We then need to compare this reader list to the effective reader set of the label for the data to be outputted. Any reader declared by the output channel must be in the effective reader set for the label of the outputted data, to ensure that we do not output data to a channel where the principals aren't allowed to read it.

2.4 Implicit flows

When assigning a value to a slot, possibly from another slot, it is an explicit flow. In addition to explicit flows, it is also possible to have implicit flows throughout a program, due to conditional control structures (such as loops). For whatever assignments we would do inside a block (or within a deeper block hierarchy) we need to take the predicates of these blocks into consideration. This is done by adding the concept of *program counter labels*, denoted $\underline{pc}$. For each scope we will have an implicit $\underline{pc}$, depending on the surrounding predicates' labels.

Example 2.2 (Implicit leak)
Consider the following program (each line commented with the current $\underline{pc}$):

```
int {{a->z,y}} val = 0;    // ⊥
int {{a->y}} cond = 1;     // ⊥
if (cond) {
  val++;                   // ⊥ ⊔ {a → y}
}
return val;                // ⊥
```

In this example we would have an implicit leak of `cond`, by way of `val`. This is due to the assignment of `val` inside of the `if`-statement, as $\underline{pc}$ is increased to the join of the outer scope (⊥) and the label of the scope's predicate ($\underline{\texttt{cond}}$). For this program to pass the flow-checking tool, we would have to add a more strict policy to `val`, so that it could match the policy of `cond`.

2.5 Authority and declassification

In order to avoid mishaps by setting certain policies too lax, so that certain flows are permitted, we can temporarily set some policies to be more strict, and then only relax (*declassify*) them when we really need to. In order to do this, the concept of *authority* is introduced.

At any point during execution of a program, there will be a *true authority*, which is the maximal authority by which the program can carry out operations. Whenever we call a function, we can do this with a certain authority, corresponding to running that method with the authority of a specific principal or principals. However, in order to take advantage of this given authority, we need to explicitly claim it. This is done by using the *if acts for* construction[1], which takes as an argument one or more principals for which we want to obtain authority to carry out statements in the block. Only after calling this function will we obtain the *effective authority* to act for the principals, enabling us to carry out the statements within the block that would normally only be permissible for the principals themselves.

If the check fails, the statements block is not executed. If the check passed, the statements will be executed under the the newly established authority, e.g. for a principal p. With p in the effective authority we can perform declassification – deliberately and explicitly relaxing the security policies in which p is owner. This is done by using the *declassify* construction[2], with a slot v and a new label l as arguments, returning the value v relabeled to l. The relabeling by declassification rule (the inference rule) is defined as follows:

[1]We use the syntax: `-->? p1, p2 { /*statements*/ }` – more about this in Section 3.2.3).
[2]We use the syntax: `<|v, l|>` – more about this in Section 3.2.3).

Definition 2.6 (Relabeling by declassification)
Let P denote the set of principals in the current authority, then

$$L_A = \bigsqcup_{p \in P} \{p \rightarrow \emptyset\}$$
$$\frac{L_1 \sqsubseteq L_2 \sqcup L_A}{L_1 \text{ may be declassified to } L_2}$$

The combination of these two concepts, *if acts for* and *declassify*, is especially useful when we have sensitive inputs to a method and want to carry out our calculations without the fear of neither explicit or implicit leakage. This way we can keep our strict policies throughout the calculations of the method and only relax the label once we want to return the result.

Example 2.3 (Temporarily restricting a label)
Building on Example 2.2, we can set the label for `val` to match that of `cond`, and then only relax that label when we need to return `val`, ensuring that we have the ability to do so:

```
int {{a->y}} val = 0;
int {{a->y}} cond = 1;
if (cond) {
  val++;
}
this -->? a {
  return <|x, {{a->y,z}}|>;
}
return -1;
```

While we still leak some information about `cond`, we now do it explicitly.

Worth noting in the above example is the use of the special keyword `this` in the if-acts-for statement. This is a run-time concept that signifies from where the specified authority (here `a`) should be obtained from. When the `this` keyword is used, it refers to an implicit principal that represents the function itself. Alternatively, if instead the `caller` keyword was stated, the specified authority is compared with an explicit authority declared for each call to the function. Using our syntax, such a function call would look like:

```
foo<<<a>>>();
```

2.6 Label constraints

Using the concept of labels as described in the above sections, code can be annotated to express what type of information flow is permissible. By examining all points in code where information flows it can then be determined if these information flows are valid.

The validity of an information flow is expressed as a constraint on the form $L_1 \sqsubseteq L_2$, as described in Definition 2.5.

When information flows from one point A to another B, it is required that B is allowed to read the same information as A. Thus we can, in general, represent such an information flow as $\underline{A} \sqsubseteq \underline{B}$. This represents that access to the information known at point A is no more restrictive than access to that of point B. An example of such an information flow is the assignment of a value to a variable:

$$var := expression$$

From this simple example it is clear that information flows from the expression to the variable and thus a constraint for this example would be $\underline{expression} \sqsubseteq \underline{var}$. Though as we described in Section 2.4 information can also flow implicitly, and because of that we must include all available information in the flow constraint. The currently available information is represented as the program counter $\underline{pc}$ and must be included on the left side of the constraint, to produce:

$$\underline{expression} \sqcup \underline{pc} \sqsubseteq \underline{var}$$

This expansion of constraints captures that information assigned to *var* could contain information about the values on which the current program counter is based. Implicit flows will thus be considered similarly to any other constraints.

Extracting constraints

The above example illustrates the concept of information flow as described by a constraint. The information flow of a program can be describe by a number of constraints. Checking that each constraint is valid using Definition 2.5 will ensure that the information flow of the entire program is valid, given the declared labels. In [2, 5, 6] an informal description of which constraints a program is associated with is provided. In Section 3.3 we provide a formalized approach to extracting constraints from label declarations.

Below is a simple example of the how labels and constraints are extracted from source code. The example does not consider some of the complexities that arise from checking a complete function (or set of functions), but merely serves to provide a simple example of what checking constraints entails.

Example 2.4 (Declaration and assignment)
Two variable declarations with associated labels and an assignment operation.

```
int {{a->y, x}} val = 5;   // {a → y,x}, 5 ⊑ val
int {{a->y}} res;          // {a → y}

res = val;                 // val ⊑ res
```

As described above, the assignment operation introduces a constraint that must be checked for the information flow to be valid. Additionally the declaration of the `val` variable includes an assignment operation. Because of this we have an additional, yet trivial, constraint for the declaration. Thus we have the following constraints for the above code and know that the information flow in the code is valid.

$$\underline{5} \sqsubseteq \underline{val} \equiv \bot \sqsubseteq \{a \to y, x\}$$

$$\underline{val} \sqsubseteq \underline{res} \equiv \{a \to y, x\} \sqsubseteq \{a \to y\}$$

2.7 Inferring labels

The components of constraints are labels. Some of these labels will be constructed from the evaluation of expressions, as is the example with the assignment above. But the values of the labels themselves all stem from a declaration somewhere. Expressions merely employ the label associated with the individual slot. In Section 2.1 we described how slots and functions are associated with labels at declaration. Using these declared labels we are able to construct constraints that can then be checked to see if a programs information flow is valid, as exemplified by Example 2.4.

As constraints use the *label of* construct to represent slots, it is required that all slots be associated with a label. In other words; the programmer must determine information flow for each slot within a program. On the surface this is a great feature of DLM, as it forces the programmer to actively consider information flow throughout his application. On the downside, the programmer is also forced to spend time considering the information flow of every variable in a function. Some of these might be trivially associated with other variables and thus simple to label. But as a rule of thumb anything that is simple and trivial to do should be automated, freeing the programmer to handle more complex tasks. Having to label every slot will also increase the complexity of maintaining code, as a single change in a label might have to propagate many places.

Inference in DLM

To address the above concerns, [2] introduces the concept of label inference. Inferring labels allows a program, such as a specialized compiler, to compute appropriate labels for unlabeled slots. The inferred labels are "reverse engineered" based on the constraints of a program.

The means of inference is described informally in [2]; both in terms of the required structures and the algorithm. Because the description is only provided informally there are some uncertainties for inference. First and foremost, how constraints are extracted from code, as has already been discussed. But additionally which label declarations can be omitted and what the meaning of such an omission is.

In our approach we allow any label declaration to be omitted, and in Section 3.3 we provide a formal definition for the meaning of each omitted label as well as a definition for how constraints are extracted. In the following we formalize the data structure (Section 2.7.1) and algorithm (Section 2.7.3) described in [2] such that they can be applied to the constraints that are extracted.

2.7.1 Label types

To manage any unlabeled entities we introduce special label types that have no actual principal policies. Instead these labels function as placeholders for policies. As such, we will need to employ a special type of handling for the $\sqsubseteq$, $\sqcup$ and $\sqcap$ operations for these new labels.

Below we introduce two new label types; variable and constant labels and define how we can apply each of the above three operations to the new label types.

2.7.1.1 Variable labels

A *variable label* represents a label that should be computed through inference. When applying inference the label is associated with another label value, known as its current upper bound. This upper bound will initially have the value $\top$, representing the most restrictive label and will be relaxed as the algorithm progresses.

When the algorithm completes, the current upper bound of a variable label represents the most restrictive label that could be employed instead of the variable label. This effectively represents the inferred label.

2.7.1.2 Constant labels

A *constant label* represents a label that is not associated with a set of policies, and one that should *not* be computed through inference. Instead constant labels are used to implement label polymorphism. Specifically this will be used to provide polymorphism for function parameters and arguments. The label of a function parameter can be represented using a constant label and will be replaced by the label of the argument when the function is called.

Because of the current upper bound of variable labels, a constant label can be propagated throughout a set of constraints. This allows variables in the function and the return label of the function to contain the constant label.

2.7.1.3 Joins and meets

As the two new label types are not directly associated with policies, the join or meet of them is only represented as a composite structure. This is true regardless of the label they are being joined or met with.

Our interest in these labels is only temporary. Either to determine the most restrictive upper bound or for the purpose of label polymorphism. Because of this we are content using a composite representation. When replacing a variable or constant label with an actual set of policies the composite structure will be removed by virtue of Definitions 2.3 and 2.4.

2.7.1.4 Label restriction

For the inference algorithm to function using the new label types it is required that we can determine if one label is no more restrictive than another label. Thus we must determine how to handle this comparison for both variable and constant labels. Additionally we must determine the comparison for both join and meet composite labels.

Firstly we note that a definition for the comparison is not required for variable labels. In the algorithm we employ the current upper bound of variable labels for the label comparison. Because of this we will never directly compare variable labels with other labels.

For any constant label L_c we define that $L_c \not\sqsubseteq L$ and $L \not\sqsubseteq L_c$ for any label L, with the natural exception of $L_c = L$. For label joins we merely seek to identify those cases where we are sure that the comparison evaluates to true, and let the remaining evaluate to false. This could result in some false negatives. With some insight into the algorithm, as presented in Section 2.7.3, we note that the effect of false negatives is that the labels inferred could be more relaxed than possibly needed. The algorithm does however ensure that all information flow constraints are still valid.

We provide the following definitions given the above description:

- $L_1 \sqsubseteq L_3 \wedge L_2 \sqsubseteq L_3 \Rightarrow L_1 \sqcup L_2 \sqsubseteq L_3$ (As described in Section 2.2.2)
- $L_1 \sqsubseteq L_2 \vee L_1 \sqsubseteq L_3 \Rightarrow L_1 \sqsubseteq L_2 \sqcup L_3$
- $L_1 \sqsubseteq L_3 \vee L_2 \sqsubseteq L_3 \Rightarrow L_1 \sqcap L_2 \sqsubseteq L_3$
- $L_1 \sqsubseteq L_2 \wedge L_1 \sqsubseteq L_3 \Rightarrow L_1 \sqsubseteq L_2 \sqcap L_3$

These definitions for determining how restrictive labels are will be used for the inference algorithm.

2.7.2 Output channels

As previously described, the way output channels are checked is by comparing the reader set of the output with the effective reader set of the outputted value(s). This means that

for an output channel with readers R_c and some labeled output values V, we must have that

$$\forall v \in V. R_c \subseteq readers(\underline{v})$$

in order for the information flow to be valid.

We will instead opt for a different method, so that we can leave the checking of output channels to our inference algorithm, similar to any other inferrable construct. This is so that we can use inference for output channels as well, which is not possible when using the "simple" comparison of reader sets.

