M.Sc. Thesis
Master of Science in Engineering

Content-Based Information Flow Verification for C

Tomasz Maciążek

Kongens Lyngby 2015

DTU Compute
Department of Applied Mathematics and Computer Science
Technical University of Denmark

Richard Petersens Plads, Building 324
2800 Kongens Lyngby
Denmark
Phone: +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Summary

Integration of hardware and software in avionics industry started in 1990s and was followed by increasing interconnectivity with external networks. Since then, the companies producing planes, designed for military and public transport alike, struggled with not only the safety, but also security problems. The certification processes that they were already using for safety had to be extended with security concerns, as these could also impact safety. Concepts such as Integrated Modular Avionics (IMA) and later Multiple Independent Levels of Security (MILS) emerged. They leveraged modularity of previously separate components and proposed solutions for controlling information flow between them. Those solutions involved enforcement of separation and restriction of communication, using *separation kernels* capable of securely partitioning resources, and *secure gateways* that could examine the content of the exchanged messages and filter them.

Although the proposed solutions considerably reduced the costs of certification for those safety-critical systems, the code ensuring separation and filtering was still required to display correctness assurance. This has lead to development of static analysis techniques allowing automatic verification of such code, where the Decentralized Label Model (DLM) seemed to be the most promising tool. Unfortunately, DLM proved to be insufficient, as it cannot be used to provide assurance for a code where the labels (the DLM version of policies) are content-dependent. An obvious solution was to augment DLM with such content-dependent policies, however, this introduced another problem – the necessity to reason about the possible states which the program may be in.

This work builds on the previous works in this topic and introduces a version of DLM with content-dependent policies for a subset of the C programming language. A formal type system, which uses Hoare logic to reason about the state of the program, is built and thoroughly explained on examples. Another product of this work is a C# implementation of a verification tool called C^2if, which is based on that type system. The implementation uses Z3, an SMT solver, as the core for the state- and constraint-based reasoning, and ATLR, a parser generator, as the tool for generating an *abstract syntax tree* (AST) from the input code.

Acknowledgements

First and foremost, I would like to express my gratitude to my supervisor, Professor Hanne Riis Nielson, for giving me an opportunity of working on such interesting and challenging project, for providing brilliant guidance, inspiring ideas and always pointing me in the right direction.

I would like to thank both Professor Hanne Riis Nielson and Professor Flemming Nielson for sharing and explaining drafts of their work on a Content-Dependent Information Flow Control type system. It was an invaluable and solid foundation on which this thesis has been built.

I would also like to thank the rest of the "Airbus Club": Jan Midtgaard, Ximeng Li and Kasper Laursen, for engaging talks during the "Airbus Club" meetings, valuable comments and suggestions.

To Kevin Müller, from the Airbus Group Innovations, I am extremely grateful for explaining usage of MILS and PikeOS in practice, as well as for providing ideas for benchmarks.

Finally, I would like to thank my family and friends for supporting me during the most challenging and stressful periods, and for always believing in my success, giving me the strength when I needed it the most.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.1.1 Certification . . . 2
1.1.2 Integrated Modular Avionics . . . 2
1.1.3 Multiple Independent Levels of Security . . . 2
1.1.4 Secure gateways . . . 3
1.2 The use case scenario . . . 4
1.3 Information flow control . . . 4
1.4 Goals . . . 6

2 Concepts of DLM and Jif **7**

3 Development process **11**
3.1 Methodology . . . 11
3.2 Technologies used . . . 12

4 Non-referential C analysis **13**
4.1 Language specification . . . 13
4.1.1 Syntax . . . 14
4.1.2 Example . . . 15
4.2 DLM labels . . . 17
4.3 Policies . . . 19
4.3.1 Policy syntax . . . 19
4.3.2 Policy semantics . . . 20
4.4 Validation of programs . . . 21
4.4.1 Informal description . . . 22
4.4.2 Type system . . . 23

5 Referential C analysis **33**
5.1 Language specification . . . 33
5.1.1 Syntax . . . 34
5.1.2 Example . . . 35
5.2 Policies . . . 36
5.2.1 Policy syntax . . . 36
5.2.2 Policy semantics . . . 36
5.3 Validation of programs . . . 37
5.3.1 Informal description . . . 37
5.3.2 Type system . . . 37

5.3.3 Tracking values of static arrays 46

6 Implementation 49
6.1 Requirements analysis 49
6.1.1 Functional requirements 49
6.1.2 Non-functional requirements 50
6.2 Architecture overview 50
6.3 Helper classes 52
6.4 Implementation details 55
6.5 Benchmarks 59
6.5.1 Verification of the extended language example 59
6.5.2 Verification of the use case scenario 61
6.5.3 Other examples 63
6.6 Unit tests 66

7 Conclusions 69

Index of notation 75

Index 77

List of Figures

1.1 The demultiplexer use case scenario 5

4.1 Type system for the non-referential language 24
4.2 Axiom for block of atomic assignments 28
4.3 Syntax driven type system for the non-referential language 32

5.1 Skip, declaration and simple assignment axioms 39
5.2 Complex assignment axioms 43
5.3 Composition and control statement judgements 45

6.1 Overview of the architecture 51
6.2 Helper classes 53
6.3 Interfaces and abstract classes 56
6.4 The architecture of labels 58

Listings

4.1 Example usage of policies in the simple language 16
4.2 Example labeling 18
4.3 Example policy specification 19
4.4 Example of a non-self-influencing assignment 25
4.5 Example of leak if self-assignments are unrestricted 26
4.6 Example of partial structure update problem 27
4.7 Example of structure initialization policy erasure problem 27
4.8 Example of sequential assignment accumulation problem 29
4.9 Example of multiple assignment problem 30
5.1 Example usage of policies in the extended language 35
5.2 Example policy specification 36
5.3 Example of sub-component pointer assignment problem 44
6.1 Output of the tool for flow validation failure 60
6.2 Output of the tool for invariant validation failure 61
6.3 The use case scenario code 62
6.4 The multiplexer – the reversed use case scenario code 62
6.5 Example of wrong structure initialization 63
6.6 Output for wrong structure initialization example 64
6.7 Example of array subscripts influence 64
6.8 Output for the array subscripts influence example 65
6.9 Output for the partially corrected array subscripts influence example 65
6.10 Example of volatility of pointers 66
6.11 Output for the volatility of pointers example 67

Chapter 1

Introduction

The main purpose of this work is to provide a proof of concept showing that content-based verification of information flow security can be done even for languages, such as C, that allow complex data structures, pointers and arrays. Another purpose is to create a tool that implements such verification and is be able to process simple programs. Thus, the product of this thesis is a type system displaying the theory behind the verification of information flow security in programs written in a modified subset of C (*C11* standard), as well as its implementation.

In this chapter I will first introduce the background and motivation behind the topic of this thesis, and shortly describe the development of the information flow analysis that has been the foundation of the solutions presented in the next chapters. In the end I will precisely state the goals that this work shall achieve.

1.1 Background

Security has become an important factor in industries where it previously was not concerned. Industries dealing with safety-critical systems such as avionics, automotive and railways started to integrate their systems into common hardware platforms and connected them to the Internet. There were multiple reasons for this development. The most important one was to reduce the costs by minimising amount of separate controllers, each of which needs maintenance and power supply. Software controllers, running on single hardware platform would require less maintenance and power. In avionics there was a secondary objective reached by this minimisation – reduction of mass and volume taken by those controllers, which neither is without an impact on the costs and revenue. Another purpose of the development was desire to remain competitive on the market were customer orientation matters the most. This has lead to necessity to provide the customer (the passenger) with entertainment and live or even interactive information about the journey.

As the integration was introduced, suddenly these industries had to face the same security threats as those faced by the IT industry. Unfortunately, some of the security threats, could have repercussions in safety, which is the main concern of the companies producing safety-critical systems. The concept

of architectural and hardware integration, including all the security and safety concerns associated with it, has been called Integrated Modular Avionics (IMA).

Safety and security issues are particularly important in avionics, because their systems cannot be simply shut down in case of a major failure, and also because their faulty operation may cause the most severe losses. That is why the case of avionics has been chosen as the focus point – the need for secure information flow is the highest in this industry.

1.1.1 Certification

In order for the systems to be regarded as safe or secure they need to undergo a standardized certification process. In terms of computer security one of the most widely used certification standards is the Common Criteria [9]. As a result of the process the system is assigned with a grade representing the level of assurance. In case of Common Criteria it is one of seven (1-7) Evaluation Assurance Levels (EAL). The certification process consumes a lot of resources, and the system in question must be methodically, semiformally or even formally designed and tested in order to reach the desired assurance level.

There exist certain models and guidelines that facilitate creation of safe and secure systems. The ARINC Report 811 [8] (briefly described in [27]) provides general security guidance on development of aircraft information systems, indicating how to work under the constrained environment of avionics and how to match security measures to threats based on risk analysis. The report describes security domains of the aircraft and its life-cycle with respect to security issues. Moreover, it specifies a security nomenclature, and gives a general *information security process framework* that can be used on top of security standards (such as Common Criteria), and is specifically designed for avionics.

1.1.2 Integrated Modular Avionics

The idea behind IMA is also to support security of the systems on-board the aircrafts, however, in this case their modularity is utilized. Before the integration happened the aircraft controllers were working as standalone systems. The IMA concept tries to maintain that separation by introduction of *partitions* – isolated processing and system (e.g. memory, I/O) resources being part of the same hardware platform. The partitions can communicate with each other using *ports* – directed channels which exist between the partitions that are supposed to exchange information. Finally, the ports need to specify policies controlling information that can be passed through them, particularly when they cross security domains. Those policies are enforced by *secure gateways*.

Thanks to partitions, applications/controllers that belong to different security domains can share the resources with minimal interference (on each other). Still, a strong assurance needs to be provided that all policies are satisfied and that the subsystems are otherwise truly separated.

1.1.3 Multiple Independent Levels of Security

Multiple Independent Levels of Security (MILS) is a high-assurance architectural approach to security problems. It is not a system architecture itself, however, it shows how to design, construct, integrate and evaluate secure systems,

and thereby reduce assurance costs. It is a slightly newer idea, in comparison to IMA, introduced in [31]. A two level approach is advocated by MILS that separates the problem of enforcing a security policy from secure sharing of resources.

According to the MILS constitution, one must first devise logically decomposed components of which the system consists. These should be as simple as possible, preferably performing just one function and behave as standalone systems – it is a "divide and conquer" approach. The components may be deemed either as trusted or untrusted. All parts of trusted components, even the operating system (OS), must display a certain level of assurance. Similarly to the IMA concept, the components can only communicate with each other via dedicated, unidirectional ports.

The next step happens at the resource sharing level. The components of the security architecture should be mapped to resources. During this process a special care should be taken to cluster components that share similar functionality or are physically collocated. The resources are shared securely, even when trusted and untrusted components are mixed, thanks to partitioning – the same kind of separation as in IMA.

The partitioning is governed by *separation kernels* – very small (few to tens of thousands of lines of code) security kernels that constrain programs *spatially* (memory), *temporally* (processor time) or *cryptographically* (filesystems or communications). Separation kernels also provide *inter-partition communication* (IPC) channels, that allow partitions to communicate with each other. Those channels are also guarded by *secure gateways*.

MILS facilitates certification by taking advantage of compositionality of the architecture that it proposes. There are two main certification approaches for MILS, as suggested in [20]. According to the Composed Assurance Package (CAP) approach, only the dependent component (such as the secure gateway) has to be evaluated in-depth, provided that the component that it depends on (e.g. the separation kernel) has displayed relevant assurance separately. This kind of evaluation can reach up to EAL 4. On the other hand, the Common Criteria Development Board (CCDB) approach can provide assurance beyond that level, however, it restricts the system under evaluation to purely hierarchical compositional architecture, so that there can only be application–platform dependencies. Furthermore, the application has to be evaluated jointly with the platform, which increases the costs of certification.

One of the operating systems based on the MILS concept that features a separation kernel is PikeOS [5]. It is a real-time OS, that is certified according to a number of standards and thus appropriate for safety-critical systems. It is, among other applications, used by *Airbus* – a consortium producing large variety of aircrafts.

1.1.4 Secure gateways

In [21], Müller et al. introduce a detailed specification and architecture of a secure gateway suited for the avionics industry. There are several requirements noted in this article that the secure gateway should meet. The most important is that it should allow content-based flow control, that is examine the content of the

messages, not only determine which partitions should be able to communicate with each other.