Definition 2.7 (Output channel constraints)
Let $\{p_1, p_2, \ldots, p_k\}$ be the set of all principals in a system. Then for an output channel with readers R_c and labeled input values V, we define an associated label:

$$L_c = \{p_1 \rightarrow R_c; p_2 \rightarrow R_c, \ldots, p_k \rightarrow R_c\}$$

That is, a label owned by all the principals in the system, where each owner allows all principals in R_c to read. With this label, we give the following definition for the output channel:

$$\forall v \in V. \underline{v} \sqsubseteq L_c$$

This approach will allow the inference algorithm to check the reader set of an output channel, effectively providing inference for any values that are passed to an output channel. Below we demonstrate that this approach is equivalent to comparing reader sets, by showing that

$$\underline{v} \sqsubseteq L_c \Rightarrow R_c \subseteq readers(\underline{v}), \text{ for any } v \in V$$

When evaluating the above constraint we note that Equation (2.1) in Definition 2.5 will always be true, as $owners(L_c)$ is the set of all principals. We will then apply Equation (2.2) to the same constraint to get

$$\forall o \in owners(\underline{v}). readers(\underline{v}, o) \supseteq readers(L_c, o)$$

From Definition 2.7 we know that $readers(L_c, o) = R_c$, for any principal o in the system. Finally we see from Definition 2.2 that $readers(\underline{v}, o) \supseteq readers(\underline{v})$, for any principal o in the system. Thus we know that

$$readers(\underline{v}, o) \supseteq readers(\underline{v}) \supseteq R_c, \text{ for any } o \in owners(\underline{v})$$

2.7.3 Inference algorithm

Using the established extensions of labels we present the inference algorithm (see Algorithm 1). In the following we provide an informal description of the algorithm to aid the reader.

Note that the input for the algorithm is a set of constraints. In Section 3.3 we detail how to extract all constraints from source code, though the input for the algorithm can be *any* set of constraints (see Section 2.6). The only exception from this rule is that $\sqcap$ labels are not allowed on the left side of a constraint. Why this is the case will become apparent as we walk through the algorithm. The algorithm does not return a result, but instead works through the side effect of updating the current upper bound for variable labels.

Initialization

As described in Section 2.7.1, the initial current upper bound of a variable label must be $\top$. Thus for all constraints with a variable label on the left hand side, we set the current upper bound of that label. We use $\text{cub}(L)$ to denote the current upper bound of a variable label L during execution of the algorithm.

With this we define another function that takes a label as input and returns a new label where variables are replaced by their current upper bound. This function will be applied to labels in the algorithm before performing comparisons of those labels.

$$\text{novar}(L) = \begin{cases} \text{cub}(L) & \text{if } L \text{ is a variable label} \\ \text{novar}(L_3) \sqcup \text{novar}(L_4) & \text{if } L_1 = L_3 \sqcup L_4 \\ \text{novar}(L_3) \sqcap \text{novar}(L_4) & \text{if } L_1 = L_3 \sqcap L_4 \\ L & \text{otherwise} \end{cases}$$

It could be the case that a constraint is on the form $L_1 \sqcup L_2 \sqsubseteq L_3$. If L_1 or L_2 is a variable label in such a constraint their current upper bound would not be set. Thus we would want to translate the constraint such that there are no joins on the left hand side of it. In Section 2.2.2 we noted that $L_1 \sqcup L_2 \sqsubseteq L_3$ and $L_1 \sqsubseteq L_3 \wedge L_2 \sqsubseteq L_3$ are equivalent.

With this we define a function that takes a constraint as input and produces a set of constraints where the left hand side has no joined labels. This function is applied to all constraints before setting the current upper bound of the variable labels.

$$\text{unjoin}(''L_1 \sqsubseteq L_2'') = \begin{cases} \text{unjoin}(L_3) \cup \text{unjoin}(L_4) & \text{if } L_1 = L_3 \sqcup L_4 \\ \{''L_1 \sqsubseteq L_2''\} & \text{otherwise} \end{cases}$$

Because meet labels are not allowed in label declarations, we can be sure that the left hand side of all constraints is either a set of policies, a variable label or a constant label.

Computing labels

Having taking the necessary initializing steps the algorithm checks that each constraint in the constraints set is valid. If all constraints are valid we know that the information flow of the program is valid and the algorithm can terminate. Thus, if we have explicitly stated all labels, the algorithm simply performs a check of weather the information flow is valid.

If a constraint is not valid one of two things will happen, based on the type of label on the left hand side of the constraint:

- If the label is a variable, we update its current upper bound to *include* the right hand side of the constraint. This is achieved by meeting the two labels.
- If the label is not a variable, we can do nothing to relax it and the inference will fail. This is the case we will end up in when considering a program with invalid information flow and only explicit or constant labels.

Overall the idea of the algorithm is to initialize all variable labels to $\top$ and then relax them until all constraints are valid. It might be tempting to make a label stricter such that a specific constraint is valid. This is however not an option, as labels are only relaxed exactly enough that constraints are valid. Making a label stricter would thus break a previously checked label.

```
Data: A set Q of constraints "L1 ⊑ L2"
1  foreach "L1 ⊑ L2" ∈ Q do
2  |  let Q′ = Q \ {"L1 ⊑ L2"}
3  |  Q := Q′ ∪ unjoin("L1 ⊑ L2")
4  foreach "L1 ⊑ L2" ∈ Q do
5  |  if L1 is a variable label then
6  |  |  cub(L1) := ⊤
7  checked:= false
8  while ¬checked do
9  |  checked:= true
10 |  foreach "L1 ⊑ L2" ∈ Q do
11 |  |  if novar(L1) ⋢ novar(L2) then
12 |  |  |  if L1 is a variable label then
13 |  |  |  |  cub(L1) := cub(L1) ⊓ novar(L2)
14 |  |  |  |  checked:= false
15 |  |  |  else
16 |  |  |  |  ERROR
```

Algorithm 1: Label inference from constraint set

Chapter 3

Decentralized Label Model in C

In this chapter we will present the C Timed Information Flow language, hereafter referred to simply as CTIF. This chapter will go into details about applying labels to a C program. The following chapter will go into the details of time policies.

CTIF is based upon C99 and will extend the syntax of C with the concept of labels, as discussed in the previous chapter. This fact gives us the following two properties:

1. A C99 program[1] is compilable with the CTIF compiler.
2. An unlabeled CTIF program can be checked by a C99 compiler.

The first property allows for incrementally adding labels to old source code. This becomes even more palpable if the CTIF compiler is able to infer labels. The second property frees the CTIF compiler from handling many of the same things that instead can be handled by already existing C99 compilers. Not having to work out every detail of a complete C compiler frees us to focus on the extension itself.

The rest of this chapter will be organized as follows. Firstly, we will present the scope under which we have limited our implementation. Then we will informally present CTIF. Lastly we will present the specifics of extracting constraints, to be used for inferring labels, as was described in Sections 2.6 and 2.7.

3.1 Scope

We have tried to include as many C constructs as possible, in order to properly ascertain the usefulness of CTIF. However, certain aspects of C have been either simplified or completely left out.

[1] Limited under the scope presented in Section 3.1.

In the following is a crude list of the constructs that are supported by CTIF. Additionally we provide elaborate descriptions for some of these constructs, as we make certain observations about their use.

- Variables
- Simple types (int, bool, char, string)
- Structs
- Pointers, including basic pointer operators
- Function prototypes, declarations, and calls
- Boolean and arithmetic operators
- *if*- and *while* control statements
- The ternary operator
- External libraries

Structs and arrays
It is possible to define structs, so that the defined type can be used similarly to any other type. Both structs and arrays can be declared, arrays by using pointer-notation. For this we allow for index and element expressions, with a simple implementation, which only takes into account the label of the struct/array variables.

We have chosen to leave out labeling of individual struct field and array elements. In [2] labels can be applied to structs by labeling the variable. Additionally, fields can be labeled, resulting in a special rule for the evaluation of accessing struct fields, which is the label join of the struct label and the accessed field's label. We, however, find it somewhat illogical to label a type, since different usage of a certain type may call for different labeling.

Pointers
Similar to both [3] and [4] we allow for pointer declarations and for the two pointer operations dereference and address-of. There is no restriction on the use of pointer arithmetics in CTIF. Pointer arithmetics is deemed too big of a subject to handle within the scope of this report, and without proper handling it is potentially highly error-prone.[2]

3.2 Informal description

CTIF extends the C language with the ability to declare labels for values and functions. Labels are declared between the type and the identifier of a variable, or between the return type and identifier of a function. The syntax for labels consists of 5 components:

[2]https://www.securecoding.cert.org/confluence/display/c/EXP08-C.+Ensure+pointer+arithmetic+is+used+correctly

- delimiters `{{ }}`
- join operator ;
- identifiers – e.g. `owner`, `reader`, or `variable`
- policy operator `->` – e.g. `{{owner->reader1, reader2}}`
- special values _ (underscore, for bottom label) and ^ (caret, for top label)

Besides the actual labels we also have the following related language constructs. The theory for these was described in Sections 2.3 and 2.5. In the sections below we describe how these constructs are handled in CTIF.

- output channel `p1, p2 <- foo();`
- if acts for `-->? p1, p2 { /* statements */ }`
- declassify `<|v, l|>`
- function call `bar<<<p1, p2>>>();`

The following sections will describe how labels are used in CTIF. The descriptions will follow the password checker example from Section 1.4.2. The fully labeled source can be seen in Listing 3.1, on page 32. Note that we make use of the `principal` declaration in the example to simulate the actual obtaining of system principals.

3.2.1 Function declarations

The first thing we would want to label in a program is what can be seen as our potentially sensitive endpoints; the functions that either obtain data from outside or send data to outside the program – our channels. In CTIF we have no distinction between input channels and any other function which has a return value. However, output channels are distinguishable by having the `<-` construct, preceded by one or more principals (readers). Output channel declarations are initially checked as with any other function.

Labeling the example

For `check_password` we have 3 auxiliary functions, as well as the `check_password` function itself, which we would label as so:

```
11 user_info {{u->u}} get_login();
12 user_info {{pc->}} *get_users();
13 u <- void send_response(bool is_match);
14
15 bool {{u->u}} check_password(char {{u->u}} *username, char {{u->u}} *
       password) {
```

Here we have two principals: user (u) and password checker (pc). The user provides some login information, to which he is the owner. The password checker is a trusted principal which has access to the user database. The final output is sent to an output channel which is readable only by the user.

`check_password` returns the comparison result, which is owned by and readable by the user only. Here we also need to add labels to the parameters, otherwise we would not be able to statically check the function, as the actual labels for the parameters could differ between different calls to the function, but by supplying the labels we know that the policies will have to hold for all usage. When supplied with a label, parameters are treated as any other variable when evaluating a function declaration. However, if no label is supplied for a parameter, that parameter is recognized as labeled with a constant label. This is used to provide label inference for function parameters, as we shall see in

3.2.2 Variable declarations

The labeling of variable declarations is done mainly to oblige the return values of functions and input, given by either parameters or auxiliary function calls. This is also why most of these labels are easier to infer, and can therefore often be left out, as we shall see in Section 3.2.4. For now we will explicitly include all labels, to properly demonstrate the information flow.

For `check_password` we have the following labeling of variables:

```
16  int {{pc->}} user_count = 100;
17  user_info {{pc->}} *users = get_users();
18  int {{pc->}} i = 0;
19  bool {{u->u;pc->}} match = false;
```

The labeling applied here is done so that it obliges the different inputs to the method. The most noteworthy here is the labeling of `match`, as it through its assignment within both the `while` and `if` blocks will implicitly obtain knowledge of all other labeled values. It is therefore labeled with the join of all those labels: `{{u->u;pc->}}` so that it matches them all.

3.2.3 If-acts-for and declassification

Going into `check_password` we have data that is privy to the user as well as data that is not. So before we can return the result to the user, we need to explicitly declare that in this particular case it is an intentional leak of the password checker's data (the collection of all user data).

This is done with the declassification operator `<| |>` which has two operands, an expression to be declassified and the new label. For `check_password` we need to declassify and return `match`:

```
28   this -->? pc {
29     return <|match, {{u->u}}|>;
30   }
```

As can be seen we also need to obtain the proper authority before such a declassification is possible. This is done by using the *if-acts-for* statement, identified by the `-->?` operator, which is preceded by type of authority and followed by principals to acquire authority for. During compile-time checking we do not differ between the special keywords `this` and `caller` (see Section 2.5). After obtaining the proper effective authority, we can declassify the label of `match`.

It is possible to leave out the explicit label of the declassification, as we will see in the following section.

3.2.4 Inference

As we have argued repeatedly throughout this report, having to declare labels for every value or function can be time-consuming and can cause clutter in a program. Therefore we can instead rely on inference so that some labels may be omitted. In [2] it is only described what leaving out labels for variables and parameters mean, and not functions. Therefore we have reasoned about the implications of leaving out the function label and come up with a reasonable solution, so that we can leave out those labels and still expect proper results.

As we have seen, explicit labels can be applied to the three types of declarations: function, parameter, and variable, as well as in the declassification expression. Here we will explain what it means to leave out any of those labels in a program. An overview of the different rules that apply are given here:

Construct	**Default label**
Variable	Variable label
Declassification	Variable label
Function	The join of all parameter labels
Parameter	Constant label

Table 3.1: Default labels for unlabeled constructs

As described in Section 2.7.1, we have the concepts of *variable labels* and *constant labels*. Variable labels are the main enablers of the inference algorithm, as it allows for implicitly labeling certain values that in themselves do not need a label, but are instead related to other values that do. Constant labels, in a similar fashion, allow for the abstraction of only applying labels where it is critical, allowing for some implicity when defining the security of functions. We have updated the password checker example (see Listing 3.2),

removing all explicitly declared labels that instead can be inferred. Similarly, the calculate bill example has been labeled as well (see Listing 3.3).

Variable

As explained above, by removing the explicitly declared label for a variable, we instead rely on the inference algorithm to satisfy any constraints. Since we have explicitly declared the labels for all our channels and `check_password`, we do not need to explicitly label all our variables as well. Because of this we are able to leave out all label declarations for variables.

Declassification

Similar to variables, we can leave out the explicitly declared label for the declassification expression. This creates a variable label for that expression, such that we can let inference determine which declassification will satisfy our remaining constraints.

Functions and parameters

If no explicitly declared label is found for a function declaration, the label defaults to a join of all parameter labels ($\perp$ if the function has no parameters). This is based on an assumption about the minimum security needed for a function, in that whatever a function returns can be related to at most whatever arguments are given. If this is not the case the label of the source where the function retrieves data is included in the functions information flow and will most likely result in an invalid constraint.

The use of implicit function return labels is useful for declaring auxiliary functions, without having to explicitly declare the security implications of such a function. For our password checker we could declare an auxiliary function `strcicmp`, for doing a case-insensitive check of usernames, with the following prototype:

```
bool strcicmp(char *a, char *b);
```

`strcicmp` would then have the label $\underline{a} \sqcup \underline{b}$, per our default label for function declarations, and any evaluation of calls to `strcicmp` would result in a label which is the join of all its arguments. This is the default behaviour of library functions, which can be overridden by declaring a labeled prototype with the corresponding library function signature.

Extending on the concept of referencing parameter labels in function return labels, we can choose which parameters to reference. If we take the function `stricmp` and want to extend it with a maximum amount of characters to compare before failing, we could have that its signature was:

```
bool {{a;b}} strcicmp(char *a, char *b, int max_length);
```

Similar to the previous example, we have that any calls to `strcicmp` will be evaluated to the join of its arguments, but this time only the arguments `a` and `b`, thus any security applied to `max_length` will not affect the label of the return value.

Alternatively, we could have opted to label the latter definition as below:

```
bool strcicmp(char *a, char *b, int {{_}} max_length);
```

Even though we have a joined label of all parameters, `max_length` would be omitted by any evaluation of labels. The label is effectively the same, namely $\underline{\texttt{a}} \sqcup \underline{\texttt{b}}$.

3.3 Constraint extraction

In this section we will give a formal definition of how constraints are extracted from a CTIF program. This formal definition will be based on denotational semantics, and will therefore utilize the concepts thereof. We will define our syntactic and semantic setup, which will be used to define how constraints are extracted from a CTIF program. Following the setup, we will present semantic equations, which describe how constraints are extracted from the individual program constructs.

3.3.1 Syntax

Below follows an abstract syntax that describes the structure of our grammar. The grammar is made to replicate the structure of the C programming language, extended with the DLM specific language constructs described in Section 3.2. To do this we have created an unambiguous parser in which certain details of are abstracted away. This parser will, for instance, manage operator precedence when reading input, but represent it using the abstraction below.

Table 3.2 contains our syntactic domains and abstract prodution rules, and will serve as a reference-point for the following sections. A few details might be of special interest:

Simplified arithmetics As the semantics, described in this section, only concerns how data *flows* and not its value on execution we can simplify the possible binary and unary operations into two rules. Doing so provides for a less cluttered syntax and a clearer focus on the label semantics that are expressed.

Pointer declaration The `*` used to indicate a pointer type, can be optionally declared for any variable or parameter. Whether a variable is a pointer or not has no meaning for the evaluation of labels, so the handling of pointer type checking is completely left to the C compiler. Therefore it will not included in the abstract syntax.

Struct fields We make use of the special variable identifier x_f for struct field access. It has no meaning in evaluation of the program, as will be apparent in the rules for the struct field expression. The actual checking of struct fields and the matching struct definition is handled by the C compiler.

```
1  #include <stdbool.h>
2  #include <string.h>
3
4  principal u, pc;
5
6  typedef struct user_info {
7    char username[20];
8    char password[20];
9  } user_info;
10
11 user_info {{u->u}} get_login();
12 user_info {{pc->}} *get_users();
13 u <- void send_response(bool is_match);
14
15 bool {{u->u}} check_password(char {{u->u}} *username, char {{u->u}} *
       password) {
16   int {{pc->}} user_count = 100;
17   user_info {{pc->}} *users = get_users();
18   int {{pc->}} i = 0;
19   bool {{u->u;pc->}} match = false;
20
21   while (i < user_count) {
22     if (!strcmp(users[i].username, username) && !strcmp(users[i].
       password, password)) {
23       match = true;
24     }
25     i = i + 1;
26   }
27
28   this -->? pc {
29     return <|match, {{u->u}}|>;
30   }
31 }
32
33 int main(int argc, char **argv) {
34   user_info {{u->u}} login = get_login();
35   bool {{u->u}} is_match = check_password(login.username, login.
       password);
36   send_response(is_match);
37 }
```

Listing 3.1: Labelled password checker example

```
#include <stdbool.h>
#include <string.h>

principal u, pc;

typedef struct user_info {
  char username[20];
  char password[20];
} user_info;

user_info {{u->u}} get_login();
user_info {{pc->}} *get_users();
u <- void send_response(bool is_match);

bool check_password(char *username, char *password) {
  int user_count = 100;
  user_info *users = get_users();
  int i = 0;
  bool match = false;

  while (i < user_count) {
    if (!strcmp(users[i].username, username) && !strcmp(users[i].
    password, password)) {
      match = true;
    }
    i = i + 1;
  }

  this -->? pc {
    return <|match|>;
  }
}

int main(int argc, char **argv) {
  user_info login = get_login();
  bool is_match = check_password(login.username, login.password);
  send_response(is_match);
}
```

Listing 3.2: Labeled password checker example – with default labels

```
principal u, ec, s;

typedef struct usage {
  int start_time;
  int usage_in_Wh;
} usage;

typedef struct price {
  int start_time;
  int price_in_cents;
} price;

usage {{u->u, ec}} *get_latest_usage();
price {{_}} *get_latest_prices();
u <- void send_to_consumer(int bill_total);
ec <- void send_to_electrical_company(int bill_total);

int {{u->u, ec}} calculate_bill() {
  int usage_count = 100;
  int prices_count = 100;
  usage *latest_usage = get_latest_usage();
  price *latest_prices = get_latest_prices();
  int result = 0;

  int i = 0;
  int j = 0;
  while (i < usage_count) {
    while ((j < prices_count-1) && (latest_prices[j+1].start_time <= latest_usage[i].start_time)) {
      j = j + 1;
    }
    result = result + latest_usage[i].usage_in_Wh * latest_prices[j].price_in_cents;
    i = i + 1;
  }
  this -->? s {
    return <|result|>;
  }
}

int main(int argc, char **argv) {
  int bill_total = calculate_bill();
  send_to_consumer(bill_total);
  send_to_electrical_company(bill_total);
}
```

Listing 3.3: Labeled smart meter bill calculation example – with default labels

$R \in$ **Prog** – Programs
$D \in$ **Dec** – Declarations
$D_F \in$ **DecF** – Function declarations
$f \in$ **Fun** – Functions
$D_V \in$ **DecV** – Variable declarations
$x \in$ **Var** – Variables
$p \in$ **Prin** – Principals
$S \in$ **Stm** – Statements
$E \in$ **Exp** – Expressions
$L \in$ **Lbl** – Label declaration
$P \in$ **Pol** – Policy declaration
$op_b \in \{$ `+`, `-`, `*`, `/`, `%`, `||`, `&&`, `<`, `>`, `==`, `<=`, `>=` $\}$
$op_u \in \{$ `!`, `-`, `*`, `&` $\}$
$k \in \{$ `true`, `false` $\} \cup$ **Num** $\cup$ **Chr** $\cup$ **Str** – Boolean, integer, char, and string literals
$t \in$ **Types** – C types (including defined structs)

$$
\begin{aligned}
R &::= D \\
D &::= D_F \mid D_V \mid D_1\ D_2 \\
D_F &::= t_f\ L_f\ f\ \texttt{(}\ t_1\ L_1\ x_1\ \texttt{,}\ t_2\ L_2\ x_2\ \texttt{,}\ \ldots\ \texttt{,}\ t_n\ L_n\ x_n\ \texttt{)}\ S \\
&\mid p_1\ \texttt{,}\ p_2\ \texttt{,}\ \ldots\ \texttt{,}\ p_k\ \texttt{<-}\ t_f\ L_f\ f\ \texttt{(}\ t_1\ L_1\ x_1\ \texttt{,}\ t_2\ L_2\ x_2\ \texttt{,}\ \ldots\ \texttt{,}\ t_n\ L_n\ x_n\ \texttt{)}\ S \\
D_V &::= t\ L\ x\ \texttt{=}\ E \mid t\ L\ x \\
S &::= E \mid S_1\ S_2 \mid \varepsilon \\
&\mid D_V \\
&\mid \texttt{while (}\ E\ \texttt{)}\ S \mid \texttt{if (}\ E\ \texttt{)}\ S \mid \texttt{if (}\ E\ \texttt{)}\ S_1\ \texttt{else}\ S_2 \\
&\mid x\ \texttt{=}\ E \\
&\mid \texttt{return}\ E \mid \texttt{return} \\
&\mid \texttt{this -->?}\ p_1\ \texttt{,}\ p_2\ \texttt{,}\ \ldots\ \texttt{,}\ p_k\ S \mid \texttt{caller -->?}\ p_1\ \texttt{,}\ p_2\ \texttt{,}\ \ldots\ \texttt{,}\ p_k\ S \\
E &::= x \mid k \mid E_1\ op_b\ E_2 \mid op_u\ E \mid \texttt{(}\ E\ \texttt{)} \mid E\texttt{.}x_f \mid E\texttt{->}x_f \mid x\texttt{[}E\texttt{]} \\
&\mid \texttt{<|}\ E\ \texttt{,}\ L\ \texttt{|>} \mid \texttt{<|}\ E\ \texttt{|>} \\
&\mid f\ \texttt{(}\ E_1\ \texttt{,}\ E_2\ \texttt{,}\ \ldots\ \texttt{,}\ E_n\ \texttt{)} \\
&\mid f\ \texttt{<<<}\ p_1\ \texttt{,}\ p_2\ \texttt{,}\ \ldots\ \texttt{,}\ p_k\ \texttt{>>>}\ \texttt{(}\ E_1\ \texttt{,}\ E_2\ \texttt{,}\ \ldots\ \texttt{,}\ E_n\ \texttt{)} \\
L &::= \texttt{\{\{}\ P\ \texttt{\}\}} \mid \varepsilon \\
P &::= x \mid p_0\ \texttt{->}\ p_1\ \texttt{,}\ p_2\ \texttt{,}\ \ldots\ \texttt{,}\ p_k \mid P_1\ \texttt{;}\ P_2 \mid \texttt{_} \mid \texttt{\^{}}
\end{aligned}
$$

Table 3.2: Syntactic domains and abstract production rules

Empty labels No value (represented by ε) can be provided for any label declaration. When omitting a label the semantic rules specify a *default* value for the label (see Section 3.2.4). This is the initial step of label inference and allows unlabeled C to be recognized and evaluated using these rules.

3.3.2 Semantic setup

This section will describe the semantic constituents which are used to translate a CTIF program into the semantical world. These constituents are based on the concepts: *semantic domains*, *semantic functions*, and *semantic equations* from [7, Chapter 9]. The semantic domains and semantic functions will be described in the next two sections. For readability purposes, the semantic equations will be spread out over the following sections.

3.3.2.1 Semantic domains

The semantic domains of CTIF are primarily used to maintain state, and to give the final result of the constraint extraction: the constraints themselves as well as the different label types which are used to form the constraints. Note that we have no need for any numerals or type enumerations, since we are only interested in checking labels and forming constraints.