The secure gateway described in [21] uses two partitions, which effectively separates it into an outbound (egress) and an inbound (ingress) part. The outbound part guards against information leakage (ensures confidentiality), while the inbound part is used to protect data integrity. Each of those parts consists of several modules: net module, routing module (ingress only), viewer module and border-crossing. The net module operates in the application layer and checks the semantic integrity of the messages. The routing module decides to which application should the message be routed. The viewer module contains filters through which each message is iterated and then it is forwarded to the next module only if it has passed all the filters. Finally, the border-crossing, which is the lowest level module, uses the IPC channels of the separation kernel to send the message to a border-crossing module of some other domain.

1.2 The use case scenario

Now that we are familiar with the background of the security issues and solutions in avionics, I shall introduce the use case scenario that is the motivation of this work. The previous section raised the subject of the secure gateways, which are an intrinsic part of security architecture in both IMA and MILS approaches. Parts of those gateways, such as filters, content-based policies and routers are external to the well established and verified secure systems (like PikeOS), and need to be additionally verified. These parts may consist of relatively small pieces of code that could be verified using static program analysis.

Such scenario has been illustrated in [19] – an article by Müller et al., where a problem of routing to the appropriate transport layer protocol decoder is raised. In this article a *demultiplexer* module is responsible for making the right choice based on the content of the inbound messages, and the code of that module is provided. The article states that it should be possible to devise some policies for the program and then using static analysis ensure that its implementation is correct with respect to information flow security – that both TCP and UDP packets are correctly, respectively forwarded to the TCP and UDP modules. Figure 1.1 depicts that use case scenario in a similar way to how [19] does.

Müller et al. find that in order to automatically verify their program, without changing its structure, the policies themselves would need to be content-dependent. Otherwise, the loops that iterate through variables of disjunctive policy nature need to be transformed into *switch* statements with excessive use of *downgrading*. They propose how those policies should look like on example of the Decentralized Label Model (DLM).

1.3 Information flow control

The choice of DLM for the problem of expressing and validating security policies in the use case scenario was not accidental. In history of information flow control there have been several major milestones.

The fundamentals were given by the specification of confidentiality lattices incorporating both levels and domains of security introduced by Bell and

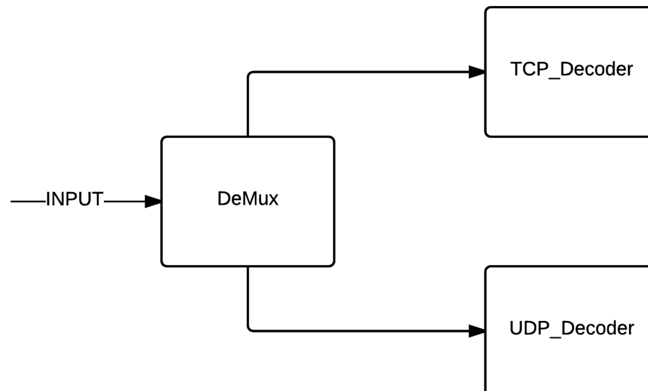

Figure 1.1: The demultiplexer use case scenario

LaPadula in [13], further refined by Biba with model of integrity in [14]. It was then used in Denning and Denning's work [16] providing an approach for certification of information flow security in programs. Their work was then formalized into a type system and proved to be correct by Volpano et al. [32].

Unfortunately those approaches were too restrictive to be used in practice. The main problem was lack of possibility to bypass the security policies (in a controlled manner) in order for the program to actually do something useful. This problem has been solved in the model proposed by Myers et al. [24] – the Decentralized Label Model – originally considering only confidentiality, later augmented with integrity in [25]. DLM offers means of declassifying (downgrading confidentiality) and endorsing (downgrading integrity) data and implicit information flows in a controlled manner. Furthermore, it introduces a notion of principals, which can act as owners, readers and writers of some data. Those principals may be a perfect representation of the security domains in avionics. The security policies in DLM take form of *labels*, which consist of owners, as well as readers and/or writers that each of them declares. DLM will be explained in details later on in this work.

Another advantage of DLM is that it has already been implemented for Java programs as *Jif: Java + information flow* [4]. Although it is C, not Java, that is used as the programming language in avionics (due to numerous challenges in assurance), Jif documentation provides a reference frame on which the specification of a solution for C can be based. Thanks to the fact that both Java and C are imperative languages, which principles are similar, much of the Jif functionality can be directly translated for C.

Unfortunately, although DLM appears to be perfect for the task it still has some limitations. As mentioned before, the DLM labels are not content-dependent, which poses a problem in case some data or channels have a disjunctive policy nature. A solution to this, for a simple language of concurrent processes and set-based policies, has been proposed by Nielson et al. in [30]. This article describes a way of specifying content-dependent policies and provides a type system that can verify correctness of programs against those policies. This

approach is later refined in [29], featuring more DLM-like policies, and is the basis of the type system presented in this work.

1.4 Goals

Based on the needs in the avionics industry and the aforementioned use case scenario, as well as the shortcomings of DLM and the previous work in attempt to overcome these, the goals of this thesis are:

1. Describe a syntax using which content-dependent policies can be specified for C. The syntax should be as simple as possible to facilitate their creation and understanding for the programmers that are going to use them. It should be based on the syntax provided in [19].

2. Construct a type system capable of verifying programs written in a subset of C. The subset should encompass simple variables, arrays, pointers, structures, declarations, definitions, assignments, arithmetic and boolean operations, as well as basic control statements: *if* conditionals and *while* loops. The language should be augmented with policies and constructs necessary to perform the verification.

3. Provide a verification tool that accepts as an input the code of programs written in the defined language, and validates the information flow security within them. In the output the tool should either indicate a success or provide information useful in finding the problem with the code. The tool shall be named C^2if , which stands for C Content-based Information Flow.

4. Identify the challenges in the content-dependent information flow security analysis and point the directions for the future development.

Chapter 2

Concepts of DLM and Jif

Jif is a well-established implementation of DLM for Java. It provides a syntax and a compiler that together allow creating software where the information flow security has the highest priority. It has originated from JFlow [23] (an early implementation of DLM on a Java-like language), and has developed since – now it also incorporates integrity labelling and many other features that facilitate secure programming and strengthen security. As C is also an imperative language, and its features are mostly a subset of what Java provides, an opportunity emerges to re-use some of Jif components and reasoning.