The following is an overview of these semantic domains (Here $\mathcal{P}$ denotes the power set):

$$\begin{aligned}
\mathbf{LV} &= v(\mathbf{Var}) \cup c(\mathbf{Var}) \cup p(\mathbb{L}) \cup j(\mathbf{LV} \times \mathbf{LV}) \\
\mathbf{Cstr} &= \mathcal{P}(\mathbf{LV} \times \mathbf{LV}) \\
\mathbf{EnvF} &= \mathbf{Fun} \cup \{\phi\} \rightharpoonup \mathbf{LV} \times \mathcal{P}(\mathbf{LV} \times \mathbf{Var}) \times \mathcal{P}(\mathbb{P}) \\
\mathbf{EnvL} &= \mathbf{Var} \cup \{\alpha, \beta\} \rightharpoonup \mathbf{LV}
\end{aligned}$$

Table 3.3: Semantic domains

$\mathbb{L}$ denotes the set of all labels.

$\mathbb{P}$ denotes the set of all principals. Principals are expected to be extracted from some system information and not defined in the source code itself. Instead we make use of a function *principal*, which represents the lookup of principal names in some system:

$$principal = \mathbf{Prin} \rightarrow \mathbb{P}$$

We will also make use of the special set P^*, where $P^* \subseteq \mathbb{P}$, representing all principals of the system.

LV is a set of *label values*, not to be confused with actual labels. In order to differ between the different types of labels (as described in Section 2.7.1), we utilize the concept of *tagged values*. Each element in **LV** is then on the form $t(v)$, where t is the type of label and v is the value, which differs depending on the type of label.

To further increase readability when using *join labels*, we will introduce another notation using quotes to signify that the value is the constituents and not the evaluated expression. Then we have that:

$$''\, lv_1 \sqcup lv_2\, '' \equiv j(lv_1, lv_2)$$

Cstr is a set of constraint tuples, where any $(lv_1, lv_2) \in$ **Cstr** can be seen as representing a constraint: $''\ lv_1 \sqsubseteq lv_2\ ''$. Similar to the join label above, we use the quoted notation to represent one of the aforementioned tuple, such that:

$$''\, lv_1 \sqsubseteq lv_2\, '' \equiv (lv_1, lv_2)$$

EnvL is the label environment, which maps variable identifiers, and the two special identifiers α (authority, see Section 2.5) and β (program counter, see Section 2.4), to label values. The special values are propagated throughout a program, as they will be updated as the scope changes.

EnvF is the function environment, which maps function identifiers, and the special identifier ϕ (current function), to a tuple containing the label value of the function along with all its arguments and output readers. Function identifiers are looked up when evaluating function calls. ϕ is used to look up the function currently being evaluated. This is used when evaluating `return` statements, as we have no means of obtaining the function name for lookup.

3.3.2.2 Semantic functions

The semantic functions describe how the syntactic domains (see Table 3.2) will be transformed into the semantic domains (see Table 3.3). We have a function for each domain, that will extract constraints, or derive the labels used in forming constraints. The formal definitions of the semantic functions can be seen in Table 3.4.

$$\mathcal{R} : \textbf{Prog} \rightarrow \textbf{Cstr}$$
$$\mathcal{D} : \textbf{Dec} \rightarrow (\textbf{EnvL} \times \textbf{EnvF} \rightarrow \textbf{Cstr} \times \textbf{EnvL} \times \textbf{EnvF})$$
$$\mathcal{F} : \textbf{DecF} \rightarrow (\textbf{EnvL} \times \textbf{EnvF} \rightarrow \textbf{Cstr} \times \textbf{EnvF})$$
$$\mathcal{V} : \textbf{DecV} \rightarrow (\textbf{EnvL} \times \textbf{EnvF} \rightarrow \textbf{Cstr} \times \textbf{EnvL})$$
$$\mathcal{S} : \textbf{Stm} \rightarrow (\textbf{EnvL} \times \textbf{EnvF} \rightarrow \textbf{Cstr} \times \textbf{EnvL})$$
$$\mathcal{E} : \textbf{Exp} \rightarrow (\textbf{EnvL} \times \textbf{EnvF} \rightarrow \textbf{Cstr} \times \textbf{LV})$$
$$\mathcal{L} : (\textbf{Lbl} \cup \textbf{Pol}) \rightarrow (\textbf{EnvL} \rightarrow \textbf{LV} \cup \{\varepsilon\})$$

Table 3.4: Semantic functions

The “entry point” of a CTIF program is simply one or more declarations, and the result of evaluating such a program is always a set of constraints. The transformation of declarations will result in a set of constraints, as well as a changed environment: the function environment for function declarations and the label environment for variable declarations. Statements are where most constraints will originate from, as this is where we have variable declarations, assignments, and control structures. Expressions will also generate constraints, due to declassifications. Due to the simplicity of the label syntactic domain, its rules have been merged with those of policy. Unlike the other semantic functions, labels and policies are evaluated to a label value so that they can be used in the forming of constraints.

3.3.3 Program and declarations

The rules for program and declarations are simple, as their main purpose is providing an entry point, as well as the ability to have variable and function declarations appear in the global scope, and in arbitrary order. $emptyenv_F$ and $emptyenv_L$ are the initial, empty, environments.

$$\mathcal{R}[\![D]\!] = cstr$$
$$\quad \text{where } \mathcal{D}[\![D]\!]\ emptyenv_L\ emptyenv_F = (cstr, env_L, env_F)$$

$$\mathcal{D}[\![D_F]\!]\ env_L\ env_F = (cstr, env_L, env_{F2})$$
$$\quad \text{where } \mathcal{F}[\![D_F]\!]\ env_L\ env_F = (cstr, env_{F2})$$

$$\mathcal{D}[\![D_V]\!]\ env_L\ env_F = (cstr, env_{L2}, env_F)$$
$$\quad \text{where } \mathcal{V}[\![D_V]\!]\ env_L\ env_F = (cstr, env_{L2})$$

$$\mathcal{D}[\![D_1\ D_2]\!]\ env_L\ env_F = (cstr \cup cstr_2, env_{L3}, env_{F3})$$
$$\quad \text{where } \mathcal{D}[\![D_2]\!]\ env_{L2}\ env_{F2} = (cstr_2, env_{L3}, env_{F3})$$
$$\quad \text{and } \mathcal{D}[\![D_1]\!]\ env_L\ env_F = (cstr, env_{L2}, env_{F2})$$

Table 3.5: Semantic equations for program and declarations

3.3.4 Label and policy

The rules for labels and policies consist of a simple transformation of the syntactical labels into their semantic equivalents. Worth noting here is that there is no "empty" label value for undeclared labels. As the label value for an undeclared label depends on its context, ε is simply passed through on evaluation.

$$\mathcal{L}[\![\varepsilon]\!]\ env_L = \varepsilon$$

$$\mathcal{L}[\![\ \texttt{\{\{}\ pol\ \texttt{\}\}}\ ]\!]\ env_L = \mathcal{L}[\![pol]\!]\ env_L$$

$$\mathcal{L}[\![pol_1\ \texttt{;}\ pol_2]\!]\ env_L = j(lv_1, lv_2)$$
where $\mathcal{L}[\![pol_1]\!]\ env_L = lv_1$
and $\mathcal{L}[\![pol_2]\!]\ env_L = lv_2$

$$\mathcal{L}[\![x]\!]\ env_L = env_L\ x$$

$$\mathcal{L}[\![p_0\ \texttt{->}\ p_1\ \texttt{,}\ p_2\ \texttt{,}\ \ldots\ \texttt{,}\ p_k]\!]\ env_L = p(\{p'_0 \rightarrow p'_1, p'_2, \ldots, p'_k\})$$
where $p'_i = principal(p_i)$, for $0 \leq i \leq k$

$$\mathcal{L}[\![\ \texttt{_}\ ]\!]\ env_L = p(\bot)$$

$$\mathcal{L}[\![\ \texttt{\^{}}\ ]\!]\ env_L = p(\top)$$

Table 3.6: Semantic equations for label and policy

3.3.5 Function declarations

Function declaration (along with function call) can easily be considered the most complex rule(s).

For the return label lv_f for a function declaration it is possible to reference the parameters of that function. To achieve this the env_{L2} environment is constructed from the parameters.

When evaluating the statements of the function, a reference to the return label of said function is required; this is included in env_{F2}, along with a label reference for the declared function itself. The latter allows for recursion.

For simplicity we have included only one production, despite the first part declaring readers p_1 , p_2 , ... , p_k <- begin optional. Function declarations are handled nearly the same, with the exception of the added reader for output channels. When no readers have been declared, R_o obtained from the condition $R_o = \{p'_i | p'_i = principal p_i\}$ is simply the empty set.

$$\mathcal{F}[\![p_1\ ,\ p_2\ ,\ \ldots\ ,\ p_k \texttt{<-}\ t_f\ L_f\ f\ (\ t_1\ L_1\ x_1\ ,\ t_2\ L_2\ x_2\ ,\ \ldots\ ,\ t_n\ L_n\ x_n\ )\ S]\!]\ env_L\ env_F = (cstr, env_{F2})$$

$$\text{where } \mathcal{S}[\![S]\!]\ env_{L2}\ env_{F2} = (cstr, env_{L3})$$

$$\text{and } env_{L2} = env_L[x_1 \mapsto lv'_1, x_2 \mapsto lv'_2, \ldots, x_n \mapsto lv'_n]$$

$$\text{and } env_{F2} = env_F[\phi \mapsto (lv'_f, \emptyset, R_o), f \mapsto (lv'_f, \{(lv'_1, x_1), (lv'_2, x_2), \ldots, (lv'_n, x_n)\}, R_o)]$$

$$\text{and } lv'_f = \begin{cases} \bigsqcup_{i=1}^{n} lv'_i & \text{if } lv_f = \varepsilon \\ lv_f & \text{otherwise} \end{cases}$$

$$\text{and } \mathcal{L}[\![L_f]\!]\ env_L = lv_f$$

$$\text{and } lv'_i = \begin{cases} c(x_i) & \text{if } lv_i = \varepsilon \\ lv_i & otherwise \end{cases}$$

$$\text{and } \mathcal{L}[\![L_i]\!]\ env_L = lv_i \text{ for all } 0 \leq i \leq n$$

$$\text{and } R_o = \{p'_i | p'_i = principal\, p_i\}$$

Table 3.7: Semantic equation for function declaration

3.3.6 Variable declaration

New labels are introduced when declaring variables. A variable declaration without initialization uses $\perp$ as the label value for the missing initialization value.

$$\mathcal{V}[\![t\ L\ x = E]\!]\ env_L\ env_F = (cstr_E \cup \{c\}, env_L[x \mapsto lv'_x])$$
$$\text{where } \mathcal{L}[\![L]\!]\ env_L = lv_x$$
$$\text{and } \mathcal{E}[\![E]\!]\ env_L\ env_F = (cstr_E, lv_E)$$
$$\text{and } lv'_x = \begin{cases} v(x) & \text{if } lv_x = \varepsilon \\ lv_x & \text{otherwise} \end{cases}$$
$$\text{and } lv_\beta = env_L\ \beta$$
$$\text{and } c = \text{"}\ lv_E \sqcup lv_\beta \sqsubseteq lv_x\ \text{"}$$

$$\mathcal{V}[\![t\ L\ x]\!]\ env_L\ env_F = (\{c\}, env_L[x \mapsto lv_x])$$
$$\text{where } \mathcal{L}[\![L]\!]\ env_L = lv$$
$$\text{and } lv_x = \begin{cases} v(x) & \text{if } lv = \varepsilon \\ lv & \text{otherwise} \end{cases}$$
$$\text{and } lv_\beta = env_L\ \beta$$
$$\text{and } c = \text{"}\ \bot \sqcup lv_\beta \sqsubseteq lv_x\ \text{"}$$

Table 3.8: Semantic equations for variable declaration

3.3.7 Statements

For the statement rules, we first cover a few simple rules. The expression statement allows us to perform function calls where we are not interested in the return value.

$$\mathcal{S}[\![\varepsilon]\!]\ env_L\ env_F = (\emptyset, env_L)$$

$$\mathcal{S}[\![E]\!]\ env_L\ env_F = (cstr, env_L)$$
$$\text{where } \mathcal{E}[\![E]\!]\ env_L\ env_F = (cstr, lv)$$

$$\mathcal{S}[\![S_1\ S_2]\!]\ env_L\ env_F = (cstr \cup cstr_2, env_{L3})$$
$$\text{where } \mathcal{S}[\![S_2]\!]\ env_{L2}\ env_F = (cstr_2, env_{L3})$$
$$\text{and } \mathcal{S}[\![S_1]\!]\ env_L\ env_F = (cstr, env_{L2})$$

Table 3.9: Semantic equations for simple statements

3.3.8 Control structures

The control flow constructs (while and if) define constraints that will allow inference to determine their basic blocks labels. These constraints allow for propagation of label constraints and are effectively the implementation of implicit flows.

$$
\begin{aligned}
&\mathcal{S}[\![\texttt{while (}\,E\,\texttt{)}\ S]\!]\ env_L\ env_F = (cstr_E \cup cstr_S \cup \{c\}, env_L)\\
&\quad \text{where } \mathcal{E}[\![E]\!]\ env_{L2}\ env_F = (cstr_E, lv_E)\\
&\quad \text{and } \mathcal{S}[\![S]\!]\ env_{L2}\ env_F = (cstr_S, env_{L3})\\
&\quad \text{and } c = \text{"}\, lv_E \sqcup lv_\beta \sqsubseteq lv_w \,\text{"}\\
&\quad \text{and } env_{L2} = env_L[\beta \mapsto lv_w]\\
&\quad \text{and } env_L\ \beta = lv_\beta\\
&\quad \text{and } lv_w = v(next)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{S}[\![\texttt{if (}\,E\,\texttt{)}\ S]\!]\ env_L\ env_F = (env_L, cstr_E \cup cstr_S \cup \{c\})\\
&\quad \text{where } \mathcal{E}[\![E]\!]\ env_L\ env_F = (cstr_E, lv_E)\\
&\quad \text{and } \mathcal{S}[\![S]\!]\ env_{L2}\ env_F = (cstr_S, env_{L3})\\
&\quad \text{and } c = \text{"}\, lv_E \sqcup lv_\beta \sqsubseteq lv_{if} \,\text{"}\\
&\quad \text{and } env_{L2} = env_L[\beta \mapsto lv_{if}]\\
&\quad \text{and } env_L\ \beta = lv_\beta\\
&\quad \text{and } lv_{if} = v(next)
\end{aligned}
$$

$$
\begin{aligned}
&\mathcal{S}[\![\texttt{if (}\,E\,\texttt{)}\ S_1\ \texttt{else}\ S_2]\!]\ env_L\ env_F = (env_L, cstr_E \cup cstr_{S_1} \cup cstr_{S_2} \cup \{c\})\\
&\quad \text{where } \mathcal{E}[\![E]\!]\ env_L\ env_F = (cstr_E, lv_E)\\
&\quad \text{and } \mathcal{S}[\![S_1]\!]\ env_{L2}\ env_F = (cstr_{S_1}, env_{L3})\\
&\quad \text{and } \mathcal{S}[\![S_2]\!]\ env_{L2}\ env_F = (cstr_{S_2}, env_{L4})\\
&\quad \text{and } c = \text{"}\, lv_E \sqcup lv_\beta \sqsubseteq lv_{if} \,\text{"}\\
&\quad \text{and } env_{L2} = env_L[\beta \mapsto lv_{if}]\\
&\quad \text{and } env_L\ \beta = lv_\beta\\
&\quad \text{and } lv_{if} = v(next)
\end{aligned}
$$

Table 3.10: Semantic equations for control structures

3.3.9 Assignment and return statements

It should be noticed that the constraints constructed for declarations, assignments, and return statements have a similar structure. This structure represents the similarity in the feature they are handling, be it assignment to a variables value or to the return value of a function.

$$
\begin{aligned}
&\mathcal{S}[\![x = E]\!]\ env_L\ env_F = (env_L, cstr \cup \{c\})\\
&\quad \text{where } \mathcal{E}[\![E]\!]\ env_L\ env_F = (cstr, lv_E)\\
&\quad \text{and } c = \text{"}\ lv_E \sqcup lv_\beta \sqsubseteq lv_x\ \text{"}\\
&\quad \text{and } lv_x = env_L\ x\\
&\quad \text{and } lv_\beta = env_{L\beta}
\end{aligned}
$$

$$
\mathcal{S}[\![\,\texttt{return}\,]\!]\ env_L\ env_F = (\emptyset, env_L)
$$

$$
\begin{aligned}
&\mathcal{S}[\![\,\texttt{return}\ E]\!]\ env_L\ env_F = (env_L, cstr \cup \{c\})\\
&\quad \text{where } \mathcal{E}[\![E]\!]\ env_L\ env_F = (cstr, lv_E)\\
&\quad \text{and } c = \text{"}\ lv_E \sqcup lv_\beta \sqsubseteq lv_\phi\ \text{"}\\
&\quad \text{and } lv_\phi = env_L\ \phi\\
&\quad \text{and } lv_\beta = env_{L\beta}
\end{aligned}
$$

Table 3.11: Semantic equations for assignment and return statements

3.3.10 Acts for statements

The last statement is the *acts for*-statement, which has two identical rules. This is because the difference between the two only exists at runtime. The lv_α label represents the effective authority on execution. This is stored in the label environment to be used for implicit declassification when evaluating expressions.

$$\mathcal{S}[\![\texttt{this -->?}\ p_1\ ,\ p_2\ ,\ \ldots\ ,\ p_k\ S]\!]\ env_L\ env_F = (cstr, env_L)$$
$$\text{where } \mathcal{S}[\![S]\!]\ env_L[\alpha \mapsto lv'_\alpha]\ env_F = (cstr, env_{L2})$$
$$\text{and } lv'_\alpha = lv_\alpha \sqcup \bigsqcup_{i=1}^{k} \{p_i \to \emptyset\}$$
$$\text{and } lv_\alpha = env_L\ \alpha$$

$$\mathcal{S}[\![\texttt{caller -->?}\ p_1\ ,\ p_2\ ,\ \ldots\ ,\ p_k\ S]\!]\ env_L\ env_F = (cstr, env_L)$$
$$\text{where } \mathcal{S}[\![S]\!]\ env_L[\alpha \mapsto lv'_\alpha]\ env_F = (cstr, env_{L2})$$
$$\text{and } lv'_\alpha = lv_\alpha \sqcup \bigsqcup_{i=1}^{k} \{p_i \to \emptyset\}$$
$$\text{and } lv_\alpha = env_L\ \alpha$$

Table 3.12: Semantic equations for acts for statement

3.3.11 Expressions

Most rules for expression evaluation produce no constraints and have a simple definition in terms of the label they return.

$$\mathcal{E}[\![x]\!]\ env_L\ env_F = (\emptyset, lv)$$
$$\quad \text{where } lv = env_L\ x$$

$$\mathcal{E}[\![k]\!]\ env_L\ env_F = (\emptyset, p(\bot))$$

$$\mathcal{E}[\![E_1\ op_b\ E_2]\!]\ env_L\ env_F = (cstr_1 \cup cstr_2, lv_1 \sqcup lv_2)$$
$$\quad \text{where } \mathcal{E}[\![E_1]\!]\ env_L\ env_F = (cstr_1, lv_1)$$
$$\quad \text{and } \mathcal{E}[\![E_2]\!]\ env_L\ env_F = (cstr_2, lv_2)$$

$$\mathcal{E}[\![op_u\ E]\!]\ env_L\ env_F = \mathcal{E}[\![E]\!]\ env_L\ env_F$$

$$\mathcal{E}[\![\,\texttt{(}\ E\ \texttt{)}\,]\!]\ env_L\ env_F = \mathcal{E}[\![E]\!]\ env_L\ env_F$$

$$\mathcal{E}[\![E\,\texttt{.}\,x_f]\!]\ env_L\ env_F = \mathcal{E}[\![E]\!]\ env_L\ env_F$$

$$\mathcal{E}[\![E\texttt{->}x_f]\!]\ env_L\ env_F = \mathcal{E}[\![E]\!]\ env_L\ env_F$$

$$\mathcal{E}[\![x\,\texttt{[}E\texttt{]}]\!]\ env_L\ env_F = (cstr_E, lv_x \sqcup lv_E)$$
$$\quad \text{where } \mathcal{E}[\![x]\!]\ env_L\ env_F = (\emptyset, lv_x)$$
$$\quad \text{and } \mathcal{E}[\![E]\!]\ env_L\ env_F = (cstr_E, lv_E)$$

Table 3.13: Semantic equations for expressions

3.3.12 Declassification

The two rules for declassification show some of the simplicity that is part of inference. Either an explicit label is defined for declassification or one (represented by the variable label lv_d) will be inferred. Aside from that the rules are exactly the same.

$$\mathcal{E}[\![\texttt{<|}\ E\ \texttt{|>}]\!]\ env_L\ env_F = (cstr \cup \{c\}, lv_d)$$
$$\quad \text{where } \mathcal{E}[\![E]\!]\ env_L\ env_F = (cstr, lv_E)$$
$$\quad \text{and } c = {}''\ lv_E \sqsubseteq lv_d \sqcup lv_\alpha\ ''$$
$$\quad \text{and } lv_d = v(next)$$
$$\quad \text{and } lv_\alpha = env_L\ \alpha$$

$$\mathcal{E}[\![\texttt{<|}\ E\ ,\ L\ \texttt{|>}]\!]\ env_L\ env_F = (cstr \cup \{c\}, lv)$$
$$\quad \text{where } \mathcal{E}[\![E]\!]\ env_L\ env_F = (cstr, lv_E)$$
$$\quad \text{and } c = {}''\ lv_E \sqsubseteq lv \sqcup lv_\alpha\ ''$$
$$\quad \text{and } lv = \mathcal{L}[\![L]\!]\ env_L$$
$$\quad \text{and } lv_\alpha = env_L\ \alpha$$

Table 3.14: Semantic equations for declassification

3.3.13 Function call

The final type of expression is a function call. There are two rules regarding function calls. One for functions that have label definitions and one for those that do not. The latter is meant for externally declared functions, such as library functions. For both rules we have that the caller authority $\texttt{<<<}\,p_1\,,\,p_2\,,\,\ldots\,,\,p_k\,\texttt{>>>}$ is optional, and in neither rule does it have any effect, being a run-time concept.

In order to evaluate function calls, where the function declaration uses constant parameters, we need to replace these constants with the corresponding argument label. In order to do this, we declare an auxiliary function:

$$replaceConstants : \mathbf{LV} \times \mathbf{EnvL} \rightarrow \mathbf{LV}$$

with the following definition:

$$replaceConstants(lv, env_L) = \begin{cases} env_L\ x & \text{if } lv = c(x) \\ j(replaceConstants(lv_1), & \text{if } lv = j(lv_1, lv_2) \\ \quad replaceConstants(lv_2)) & \\ lv & \text{otherwise} \end{cases}$$

$$\mathcal{E}[\![f \texttt{<<<} p_1 \texttt{,} p_2 \texttt{,} \ldots \texttt{,} p_k \texttt{>>>} \texttt{(} E_1 \texttt{,} E_2 \texttt{,} \ldots \texttt{,} E_n \texttt{)}]\!]\ env_L\ env_F = (cstr_a \cup cstr_p \cup cstr_o, lv'_f)$$

if $env_F\ f = (lv_f, \{(lv_1, x_1), (lv_2, x_2), \ldots, (lv_n, x_n)\}, \{r_1, r_2, \ldots, r_k\})$ and

where $replaceConstants(lv_f, env_{L2}) = lv'_f$

and $cstr_a = \bigcup_{i=1}^{n} cstr_i$

and $cstr_p = \bigcup_{i=1}^{n} \begin{cases} \emptyset & \text{if } lv_i \in c(\mathbf{Var}) \\ \{''\, lv_{E_i} \sqsubseteq lv_i\, ''\} & \text{otherwise} \end{cases}$

and $\mathcal{E}[\![E_i]\!]\ env_L\ env_F = (cstr_i, lv_{E_i})$ for $0 \leq i \leq n$

and $env_{L2} = env_L[x_1 \mapsto lv_{E_1}, x_2 \mapsto lv_{E_2}, \ldots, x_n \mapsto lv_{E_n}]$

and $cstr_o = \begin{cases} \bigcup_{i=1}^{n} \{''\, lv_{E_i} \sqsubseteq lv_o\, ''\} & \text{if } k > 0 \\ \emptyset & \text{otherwise} \end{cases}$

and $lv_o = \bigcup_{p \in P^*} \{p \rightarrow r_1, r_2, \ldots, r_k\}$

$$\mathcal{E}[\![f \texttt{<<<} p_1 \texttt{,} p_2 \texttt{,} \ldots \texttt{,} p_k \texttt{>>>} \texttt{(} E_1 \texttt{,} E_2 \texttt{,} \ldots E_n \texttt{)}]\!]\ env_L\ env_F = (cstr_E, lv_f)$$

if $env_F\ f = undefined$ and

where $cstr_E = \bigcup_{i=1}^{n} cstr_i$

and $lv_f = \bigsqcup_{i=1}^{n} lv_i$

and $\mathcal{E}[\![E_i]\!]\ env_L\ env_F = (cstr_i, lv_i)$ for $0 \leq i \leq n$

Table 3.15: Semantic equations for function calls

Chapter 4

Time Policies

In this chapter we will extend our previous description of CTIF with the use of time policies. The same concepts as previous still hold, as we want a simple way of declaring time policies to certain values in a program, in order to restrict the access to these values.

We will make a proposal for a practical extension, of the previously discussed security labels, using time policies. After providing a description of these new time policies, we show how they can be translated into timed automata, so that relevant theory may still apply.

Finally we compare the time policies of CTIF with TDLM [8]. The two projects provide means of specifying time policies, though through different approaches. Where CTIF was constructed from a practical and directly applicable perspective, TDLM tales a more formal approach.