In this chapter I will describe those components in details and analyse their usefulness in relation to the use case scenario and the avionics domain in general. Based on that and on their complexity, I will determine which of these components are going to be implemented in the solution presented in this work.

Principals

The core of the security properties and analysis of Jif (and DLM) are principals – abstract actors in the system which are owners of data and on behalf of which programs are executed. Owners can declare authority to read or write their data for other principals, which are therefore called readers and writers. Jif goes even further and allows principals to delegate full authority to act for them, thus constructing a principal hierarchy. In the avionics scenario, the principals can represent different MILS components, or even ARINC 811 security domains, depending on the choice of granularity. Nevertheless, they do not need to be allowed to act for each other, as each component should maintain control over its own information. Furthermore, the domains and the components are not hierarchical with respect to security. That is why the concept of principals can be certainly re-used, not including, however, the hierarchy.

Confidentiality and integrity labels

Jif provides syntax of labels that capture both confidentiality and integrity properties of information flow. The above-mentioned principals are the first class citizens (the only ones) of the labels. DLM defines those labels using the principal hierarchy concept that I have already decided to reject, which means that re-using labels will require their redefinition.

Both confidentiality and integrity labels are crucial for avionics, as the information may flow in both directions between domains of different levels of security. Moreover, these aspects of security are dual in their nature and they display similar partial ordering properties. Hence, both data confidentiality and integrity will be included in the analysis presented in this work.

Explicit and implicit flow

Just as in classic information flow security approaches, DLM separates information flow into two classes – explicit and implicit. The explicit information flow normally occurs while assigning to a variable or writing data to a channel, while implicit flow occurs when the explicit flow depends on some condition, which happens, amongst others, inside conditional statements and loops. The implicit flow can be also defined as a flow that does not appear in all possible execution paths.

For capturing and analysing security of explicit flows, DLM attaches labels to all variables in the program. As for the implicit flows, it deduces a label for each statement in the code, or in other words for the program counter (PC). That label is a combination of the labels of variables on which the execution of that statement depends. Those concepts are the very core to the C^2if implementation and can be re-used with little or no changes.

Methods

Information flow security analysis for methods is not trivial, as these may have side-effects and multiple ways of termination (e.g. exceptions). Both those complexities have been solved in Jif, by begin and end labels of methods respectively. The begin label influences the program counter label inside the body of the method, while the end label changes the program counter label after the method call. As Java methods are very similar to C functions it should not be difficult to re-use Jif solutions. However, due to the complexity of other elements of C considered in this work, implementation of functions is left for future work.

Label inference, default labels and polymorphism

Jif provides some features that ease the burden of security annotation of code that is put on the programmer. The first feature, label inference, allows to omit labels in variable declarations if these are unimportant, and then those labels will be inferred by the Jif compiler to fit to other labels in the program. The second – default labels – allows to omit labels in other places, such as begin labels of methods, which are then assigned a default value by the compiler (in case of begin labels – the most restrictive). Finally, polymorphism allows not writing labels of method arguments and return values, which are then instantiated with the labels at the call place.

Although these features are very useful they do not introduce better expressive power to the language. They are, however, desired by the industry, as they minimize changes to the code required to verify it. Hence, these features should be implemented in the future.

Arrays

In Jif arrays have two labels – for the array elements and for the variable that points to the array itself. Arrays exist both in Java and in C in virtually the same form, which is why incorporating that part of Jif in this work is undoubtedly justified. Furthermore, some parts of analysis concerning arrays can be helpful in analysing information flows related to pointers.

Exceptions

Exceptions change the normal execution paths of programs which is why these have to be considered in information flow analysis. In Jif the problem of exceptions is solved by changing the PC label after any statement that can produce an exception and augmenting the PC label inside exception handlers. C does not have any concept of exceptions, but a program may be terminated at any moment unexpectedly or using the *exit()* directive. The former case may be very elusive because there are many reasons for which the program could stop that are hard to analyse. The latter is easy to reason about, however, some information flow that it may cause is not explicit – it may be used to create a timing covert channel. Neither of the two cases is going to be analysed in-depth in this work.

Dynamic and runtime labels and principals

Jif introduces labels as first class citizens of the language and allows evaluating them in run-time. This is particularly useful when the system is heterogeneous with respect to its principals, that is, there are types of principals which can be represented by many individuals (actors) and each of them needs a separate principal instance. In the use case scenario we do not need that feature, as the set of principals is limited to a finite set of protocols. As for other cases, even though principals could be heterogeneous (e.g. passengers), in a system where their safety is the most important factor, also for certification, that fact could be neglected and they could be aggregated into a single domain principal.

Downgrading

The most powerful and innovative features of DLM are declassification and endorsement, which allow controlled information flow that is not restriction in terms of confidentiality and integrity respectively. In Jif both variables (explicit flow) and the PC (implicit flow) can be declassified/endorsed. Although those features are actually the most important part of DLM they are not needed for the demultiplexer module scenario. The reason is that the purpose of this module is only to pass data that it receives correctly according to some policies. Those policies establish the labels of the variables that carry the data and these labels should not be changed in any place of the code of the demultiplexer. However, downgrading may be useful in the code of the components that communicate via the module, and hence, in order to provide a versatile tool for static analysis of information flow, it should be also implemented. Nevertheless, due to complexity introduced by the content-dependent policies and static reasoning about values, this aspect of DLM is left out as the future work.

Chapter 3

Development process

Development of a type system capable of handling content-dependent policies and complexities of the C language is not an easy task. There are many pitfalls and if errors are made in the type system then it becomes useless for verification of information flow security. Furthermore, the choice of technology is also important. It not only influences the pace of the development process, but also the potential and performance of the product. This chapter discusses both the methodology of development and the technologies used for implementation.

3.1 Methodology

As mentioned before, the solution has been based on DLM and its implementation, Jif. In particular, labels and most of their semantics have been preserved. This allowed their implementation in this solution to be benchmarked against Jif in order to ensure correctness.