4.1 Extending the security model

Extending on the concept of controlling how information should exit the system, as we did previously using channels, we want to add time constraints to the security model. As we have different principals in our system, which have different requirements/rights, we want to be able to express multiple time policies which apply to different principals. To do this we extend the previously defined syntax for labels and include some language constructs that simplify how to work with time considerations. This language feature is not considered an extension of the labels defined by DLM. One of the reasons for this is that statically determining which time policy is the most restrictive and how to join two policies together presents some difficulties. In Section 4.1.3 we will go into details about these challenges.

Even though the time policies do not fit in the same model as the labels defined by DLM, we have chosen to provide a shared syntax for both types of security policies. Grouping all

the security policies together like this, will make it easier for the programmer to identify which policies apply in a specific context.

4.1.1 Expressiveness

We expand on the concept of labels using a set of properties that describe rules for when, and how often, data can be read. We want our time policies to be expressive, so that we can cover a wide variety of cases. We therefore introduce three types of time constituents, where any combination of these will form a time policy.

In the following, we will consider time values as positive integers representing milliseconds. For simplicity, we will use the time postfixes: $\mathbb{T} = \{h, m, s, ms\}$, which are to be seen as factors to be applied to times. For a value $n \in \mathbb{N}$ followed by a postfix $T \in \mathbb{T}$, we have that:

$$nT = \begin{cases} n \times 60 \times 60 \times 1000 & \text{if } T = h \\ n \times 60 \times 1000 & \text{if } T = m \\ n \times 1000 & \text{if } T = s \\ n & \text{otherwise} \end{cases}$$

Period
Period represents a start- and end-time, where access is only permitted within that period. E.g. `{{u->u, ec@09:00-10:00}}` signifies that data can only be accessed when the time is between 09:00 and 10:00 in the morning.

Formally we represent a period as the pair (p_s, p_e) describing when a period starts (inclusive) and ends (exclusive). A period cannot specify rules for particular dates or days of the week.

Interval
Interval is a constant value indicating the minimum amount of time there should pass between each access. The interval can be described as a combination of milliseconds, seconds, minutes, hours, and days. E.g. `{{u->u, ec@10m30s}}` requires a minimum of 10 minutes and 30 seconds between each access.

Count
Count is in itself without meaning, as it says how many consecutive accesses within the given period and/or interval are allowed. However, due to default values some meaning may apply to a case such as `{{u->u, ec@10m*5}}`, where we have that 5 consecutive accesses are allowed in a 10 minute interval.

Combined time policies
It is possible to combine two or all three of the time constituents. The count component even requires such a combination. Count should be considered a "*number of reads in some*

period or interval". If an interval is declared, count specifies the number of reads in that interval (defaulting to 1). If an interval is not declared, but a period is, count specifies the number of reads in that period (defaulting to ∞).

> **Example 4.1 (Time policies)**
> The `{{pc->u@10m*3}}` policy indicates that access is only allowed access thrice per 10 minutes. Another way of explaining it; after the initial read another two reads *can* follow, either way it is reset 10 minutes after the initial read.
> Another example `{{u->u, ec @ u:00:00-24:00 30m; 00:00-09:00}}` allows `u` unlimited access. Everyone else are however restricted to reading between midnight and 9 in the morning, and with a minimum of 30 minutes passing between each access.

From the above examples we have that a label can be extended with time policies using a `@` followed by a list of time policies. All except the last policy are preceded with the name of the principal to whom the policy applies. The last policy is not associated with a principal and serves as the default time policy. In Section 4.1.5 we describe how a policy is selected from such a declaration of policies.

4.1.2 Applying time policies to the examples

With the language constructs described in the previous section, we will now revisit our two code examples. In the following we will discuss how time policies can be added to the password checker and the bill calculator (see Listings 3.2 and 3.3 on pages 33 and 34 respectively).

As with labels, we will only apply time policies where they directly apply to data sources. The programmer should not be forced to declare additional policies in order to satisfy certain constraints that the compiler is trying to enforce. In Section 4.1.3 we describe difficulties in using inference for time policies and in Section 4.1.4 we define a set of constructs that the programmer can use to define how his time policies should be handled.

Password checker

In the password checker we would like to restrict how often the collection of user information `get_users` can be accessed. To achieve this effect we update the function declaration:

```
user_info {{pc-> @ 10m * 3}} *get_users(){};
```

Access to the data will then be restricted to three times every ten minutes, rendering brute force attacks very slow. Unfortunately, this time policy is shared across all principals, meaning that all principals/users will have to share the three accesses every ten minutes. A simple solution to this is to define a time policy for each principal, and possibly the same policy for all.

This does not provide a dynamic solution when the number of principals is not known in advance. But as CTIF was primarily developed for smaller embedded devices it can be expected that the set of principals will not be expanded dynamically.

Bill calculation

In the smart meter bill calculation we would like to restrict how often different principals are allowed to retrieve the consumption data of a users smart meter. To do this we add a time policy to the `get_latest_usage` function:

```
usage {{u->u, ec @ u: 1s, 00:00-01:00 14d}} *get_latest_usage(){};
```

Here we specify that the user is allowed to read his usage data every second, but that everyone else can only read once every other week and only during the first hour of the day. The granularity of these policies could naturally be altered if so needed, but this policy demonstrates how easy it is to define a timed privacy policy. The user will be able to closely watch his consumption data, while the electrical company will retrieve billable consumption data every other week.

It should be noted that in order to make this example work, the `get_latest_usage` function would have to be maintain a list of what data each principal has received. Though this could be handled strictly using static evaluation (using the acts for construct), it would prove a more dynamic solution to use some runtime information about which principal is executing the function.

Runtime issues

Having expanded the examples with time policies, we have noted that there are some difficulties with providing principal-specific feedback from functions. If the set of principals is fixed (which might often be the case for embedded devices) the issues can be handled at compile time. Should we however desire a dynamic solution for principals we would require specific runtime handling of principals. This has not been part of CTIF.

4.1.3 Inference of time policies

Having described how label information is propagated from declaration to dependencies, we effectively allow the programmer to specify policies only where they make some intuitive sense. The many benefits of this has already been described and is due to the use of the inference algorithm. We would like for this algorithm to infer time policies in the same fashion as it infers owner/reader policies. To do that we would have to expand the label model with time policies.

Time policies could be defined as a tuple (p_s, p_e, i, c), describing the period, interval and count of the policy. Part of a label would then involve a function that defines the time policy of each principal.

In the following we will use various tuple sizes for representing time policies, depending on which components are relevant to the context. A tuple that does not contain all four

time policy components is meant to represent only a part of a policy. Which part will be clear from the use of p, i and c in each context.

Examining the algorithm (page 23) we see that in order to infer time policies we must be able to define the $\sqsubseteq$ and $\sqcap$ operations for these time policies. Additionally we would want to define $\sqcup$ for a complete lattice of time policies.

4.1.3.1 Time policy restriction

We will start with a definition for whether a time policy $(p_{s1}, p_{e1}, i_1, c_1)$ is no more restrictive than another $(p_{s2}, p_{e2}, i_2, c_2)$. Defining this operation for any single time policy component is quite simple:

- If a period contains another period it is no more restrictive than that period; $(p_{s1}, p_{e1}) \sqsubseteq (p_{s2}, p_{e2})$ if $[p_{s1}; p_{e1}[\supseteq [p_{s2}; p_{e2}[$
- A time interval is no more restrictive than a longer time interval; $i_1 \sqsubseteq i_2$ if $i_1 \leq i_2$
- A number of allowed counts is no more restrictive than a lower number of counts; $c_1 \sqsubseteq c_2$ if $c_1 \geq c_2$

When comparing two full time policies, we can start by comparing the time periods. If we find that a policy is more restrictive here, we say that the entire policy is more restrictive. If not, we will look at the interval and count component. We make this distinction, as the period component specifies at which point in time the remaining components apply.

When comparing the interval and count the process is not as obvious, as we present three options for how these two components could be compared:

1. Determine if interval is more restrictive and if not, determine count.
2. Determine if count is more restrictive and if not, determine interval.
3. Compare the two policies by the number of reads allowed per time unit; $(i_1, c_1) \sqsubseteq (i_2, c_2)$ if $\frac{c_1}{i_1} \geq \frac{c_2}{i_2}$

We do not consider either of these policies to be the *correct* one. It might be tempting to consider the latter as the best choice since it simultaneously considers both interval and count. It does however not encompass all considerations. Below follows an example that illustrates the difficulty associated with selecting a definition:

Example 4.2 (Comparing time policies)
Consider the two time policies t_1 and t_2, declared as `10m*10` and `5m*5`.
If we apply the first definition, we note that `10m` is more restrictive than `5m` and so $t_1 \sqsubseteq t_2$ must be false. As `*10` is less restrictive than `*5`, the second definition yields the same result. Finally using the third definition $t_1 \sqsubseteq t_2$ must be true, as each policies allows for one read per minute.
If we flip the operands to $t_2 \sqsubseteq t_1$ the definitions yields true, true and true. Thus the final definition is not able to establish an order for the two policies.
Having the freedom to do 10 reads at any time in 10 minutes does however provide some additional freedom over two times 5 reads in 5 minutes; the option to freely determine when the 10 reads are used.

We will not go into further details about selecting an appropriate definition. Selecting which to use could be left to the programmer when applying time policies to his application.

4.1.3.2 Combining time policies

Having an approximate definition for the $\sqsubseteq$ operation on time policies we will try to define a matching definition for joins and meets of time policies. As one is the inverse of the other we will only consider meets of policies. This fills our requirement for the inference algorithm. With a definition for meet it should be simple to also provide a definition for joins. The meet of two time policies should be:

The most restrictive policy that is no more restrictive than either operand.

As above we start by only considering time policy periods, and provide the following definition:

$$(p_{s1}, p_{e1}) \sqcap (p_{s2}, p_{e2}) = (\max(p_{s1}, p_{s2}), \min(p_{e1}, p_{e2})$$

Again we note that time periods yields a simple definition for our operations. However when we examine intervals and count we run into difficulties. As with $\sqsubseteq$ we could opt for a per-component evaluation of policies. Thus we would use the least restrictive interval and count in our resulting policy. We could also employ the *reads per time unit* definition and use some *reasonable* interval and count for the resulting policy.

Thus is it possible to define some set of operations for time policy inference. Unfortunately, given the example below, the result is of little use to us.

Example 4.3 (Issues with inference)
Consider two slots with the time policies `10m*10` and `5m*5`. A natural use of inference is to store the sum of the two slots in a third slot, yielding a new time policy for that slot. Reading the value from this slot does not require time policies, as it will not be updated with a new value until the two original slots are updated. And these are protected by time policies.

The issue we encounter has to do with the nature of time policies as opposed to labels as we know them from DLM. Labels specify what we are allowed to do, but time policies specify only when we can do it.

This is quite a different concept to handle, as understanding when something can happen should be evaluated at runtime, whereas our handling of labels and the tool we provide is based on static evaluation. We note that labels in DLM are idempotent, as meeting or joining any label L with itself results in that same label. As this is one of the properties of a lattice, we would require our time policies to have this property as well. However if we chose to read from the same slot twice we would have used two reads and thus the resulting label could not be the same as that of the slot.

From this it is clear that we can not provide inference of time policies in the same way that we do for labels. Though at the same time it is also clear that we do not stand to gain much from time policy inference, as demonstrated by the example above. Instead we have chosen to introduce certain language constructs that will allow time policies to be evaluated statically.

4.1.4 Time constructs

When calling a function that has a time policy we could, at runtime, be in one of two different states. Either we are in a state where the function can not be called or in one where it can.

If the function can be called we would like to do so in as normal a fashion as possible. If the function can not be called we would require some means for how to handle a function anyway. To satisfy these considerations we introduce the following two language constructs, specific to time policies:

- if-can-call `@?`
- await `@`

The two constructs represent run-time checks and are implemented as expressions with the same precedence as function call[1]. The `@?` construct must precede a function identifier, and returns a boolean indicating whether that function is currently callable or not, as per the time constraints. The `@` construct is used to extent function calls. Placing `@` before a function call, forces the program to busy-wait until the function is callable and then calls the function.

Using these constructs we can perform a check, similar to a return-check, where each branch of a functions control-flow will be evaluated using these constructs. They could for instance be used in a conditional expression:

```
if (@?foo) {
  foo(); // Valid, due to above check.
  foo(); // Invalid, due to above call of same function.
```

[1] http://en.cppreference.com/w/c/language/operator_precedence

```
}

if (@?foo) {
  foo(); // Valid, due to above check.
  @foo(); // Valid, will busy-wait if needed.
}

if (@?foo) {
  @foo(); // Valid, but consumes check.
  foo(); // Invalid, due to above call of same function.
}
```

Additionally, we can include checks in the condition of a while loop, letting it continuously call a function until its time policy no longer allows it. We can also perform one-line checks and calls to evaluate the value returned from a function:

```
if (@?foo && foo() > 10) {
  // The value was read and it was greater than 10.
}
else {
  // The value could not be read, or was less than 10.
}
```

The same property does not apply to the `||` operator due to short-circuit evaluation. However negation of a `@?` check is possible, and will allow for calling the function in the else branch of a conditional statement.

Using the ternary operator we can specify default values for functions that cannot be called. Here negation is also supported.

```
int a = @?foo ? foo() : -1; // Valid
int b = !@?foo ? -1 : foo(); // Valid, but somewhat confusing
```

4.1.5 Selecting a policy

As described in Section 4.1.1, a time policy can be associated with a principal or it can be the default policy and thus apply to all other principals. A single label could specify multiple time policies applying to various principals. An example of such a set of policies is given below:

```
{{u->u, ec @ u:5m*10; 1h}}
```

Here the principal u is allowed to access ten times in five minutes. All other principals will only be allowed access once an hour.

To determine which policy applies in a given context we employ the effective authority, as described in Section 2.5. Thus for the above policies, a programmer will have to raise the

effective authority if he wants to employ us time policy. The employed authority will be the least restrictive one available, given the effective authority when the function is called. As described in Section 4.1.3.1 there are multiple ways to define which time label is the most restrictive. We do not provide means for declaring which type should be used by the tool. Such a check is only required for the runtime aspects of time policies and is therefore ignored in the context of our tool.

In the example below we demonstrate how authority can be used to determine if and how a function should be called.

Example 4.4 (Using authority for time policies)
In the following code example we make use of all the constructs specific to time policies. We allow the program to busy-wait if we are allowed to act for the principal u, otherwise we call the function foo if we can do so without waiting.

```
int {{u->u, ec @ u:5m*10; 1h}} foo() {
  // Method body
}

int getfoo() {
  this -->? u { // Executed if authority can be raised
    return @foo(); // Will wait up to five minues before execution
  }
  else if (@?foo) { // Executed if the function can be called
    return foo();
  }
  else {
    return -1;
  }
}
```

As the tool developed only considers static evaluation we do not provide a run-time implementation of time policies or the effect of using the two constructs defined above. In Section 4.2.1 we describe how timed automata can be constructed from our time policies.

4.1.6 Applying time constructs to the examples

Now that we have defined time specific language constructs, we will demonstrate how they can be used to handle the time policies defined in Section 4.1.2. The constructs will only be used in conjunction with the functions that we applied time policies to.

Password checker

Handling the time policy of the password checker will be done in a very simplistic fashion. If the collection of user information cannot be accessed, due to time restrictions, the

function will simply return false. This is achieved by inserting the following lines in the start of the function:

```
if (!@?get_users)
  return false;
```

As the get_users function call is performed only once after this check, the code is now valid. When more than three attempts to validate a password has been made, the function will fail validation on all requests. Alternatively, we could make the user wait for a response from the server, by simply prepending the get_users function call with a @:

```
user_info *users = @get_users();
```

In this case the response time of the function is massively increased when too many request are made. The function will however not produce any false negatives.

Bill calculation

For bill calculation it would not be wise to simply just employ the @ wait construct for usage retrieval. The electrical company could then theoretically request data and await 14 days for the result. Instead we utilize the scheme for function calls that was demonstrated in Example 4.4, and define a helper function for the get_latest_usage function:

```
usage *get_latest_usage_timed()
{
  this -->? u
    return @get_latest_usage();
  else if (@?get_latest_usage)
    return get_latest_usage();
  else
    return NULL;
}
```

As u can read every second we will accept the waiting period for the user's request. The electrical company will receive a null pointer value when the time policy does not allow a read.

Naturally we replace the old get_latest_usage function call with one to get_latest_usage_timed and include the following check:

```
if (usage == NULL)
  return -1;
```

Timed code examples

With these new constructs the time policies are handled in the both code examples. We saw that the constructs made it easy to express simple means of handling time policies,

that there are multiple ways to handle policies, and that sets of multiple time policies can easily make use of multiple handling methods.

In Listings 4.1 and 4.2 we present the updated code examples, including time policies and the handling of these.

4.2 Timed automata

As the first step towards formalising the time policies of CTIF, we will show how they can be translated into timed automata. Before doing so in Section 4.2.1, we will first shortly define and describe timed automata, based on [9].

The timed automata is an extension to the ω-automata formalism, allowing for the manipulation of clock-variables, as well as clock-based constraints. The formal definition for a timed automaton can be seen in Definition 4.1.

Definition 4.1 (Timed automaton)
A timed automaton is a tuple (Σ, S, S_0, C, E), where

- Σ is a finite alphabet,
- S is a finite set of states,
- $S_0 \subseteq S$ is a set of start states,
- C is a finite set of clocks, and
- $E \subseteq S \times S \times \Sigma \times 2^C \times \phi(C)$ gives the set of transitions.
  An edge $(s, s', \sigma, \lambda, \delta)$ represents a transition from state s to state s' on input symbol σ. The set $\lambda \subseteq C$ gives the clocks to be reset with this transition, and δ is a clock constraint over C.

In Example 4.5 can be seen a visualization of a timed automaton. Generally, for each edge we have the input σ, clock resets λ, and clock constraints $\phi(C)$. In our automata, the input will represent the action of reading a value with the corresponding name. Clock resets are denoted $x := 0$, which corresponds to $\lambda = \{x\}$, and in this case will reset the clock variable x to 0. Clock constraints are denoted $(x >= 10m)?$, which is a single clock constraint for the clock variable x. Several input characters, clock resets, and clock constraints can be defined for a single edge, the syntax is used to distinct between them. It should also be noted that, in our clock constraints, we will use the same concept of time factors as those described for our time policies (see Section 4.1.1).

```
1  #include <stdbool.h>
2  #include <string.h>
3
4  principal u, pc;
5
6  typedef struct user_info {
7    char username[20];
8    char password[20];
9  } user_info;
10
11 user_info {{u->u}} get_login();
12 user_info {{pc-> @ 10m * 3}} *get_users();
13 u <- void send_response(bool is_match);
14
15 bool check_password(char *username, char *password) {
16   int user_count = 100;
17   user_info *users = @get_users();
18   int i = 0;
19   bool match = false;
20
21   while (i < user_count) {
22     if (!strcmp(users[i].username, username) && !strcmp(users[i].
       password, password)) {
23       match = true;
24     }
25     i = i + 1;
26   }
27
28   this -->? pc {
29     return <|match|>;
30   }
31 }
32
33 int main(int argc, char **argv) {
34   user_info login = get_login();
35   bool is_match = check_password(login.username, login.password);
36   send_response(is_match);
37 }
```

Listing 4.1: Timed password checker example

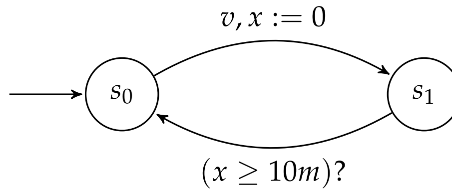
```
13 usage {{u->u, ec @ u: 1s, 00:00-01:00 14d}} *get_latest_usage();
14 price {{_}} *get_latest_prices();
15 u <- void send_to_consumer(int bill_total);
16 ec <- void send_to_electrical_company(int bill_total);
17
18 usage *get_latest_usage_timed()
19 {
20   this -->? u
21     return @get_latest_usage();
22   else if (@?get_latest_usage)
23     return get_latest_usage();
24   else
25     return NULL;
26 }
27
28 int {{u->u, ec}} calculate_bill() {
29   int usage_count = 100;
30   int prices_count = 100;
31   usage *latest_usage = get_latest_usage_timed();
32   price *latest_prices = get_latest_prices();
33   int result = 0;
34
35   if (usage == NULL)
36     return -1;
37
38   int i = 0;
39   int j = 0;
40   while (i < usage_count) {
41     while ((j < prices_count-1) && (latest_prices[j+1].start_time <=
       latest_usage[i].start_time)) {
42       j = j + 1;
43     }
44     result = result + latest_usage[i].usage_in_Wh * latest_prices[j].
       price_in_cents;
45     i = i + 1;
46   }
47   this -->? s {
48     return <|result|>;
49   }
50 }
51
52 int main(int argc, char **argv) {
53   int bill_total = calculate_bill();
54   send_to_consumer(bill_total);
55   send_to_electrical_company(bill_total);
56 }
```

Listing 4.2: Timed smart meter bill calculation example. Struct declarations have not been included here to make the code example fit in a single page. The structs are unchanged from Listing 3.3

Example 4.5 (Visual representation)
In this example we have a timed automata with the following characteristica:

- $\Sigma = \{v\} \cup \{\varepsilon\}$
- $S = \{s_0, s_1\}$
- $S_0 = \{s_0\}$
- $C = \{x\}$
- $E = \{(s_0, s_1, v, \{x\}, \emptyset), (s_1, s_0, \varepsilon, \emptyset, \{10m \geq x\})\}$

This timed automaton is concerned with protecting the value of v, which should be read with at least 10 minutes inbetween reads. We start in state s_0, with all clock variables initially set to 0 (although this is not important for this example). At some point, we can make the transition from s_0 to s_1, which is an unconstrained transition. Once we do this x will be reset and we perform the action of reading v. Now, in s_1, we cannot transition back to s_0 before we have waited for at least 10 minutes, hence the constraint on the edge going from s_1 to s_0. Once enough time has passed, we can carry out the transition and from there we are again permitted to perform the transition from s_0 to s_1 in order to read v.



4.2.1 Time policies and timed automata

In this section, we will show how it is possible to transform CTIF time policies into timed automata. This will be done by defining several abstract time policies, representing any variation depending on the included time constituents, and showing for each of these a corresponding timed automaton.

We assume having a special clock variable τ_s representing the current system time, which will be used for evaluating period constraints. For readability purposes, whenever we have a time policy with a period (p_s, p_e) we will use the constraint δ_p, representing our period constraint:

$$\delta_p = (p_s \leq \tau_s < p_e)$$

along with its negation

$$\neg\delta_p = (p_s > \tau_s \geq p_e)$$

4.2.1.1 Period

A period-only policy is represented by a tuple (p_s, p_e, ∞), since we previously defined it as having an implicit count value of $c = \infty$. In this case, however, $c = \infty$ simply means that we can read without further restrictions, as long as our period constraint holds.

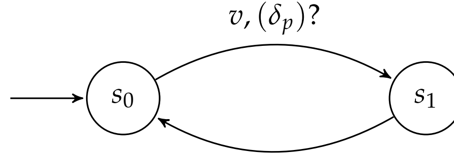


Figure 4.1: Abstract period-only policy

4.2.1.2 Interval

An interval-only policy is represented by a tuple $(i, 1)$, with the implicit count value of $c = 1$. We will now need a clock variable in order to limit the number of consecutive reads within the given interval. Additionally, s_1 will now also represent a state in which the first (and in this case only) read has been performed. The automaton can now be stuck in state s_1, while waiting for the interval constraint to be fulfilled.

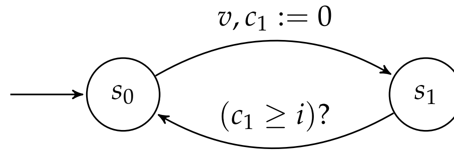


Figure 4.2: Abstract interval-only policy (with implicit $c = 1$)

4.2.1.3 Period and count

Similar to the interval-only policy, when extending a period-only policy with a specific count: (p_s, p_e, c), we will need a state for each count in order to represent how many consecutive reads have been performed. Unlike the interval-only policy, we do not need a clock variable. Instead, we need only ensure the period constraint for each read, as well as reset (by returning to s_0) whenever we are no longer within the period.

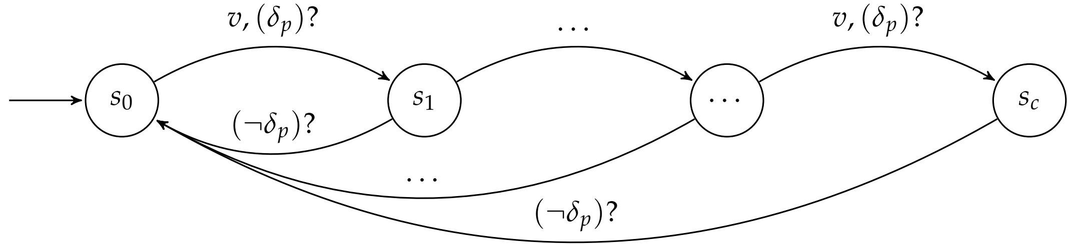


Figure 4.3: Abstract policy with a period and count

4.2.1.4 Interval and count

When extending an interval-only policy with a specific count, something interesting happens. In order to visualize this, we will here make us of a concrete count, so that we have a policy: $(i, 2)$. This results in 4 states, one state representing each possible combination of consecutive reads, having that we have performed reads: none, first only, first followed by second, second only, second followed by first (after a reset).