Furthermore, the development of the type system has been divided into two major milestones, which is also reflected in the structure of this thesis. The first, the *non-referential C analysis*, covers all parts of the desired C syntax except for pointers and arrays. Although arrays may be static and do not have to store addresses of other variables, as pointers do, pointers and arrays have so much in common in C that it would be hard to separate them. This, and the complexity associated with storing many values under one variable, is why arrays are not part of the first milestone. The second milestone, the *referential C analysis*, extends the first one with the above-mentioned arrays and pointers. Thanks to this approach the development process became less challenging, because the first milestone provided a strong foundation for introducing the second, which is much more complicated and error prone. Another advantage is that the type system, being gradually introduced, is easier to comprehend.

Finally, each milestone has been developed in smaller iterations, each introducing just a few functionalities. Whenever a functionality was theoretically described, it was then implemented and thoroughly tested. The testing has not only ensured correctness of the implementation, but on numerous occasions it has revealed problems with the theory behind it.

3.2 Technologies used

The first choice that I had to make, as far as the technologies are concerned, was the choice of an SMT solver. A solver of that kind is necessary to perform a Hoare logic driven reasoning about possible states of the program and use this information in verification of policies in a way that is similar to what is presented in [29]. Programs contain arithmetic and boolean expressions that need to be interpreted, which is why a SAT solver would not be enough. There are tens of SMT solvers available and free to use. The three most versatile and well known are Z3 [7], CVC4 [2] and Yices2 [6]. There are no significant differences in what kind of problems any of those three SMT solvers can process. All three are also able to accept input in the SMT-LIB 2.0 format [12]. Yices2, however, provides an API only for C – a very low level language, and thus, unsuitable for the task. As for the other two solvers, they provide well documented, native APIs for high level languages such as Java (CVC4) and C# (Z3). They both also perform very well, which has been proven in an international SMT solvers competition [1]. Hence, the choice of the solver had to be determined by the choice of the programming language.

As mentioned above, the two options for the programming language of the implementation were Java and C#. Both are high level languages with little or no difference in the performance, similar capabilities and testing possibilities. The most recent, eight, version of Java, also supports lambda expressions that are very useful in processing all sorts of collections, and which have been available in C# for a long time. Now, one of the very few differences is that C# has slightly more of "syntactic sugar", which makes the development process easier and more enjoyable. Thus, I have chosen C# – it has been a quite personal choice in the end.

Of course, choice of the programming language was not without influence on the choice of the *Integrated Development Environment* (IDE), and its implementation that provides the class library. For C# the obvious option was *Visual Studio* with the implementation of the .Net Framework – an IDE developed by Microsoft tailored for development of C# programs.

The last choice, as far as the technologies are concerned, was the parser generator. Such tool generates a parser given a formal description of the language of the input and rules for creating an *abstract syntax tree* (AST). Since I only had (a good) experience with ANTLR, it has been my first choice. Nevertheless, I conducted a quick research in the field of parsers and found that the second most recommended generator is GOLD [3]. It uses different grammar notation and parsing algorithm than ANTLR. The notation in GOLD is BNF and the algorithm is LALR, while for ANTLR it is EBNF and LL respectively. This amounts to the fact that writing grammars in ANTLR is less error prone (nondeterministic grammars are impossible) and it is easier to express repetitions and optional occurrences, which confirmed my choice.

To summarize, here is the list of the technologies used in this work, with their versions:

Z3 v4.4.0 – an SMT solver.

C# v5.0 – a high level programming language.

.NET Framework v4.5 – a software framework providing the class library.

ANTLR v4.3.0 – a parser generator.

Chapter 4

Non-referential C analysis

In this chapter I will introduce a type system for validation of information flow security for programs written in a small subset of the C language. I will start with specification of the language and provide its syntax with emphasis on where the security policies are bound to the data structures. Then, I will introduce a version and interpretation of the DLM labels that is used in those policies. After that, I will present the concept of the content-dependent policies, in the context of this work, their syntax and semantics. Next, I will informally describe the idea behind verification in that language and finally propose a formal type system, according to which the C^2if validation tool is implemented.

4.1 Language specification

As mentioned in the introduction, the language considered in this first milestone is going to be a small subset of C containing the following:

- Integer, decimal (float) and boolean variables
- Structures and structure initialization
- Declarations, definitions and assignments
- Arithmetic and boolean operations
- Conditional statements (*if* conditionals)
- Simple iteration statements (*while* loops)

The language, as it is defined, will serve as a proof of concept and will be built upon in the subsequent milestone presented in the next chapter. Using the aforementioned components the solution will be able to address a simplified version of the previously defined use case scenario.

4.1.1 Syntax

The following provides the syntax of this simple language:

$$
\begin{aligned}
t &::= \texttt{int} \mid \texttt{float} \mid \texttt{bool} \\
lt &::= t\ \texttt{\{}policy\texttt{\}} \\
stc &::= \texttt{struct}\ s\ \texttt{\{}decls\texttt{\}}\ \texttt{\{}policy\texttt{\}}; \\
stci &::= \texttt{struct}\ s\ \texttt{\{}decls\texttt{\}}\ \texttt{\{}policy\texttt{\}}\ si \mid \texttt{struct}\ s\ \texttt{\{}policy\texttt{\}}\ si; \\
decls &::= decl \mid decls_1\ decls_2 \\
decl &::= lt_a\ v_a; \mid lt_b\ v_b; \mid stci; \\
decli &::= lt_a\ v_a\texttt{=}a; \mid lt_b\ v_b\texttt{=}b; \\
a &::= n \mid x_a \mid a_1\ op_a\ a_2 \\
b &::= \texttt{true} \mid \texttt{false} \mid \texttt{!}b \mid x_b \mid b_1\ op_b\ b_2 \mid a_1\ op_r\ a_2 \mid e_1\ \texttt{==}\ e_2 \\
e &::= a \mid b \\
x_a &::= si.x_a \mid v_a \\
x_b &::= si.x_b \mid v_b \\
xv &::= x_a \mid x_b \\
xs &::= si.xs \mid si \\
init &::= \{inlst\} \\
inlst &::= e \mid inlst_1, inlst_2 \mid init \\
stinit &::= stci\texttt{=}init; \\
assign &::= x_b\texttt{=}b; \mid x_a\texttt{=}a; \mid xs_t\texttt{=}xs_s \\
S &::= stc \mid decl \mid decli \mid stinit \mid \\
&\quad ; \mid assign \mid \texttt{if (}b\texttt{) \{}S\texttt{\}} \mid S_1\ S_2 \mid \\
&\quad \texttt{if (}b\texttt{) \{}S_1\texttt{\} else \{}S_2\texttt{\}} \mid \texttt{while (}b\texttt{) \{}S\texttt{\}} \mid \\
&\quad \texttt{while (}b\texttt{)[}\psi\texttt{] \{}S\texttt{\}}
\end{aligned}
$$

The syntax contains an undefined rule *policy* which will be defined later in section 4.3, as it requires that the DLM labels are introduced first. Apart from that, there are a few other symbols that need to be clarified. The symbol n stands for an integer or a floating point constant, xv and xs are slots (identifiers of places in program's memory), a means an arithmetic expression and b means a boolean expression, while op_a and op_b are respective operators. There is a limited type-checking that is provided with subscripts of slots and variables.