Generally, we will have 2^c states, and the ability to transition from any state where we either perform a new read or the clock variable for a former read has been reset.

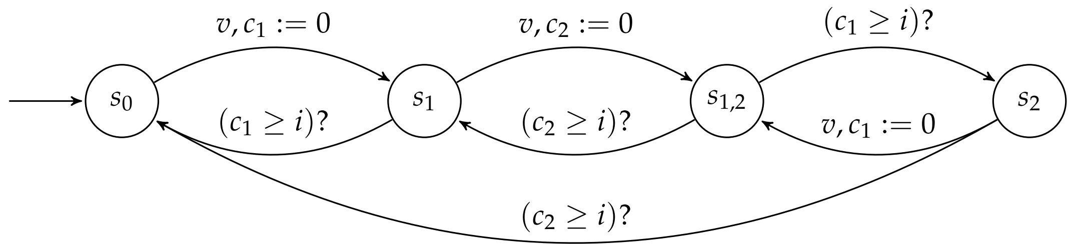


Figure 4.4: Abstract policy with interval and count ($c = 2$)

4.2.1.5 Period, interval, and count

If we now try to combine all three time policy constituents, while still having a concrete count: $(p_s, p_e, i, 2)$, we can see that little change is needed.

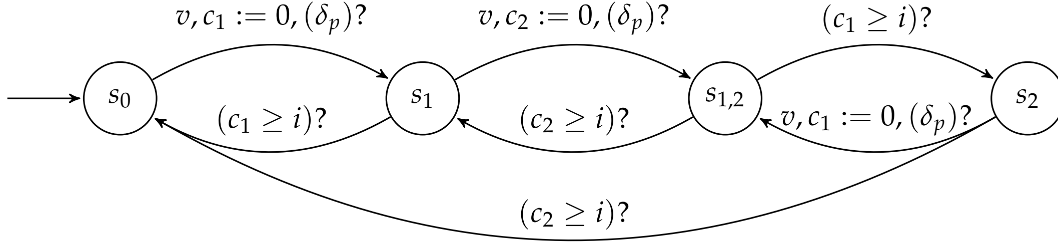


Figure 4.5: Abstract policy with period, interval, and count ($c = 2$)

4.3 The Timed Decentralized Label Model

The Timed Decentralized Label Model (TDLM) [8] is an extension to the original DLM. TDLM extends on the security labels of DLM with the addition of time policies. We are interested in is what kind of policies can be expressed using TDLM. We will first briefly explore its usage and afterwards we will compare it to CTIF.

4.3.1 Usage

The overall idea is that *clock comparison* rules can be added to the policies of labels. These rules are logical comparisons of some *clock variables*, or constant values, which must hold before data can be read by the concerned principal.

4.3.1.1 Clock comparisons

Clock comparisons can be added to any principal in a policy and time policies may differ between principals. If a clock comparison is added to the *owner* of a policy, it applies to all *readers* in that policy. In the following label, a simple time restriction, containing a clock comparison, is added to the read rights of principal r_1:

$$\{o : r_1(x > 500), r_2, r_3\}$$

The policy simply states that the principal r_1 is not allowed to read the attached value until the clock variable x is greater than 500 ms. There are no *clock variable parameters* attached to x (see below), so no reset rules have been defined, meaning that once 500 ms has passed r_1 may read infinitely many times, or until x is reset elsewhere. The other readers r_2 and r_3 have no time restrictions.

4.3.1.2 Upper limit and reset value

Extending on the previous example, we now have that x includes some simple reset rules:

$$\{o : r_1(x[1000; 0] > 500), r_2, r_3\}$$

Here we have specified that once the clock reaches an *upper limit*, in this case 1000, it is reset to its *reset value* of 0. Now, r_1 has a 500 ms window for every second in which it can read the attached value.

4.3.1.3 Reset events

Alternatively, and not excluding the use of the previous reset method, we can add reset events to a clock variable:

$$\{o : r_1(x[?xreset] > 500)[*xreset], r_2, r_3\}$$

Here we have substituted the upper limit with a reset event *xreset*, which will reset clock variable x whenever the event is triggered.[2] ?*xreset* names a reset event name for x, while $*$*xreset* triggers the event *xreset* any time r_1 reads the attacked value. Now, instead of resetting every 1 second, x is reset to 0 any time that r_1 reads the attached value, effectively limiting r_1 to wait at least 500 ms between each read.

4.3.2 Comparison

We now proceed to compare the time constraints expressed in TDLM and CTIF. We note that a major difference between the two models is that TDLM supports write policies and that CTIF does not. As described in Section 2.1 the concepts of read and write policies are similar, and we expect that any actual application of CTIF would require an extension to support write policies. Thus we consider a comparison of read policies to be a fair comparison of the two models.

Naturally we identify that TDLM supports some complex time policies that cannot be expressed in CTIF. However which policies that includes is not apparent.

We consider this the main strength of the time policies defined by CTIF. Due to the simplicity of the time policies we are able to clearly express properties of CTIFs time constraints. One such property is the ability to define a count in a time policy, which is not directly possible to do in TDLM. The fact that this can be expressed in such understandable terms demonstrates the practical nature of CTIF.

From [8] we have a policy attached to smart meter consumption data. We have removed write policies from it, to get the policy below:

$$\{s_i : u_i, e_j(x[?reset : 1] > 90)[*reset]\}$$

[2]Note that there is an implicit reset value of 0 if no value is given.

Here we have a smart meter s_i as owner, allowing for user u_i and electrical company e_j as readers. The electrical company is associated with a time policy, determining when it can perform reads. From reading the policy it is not clear what the desired effect of that policy is. This makes it hard for the programmer to reason about whether or not he is using the *correct* policy.

The policy ensures that the electrical company can only read consumption data once every 90 days. It provides no restrictions on other readers. In CTIF we can express a similar policy:

```
{{s->u, e @ u: 1s; 90d }}
```

This policy is much clearer in the restrictions it expresses. Because of this CTIF policies can be read, left to right, directly from code. This makes it very easy for a programmer to reason about the policy. The above example is read as:

> *The data is owned by the smart meter* `s`*, allowing reads* `->` *by the user* `u` *and the electrical company* `e`*. With regards to time* `@`*, the user* `u` *is allowed to read at most once per second* `1s`*, while everyone else at most once per 90 days* `90d`*.*

In the TDLM policy we have a lot of complex concepts, arriving from its theory-near approach. The syntax is near its underlying driver: the timed automata, as can be seen from the use of an actual clock comparison and the reset event and value.

We have however opted for abstracting away from these more complex concepts, such that the programmer need only focus on describing his policy in real-world terms.

Chapter 5

Conclusion

In this report we have provided a formalization of the static label checking of the Decentralized Label Model. We have provided a formal set of semantic functions describing the constraints that are associated with a program. By further specifying default label values for any missing label declaration, we are able to apply this specification to both labeled and unlabeled programs. This is achieved through an algorithm that infers labels for these missing label declarations. The algorithm is originally described in [2]. In this report we have presented an actual algorithm that replicates this description.

All of the above have been implemented in a tool for checking information flow in programs. Using simple constructs they have been created as an extension of the C programming language. This language was chosen in order to provide a simple-to-use security model for a low level language, that is used for embedded devices.

Through the development process we have continuously experimented with the use of labels in programs. Because of this hands-on experience the tool has been developed bottom-up, with a focus on what the programmer requires. The result of this is a tool that provides feedback for the programmer similar to a modern IDE. This feedback describes if any information flow policies are invalid or could not be determined through inference.

Further we have developed a set of language constructs for describing time policies alongside the existing labels. When developing these time constructs, the focus on programmer-first was maintained. This resulted in a minimalistic, yet quite expressive, set of time policy constructs for the language. Furthermore, we have demonstrated how these policies can be represented as timed automata, providing a formalism that is associated with much existing theory.

In the following we will attempt to evaluate our results with respect to the above, and discuss some of the difficulties we have noted in our work with DLM and time policies. These range from issues that we would have liked to have solved with our tool, to some that would require a different type of tool than the one we have created.

5.1 Runtime model

A common denominator for many of the difficulties we have had is the lack of a runtime representation of our model. This applies to both the formalization of DLM and our time policies. One example of this is the lack of runtime labels. DLM fully describes labels as an extension of a type system, and as such you are allowed to operate on labels as first-order types.

Our model has a different perspective on labels, considering only the static declarations. Not being able to pass label values as function parameters hinders some aspects of label polymorphism and provides for a less flexible model. On the other hand, a less flexible model is also a sturdier one. The fact that all our evaluation can be done at compile time allows us to provide feedback to the programmer, about every label in his program, while he is editing code. We consider this a strength of our tool, though recognizing that lack of flexibility it brings.

The same can be said for time policies, where it can be particularly difficult to define dynamic handling of requests from different users, as described in Section 4.1.2. However, as the domain we wish our tool to apply to is mainly IoT embedded devices, we recognize that the need for a dynamic approach to labels and/or authority will not be of much importance.

5.2 Code generation

Dual to the lack of a runtime model, the developed tool does not perform code generation. We expect a finalized version of the tool to output C code that meets the requirements of the policies defined. Though in terms of code generation we identify a distinction between labels and the time policies.

As labels are statically evaluated, and since we do not support runtime labels, code generation is almost as simple as removing any label-specific policies. The only addition to that is the acts-for construct. This requires us to generate some system-dependent code that will manage the authority under which code is being executed. Asides from this management of authority, labels provide no additional overhead on the generated code. The information flow is simply checked at compile time, but is not required as part of the generated code.

Time policies on the other hand would require additional code generated, in order to handle the time checks. One simple approach is to implement timed automata as described in Section 4.2.1. Though as we described, the automata grows exponentially relative to the count component of policices. Because of this, the implementation might differ from a strict implementation of the automata.

Handling time policies
The language constructs used to handle time policies (if-can-call `@?` and await `@`) have not

been included in the timed automata described in Section 4.2.1. The use of these could lead to race conditions. Since the constructs have not been included in the automata, we cannot determine if that is the case for the use of one individual time policy. We have however already demonstrated examples of race conditions:

```
if (@?foo)
  foo();
```

Here it could be the case that from the check of whether `foo` can be called and to the actual call, we are no longer within the period in which `foo` can be read. Since the check is only required to be performed *before* the function call, and not right before, we have no way of knowing how much time passes between the two. This issue is specific to the period component, as it is the only type of time restriction where waiting a longer time can make a function call invalid.

Bibliography

[1] Mikkel Sandø Larsen, Stefan Marstrand Getreuer Micheelsen, Bruno Thalmann, and Mikael Elkiær Christensen. *Smart Meter Security Analysis*. Tech. rep. Department of Computer Science, Aalborg University, 2016. URL: http://projekter.aau.dk/projekter/files/225956709/final.pdf.

[2] Andrew C Myers and Barbara Liskov. *A decentralized model for information flow control*. Vol. 31. 5. ACM, 1997.

[3] Kevin Müller, Sascha Uhrig, Michael Paulitsch, and Georg Sigl. "Cif: A Static DLM Analyzer to Assure Correct Information Flow in C". en. In: *ICSEA 2015*. Barcelona, Spain, 2015.

[4] Tomasz Maciazek, Hanne Riis Nielson, and Flemming Nielson. *Content-Dependent Security Policies in Avionics*. 2016. DOI: 10.5281/zenodo.47981. URL: http://dx.doi.org/10.5281/zenodo.47981.

[5] Andrew C Myers and Barbara Liskov. "Complete, safe information flow with decentralized labels". In: *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on*. IEEE. 1998, pp. 186–197.

[6] Andrew C Myers and Barbara Liskov. "Protecting privacy using the decentralized label model". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9.4 (2000), pp. 410–442.

[7] K. Slonneger and B.L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. NATO Asi Series A. Life Sciences; 282. Addison-Wesley Publishing Company, 1995. ISBN: 9780201656978. URL: https://books.google.dk/books?id=HIRQAAAAMAAJ.

[8] Martin Leth Pedersen, Michael Hedegaard Sørensen, Daniel Lux, Ulrik Nyman, and René Rydhof Hansen. "Secure IT Systems: 20th Nordic Conference, NordSec 2015, Stockholm, Sweden, October 19–21, 2015, Proceedings". In: ed. by Sonja Buchegger and Mads Dam. Cham: Springer International Publishing, 2015. Chap. The Timed Decentralised Label Model, pp. 27–43. ISBN: 978-3-319-26502-5. DOI: 10.1007/978-3-319-26502-5_3. URL: http://dx.doi.org/10.1007/978-3-319-26502-5_3.

[9] Rajeev Alur and David L Dill. "A theory of timed automata". In: *Theoretical computer science* 126.2 (1994), pp. 183–235.