The slots are defined using a recursive rule over the accessors (dot separated structure–component sequences) that are used to build the *fully qualified name* of the slot. In this syntax, the final component of an xv slot is always a simple type variable identifier, here denoted by v, while for xs it is always a structure instance identifier – si.

For the purpose of this work, data structures having simple type (can also be components of structures) will be referred to as *variables*. When denoted with v their short (local if in structure) name is considered, while xv will refer to their fully qualified name – with dot-separated parent structures if applicable. An analogous rule applies to the structure instances.

According to this specification, the language gives raise to several types, or in other words domains, of slots. The **Var** domain consists of variables, while structure instances belong to the **Str** domain. Finally, these altogether constitute the domain of all slots **Slot** = **Var** $\cup$ **Str**, which will be symbolized by x (without subscripts).

The policies can be specified for variables, structures and their and instances. The scope of the policies (the slots on which they are dependent and which they influence) is limited by the point of attachment. In case of a variable, the scope is that variable itself. As for a structure, it is all its components, and subcomponents if the structure contains nested structures. The programmer may attach the policy to both the structure and its components, so that he has macro- and micro-control over the policies.

Declarations are split into pure ones and those with initialization. The structures are also split into pure declaration of the type and instantiation, where s is the name of the structure and si is the name of the structure instance. Also note that the programmer can attach the policy to both, the declaration of type and instantiation, however, not when declaring and instantiating the structure at the same time – there is no reason for splitting the policy definition in that case.

The rule *stinit* refers to aggregate structure initialization as defined in the C standard [10]. It is necessary to introduce structure initialization of this kind from the information flow security analysis point of view, as explained in section 5.3.2. It virtually allows grouping assignments together and initializing a structure atomically without violating the content-dependent policies.

The rule for the *while* loop has a version where an additional parameter ψ is specified. It is a *loop invariant* that the programmer may specify. It has been introduced in order to avoid fixed-point analysis, which would be otherwise necessary to reason about the state inside loops.

4.1.2 Example

Listing 4.1 presents an example program written in the language defined in this chapter. This code snippet is interesting, as it shows a scenario where if the policies were not used then declassification would be necessary in order to make this simple program correct with respect to information flow security. Here we have a structure `x` that has a policy depending on its value. Another two variables `y` and `z` have static labels each corresponding to one of the `x`'s data labels. The structure is initialised with some input in an initializer list that also sets the determinant accordingly. The *if* conditionals determine whether `x.data` should be assigned to `y` or `z`. Thanks to that, the data will be securely assigned to `y` and the program will be successfully validated. Note that label of the `determinant` is just unrestrictive enough to allow assignment in both conditionals.

```
int {{Alice->Bob}} y;
int {{Alice->Chuck}} z;

int {{Alice->Bob}} input;
struct s {
        int {{Alice->Bob,Chuck}} determinant;
        int data;
}{
        (self.determinant == 1 => self.data={Alice->Bob});
        (self.determinant == 2 => self.data={Alice->Chuck})
} x = {
        1,
        input
};
if(x.determinant == 1) {
        y = x.data;
}
if(x.determinant == 2) {
        z = x.data;
}
```

Listing 4.1: Example usage of policies in the simple language

4.2 DLM labels

As mentioned in the introduction, a DLM label consists of confidentiality and integrity parts. Here, I will describe and re-define the labels following the specification from [22], however, not using the principal hierarchy.

Labels have the form $\{O_1 \rightarrow R_1; ...; O_n \rightarrow R_n; O_1 \leftarrow W_1; ...; O_n \leftarrow W_n\}$. O_i is a set representing *owners*, however, in policy specification (for simplicity) it can be only either a singleton (one owner), or a set of all principals (denoted `*`), or an empty set (denoted `_`). R_i is a comma separated set of *readers* designated by the owners, and W_i is the set of writers who the owners believe that might have influenced the data. All three, owners, readers and writers are *principals*, that is, entities that can perform some actions in the system. The labels are partially ordered by the $\sqsubseteq$ relation, which is defined as follows:

$$\begin{aligned} L_1 \sqsubseteq L_2 \text{ iff } \forall_p :\ & \mathsf{readers}(L_1, p) \supseteq \mathsf{readers}(L_2, p) \\ & \wedge \mathsf{writers}(L_1, p) \subseteq \mathsf{writers}(L_2, p) \end{aligned}$$

where p is a principal, while $\mathsf{readers}(L, p)$ and $\mathsf{writers}(L, p)$ are defined in the following manner:

$$\begin{aligned} \mathsf{readers}(O \rightarrow R, p) &= \begin{cases} p \cup R & \text{iff } p \in O \\ \texttt{*} & \text{otherwise} \end{cases} \\ \mathsf{readers}(L_1; L_2, p) &= \mathsf{readers}(L_1, p) \cap \mathsf{readers}(L_1, p) \\ \mathsf{writers}(O \leftarrow W, p) &= \begin{cases} p \cup W & \text{iff } p \in O \\ \texttt{_} & \text{otherwise} \end{cases} \\ \mathsf{writers}(L_1; L_2, p) &= \mathsf{writers}(L_1, p) \cup \mathsf{writers}(L_1, p) \end{aligned}$$

In other words, $\mathsf{readers}(L, p)$ is the set of readers designated by p in label L with p itself included; which means all the readers that p allows to read the data. If p is not in the label it allows reading for all principals by default. The $\mathsf{writers}(L, p)$ is the set of writers designated by p in label L and p itself included here as well; which means all the writers that p believes that may have influenced the data. If p is not in the label it by default believes that no one has influenced the data.

That definition of the DLM labels gives rise to top $\top$ and bottom $\perp$ labels of the label partial ordering:

$$\begin{aligned} \top &= \{\texttt{*->_;*<-*}\} \\ \perp &= \{\texttt{_->}\cdot\texttt{;_<-}\cdot\} \end{aligned}$$

The top label means that all principals restrict the read operation to themselves, and all principals believe that anyone could have influenced the data. The bottom label indicates that no principal imposes any restriction for reading, and no principal believes that the data have been influenced by anyone. The dot symbol $\cdot$ expresses that it can be substituted with anything, and this follows from the fact that no principal can be matched with the empty owner set `_`.

An example of labelling is presented in listing 4.2. Here, the part of label with the right-arrow (`->`) concerns the confidentiality and the part with the

```
int {Alice->Bob; Bob<-_} x;
int {Alice->_; *<-*} y;
```

Listing 4.2: Example labeling

left-arrow (`<-`) concerns integrity. In the example, *Bob* is allowed to read the variable `x` by *Alice* and *Alice* is also an implicit reader. *Bob* also believes that no one (but him) has influenced the variable. As for `y`, *Alice* does not allow anyone (but her) to read it, however, everyone believes that anyone might have influenced the data.

4.3 Policies

As mentioned before, we would like to support the demultiplexer scenario, without using downgrading, so that everything can be checked statically and better security proof is provided. In order to do that, the DLM labels attached to slots need to be dependent on their values, or values of some other slots in the code.

4.3.1 Policy syntax

Providing dependence of labels on values can be accomplished by introducing *policies* that will dictate the labels of slots to which these are attached based on some conditions. Such policies with their syntax and semantics have been introduced in [19]. In this work, I am going to consider a version of policies provided in that paper:

$$\begin{aligned}
policyDef &::= \texttt{policy}\ \ \mathit{name}\texttt{=}\{\mathit{policy}\} \\
policyUse &::= \{\mathit{policy}\} \\
\mathit{policy} &::= \mathit{name} \mid \mathit{policy}\texttt{;}\ \ \mathit{policy} \mid (\mathit{condition}\texttt{=>}\mathit{policy}) \mid \mathit{result} \\
\mathit{condition} &::= \mathit{slot}\texttt{==}\mathit{value} \mid \mathit{condition}\ \ \texttt{\&\&}\ \ \mathit{condition} \mid \mathit{condition}\ \ \texttt{||}\ \ \mathit{condition} \\
\mathit{result} &::= \mathit{result}\ \ \texttt{\&\&}\ \ \mathit{result} \mid \mathit{slot}\texttt{=}\mathit{DLMLabel} \mid \mathit{DLMLabel} \\
\mathit{slot} &::= \texttt{self} \mid \mathit{slot.component}
\end{aligned}$$

In this syntax, the policies are allowed to specify *conditions* on the slot, and if these are met then some other policy is applied, which might actually be a *result* that resolves to appropriate labelling (with a DLM *label*) of some slot. If no condition is specified (only the *result*) then it is assumed to be equivalent to `true` and the policy always holds. Note that the slot to which the policy is applied restricts the scope of conditions and results to that slot, and its components (if applicable).

There can be several results of a policy, for example, concerning different slots. If no slot is specified then by default it is the slot to which the policy is applied that is concerned (i.e. `self`).

Policies can be declared separately, named and then assigned to slots by providing the policy name inside curly brackets as shown in the *policyUse* rule. It is also possible to in-line the policy specification at the place where it is used. More than one policy can be specified using semicolon as a separator.

An example of a simple policy specification is shown in listing 4.3.

```
1 int {{Alice->Bob; Bob<-_}} x;
2 int {(self == 2 => {Alice->_; *<-*})} y;
```

Listing 4.3: Example policy specification

Here, we have an unconditional policy for `x` that effectively is equivalent to the simple DLM label described in section 4.2. As for `y`, it is governed by a policy that assigns the specified label to it only if its value is equal to 2. In other cases the label of `y` is $\bot$.

4.3.2 Policy semantics

So far we have seen the syntax of the policies that will be used in code. However, in order to make use of the policies in the type system, their semantics need to be defined. The policies defined previously in the syntax result semantically in a global policy defined as follows:

$$\begin{aligned}
\texttt{P} &::= X : L \\
&\mid \phi \Rightarrow \texttt{P} \\
&\mid \texttt{P}_1 ; \texttt{P}_2 \\
\phi &::= x = n \\
&\mid \phi_1 \wedge \phi_2 \\
&\mid \phi_1 \vee \phi_2
\end{aligned}$$

where X is a set of slots initially containing exactly one slot, L is the previously defined DLM label, ϕ is a condition, x is a slot, and n is a constant. The use of a set of slots might seem excessive here, but it will be necessary for the type system defined later on.

The translation from the policy syntax to the semantics is given by the following set of translation functions:

$$\begin{aligned}
T_{P_{def}}(\texttt{policy}\ name = \{policy\}, \mathcal{P}) &= \mathcal{P}[name \mapsto \lambda x.T_P(policy, \mathcal{P}, x)] \\
T_{S_{def}}(\texttt{struct}\ s\{decls\}\ \{policy\}, \mathcal{P}) &= \mathcal{P}[s \mapsto \lambda x.(T_P(policy, \mathcal{P}, x); T_{decl}(decls, \mathcal{P}, x))] \\
T_{def}(\texttt{type}\ x\ \{policy\}, \texttt{P}, \mathcal{P}) &= \texttt{P}\texttt{;}T_P(policy, \mathcal{P}, x) \\
T_{def}(\texttt{struct}\ s\{decls\}\ \{policy\}\ si, \texttt{P}, \mathcal{P}) &= \texttt{P}\texttt{;}T_P(policy, \mathcal{P}, si)\texttt{;}T_{decl}(decls, \mathcal{P}, si) \\
T_{def}(\texttt{struct}\ s\{policy\}\ si, \texttt{P}, \mathcal{P}) &= \texttt{P}\texttt{;}T_P(policy, \mathcal{P}, si)\texttt{;}\mathcal{P}[s](si) \\
T_{decl}(\texttt{type}\ var\ \{policy\}, \mathcal{P}, x) &= T_P(policy, \mathcal{P}, x.var) \\
T_{decl}(\texttt{struct}\ s\{decls\}\ \{policy\}\ si, \mathcal{P}, x) &= T_P(policy, \mathcal{P}, x.si)\texttt{;}T_{decl}(decls, \mathcal{P}, x.si) \\
T_{decl}(\texttt{struct}\ s\{policy\}\ si, \mathcal{P}, x) &= T_P(policy, \mathcal{P}, x.si)\texttt{;}\mathcal{P}[s](x.si) \\
T_{decl}(decls_1\ decls_2, \mathcal{P}, x) &= T_{decl}(decls_1, \mathcal{P}, x)\texttt{;}T_{decl}(decls_2, \mathcal{P}, x) \\
T_P(name, \mathcal{P}, x) &= \mathcal{P}[name](x) \\
T_P(policy_1;\ policy_2, \mathcal{P}, x) &= T_P(policy_1, x)\texttt{;}T_P(policy_2, x) \\
T_P(condition\texttt{=>}policy, \mathcal{P}, x) &= T_C(condition, x) \Rightarrow T_P(policy, \mathcal{P}, x) \\
T_P(result, \mathcal{P}, x) &= T_R(result, x) \\
T_C(condition_1\ \texttt{||}\ condition_2, x) &= T_C(condition_1, x) \vee T_C(condition_2, x) \\
T_C(condition_1\ \texttt{\&\&}\ condition_2, x) &= T_C(condition_1, x) \wedge T_C(condition_2, x) \\
T_C(slot\texttt{==}value, x) &= T_S(slot, x) = value \\
T_R(result_1\ \texttt{\&\&}\ result_2, x) &= T_R(result_1, x)\texttt{;}T_R(result_2, x) \\
T_R(slot\texttt{=}DLMLabel, x) &= T_S(slot, x) : DLMLabel \\
T_S(slot.component, x) &= T_S(slot, x).component \\
T_S(\texttt{self}, x) &= x
\end{aligned}$$

where `type` is a type of a variable (irrelevant for the policy parsing) and $\mathcal{P}$ is a dictionary of policy specifications.

The translation is performed in a *compilation* phase for each definition statement in the code in the order of appearance. The two functions $T_{P_{def}}$ and $T_{S_{def}}$ return updated policy specification dictionaries. Lambda expressions are used so that the `self` tokens defined in the policy are instantiated with concrete slots at the place of use. The three T_{def} operations return the new updated global policies are used as an argument in parsing the definition statements that follow. All other translation functions are called from the first five operations and their results are used in updating the global policy.

For the semantic policies we define the $R_{\text{eff}}(\mathtt{P}, \sigma, x, p)$ and $W_{\text{eff}}(\mathtt{P}, \sigma, x, p)$ functions that return the effective readers and writers respectively for slot x, state σ and principal p, given policy $\mathtt{P}$:

$$
\begin{aligned}
R_{\text{eff}}(X : L, \sigma, x, p) &= \begin{cases} \mathsf{readers}(L, p) & \text{iff } x \in X \\ \mathtt{*} & \text{otherwise} \end{cases} \\
R_{\text{eff}}(\phi \Rightarrow \mathtt{P}, \sigma, x, p) &= \begin{cases} R_{\text{eff}}(\mathtt{P}, \sigma, x, p) & \text{iff } \sigma \models \phi \\ \mathtt{*} & \text{otherwise} \end{cases} \\
R_{\text{eff}}(\mathtt{P}_1; \mathtt{P}_2, \sigma, x, p) &= R_{\text{eff}}(\mathtt{P}_1, \sigma, x, p) \cap R_{\text{eff}}(\mathtt{P}_2, \sigma, x, p) \\
W_{\text{eff}}(X : L, \sigma, x, p) &= \begin{cases} \mathsf{writers}(L, p) & \text{iff } x \in X \\ \mathtt{_} & \text{otherwise} \end{cases} \\
W_{\text{eff}}(\phi \Rightarrow \mathtt{P}, \sigma, x, p) &= \begin{cases} W_{\text{eff}}(\mathtt{P}, \sigma, x, p) & \text{iff } \sigma \models \phi \\ \mathtt{_} & \text{otherwise} \end{cases} \\
W_{\text{eff}}(\mathtt{P}_1; \mathtt{P}_2, \sigma, x, p) &= W_{\text{eff}}(\mathtt{P}_1, \sigma, x, p) \cup W_{\text{eff}}(\mathtt{P}_2, \sigma, x, p)
\end{aligned}
$$

where $\sigma \models \phi$ holds whenever state σ satisfies the condition ϕ.

We can now define the following partial ordering of policies, which is to be read as "policy $\mathtt{P}_2$ is at least as restrictive as policy $\mathtt{P}_1$":

$$
\mathtt{P}_1 \sqsubseteq \mathtt{P}_2 \quad \text{iff } \forall_{\sigma,x,p} : \begin{pmatrix} R_{\text{eff}}(\mathtt{P}_1, \sigma, x, p) \supseteq R_{\text{eff}}(\mathtt{P}_2, \sigma, x, p) \wedge \\ W_{\text{eff}}(\mathtt{P}_1, \sigma, x, p) \subseteq W_{\text{eff}}(\mathtt{P}_2, \sigma, x, p) \end{pmatrix}
$$

The structure of these functions and the partial ordering rule is similar to what has been presented for the DLM labels. The difference is that these consider all slots of the program (the policy is global) and all possible states. In the type system, the global policy will be modified into two versions to reflect the information flow, and these two versions will be compared to determine whether the flow is valid.

4.4 Validation of programs

In this section I will first provide informal rules for validation of programs written in the language defined in this chapter. This description can be used by the user of this language as a reference to understand how to write secure programs. Then I will formalize the reasoning behind the verification into a type system.

4.4.1 Informal description

All variables and structures – altogether slots – in the program have a policy attached to it. If it is not specified then it is equal to the least restrictive label $\bot$. In this description, the policies of slots will be denoted by underlining (e.g. $\underline{xv}$ is the policy applying to variable xv).

Let us start from defining the context that is provided by the *if* conditionals and *while* loops. These statements create blocks through which not all execution paths are passing, and thus, if an assignment occurs inside those statements, then some information is passed to the assigned slot about the slots that are present in the conditions (the boolean expressions guarding both *if* and *while* statements). In order to register that phenomenon, we will maintain a set of slots that influence the current program statement (X), which will keep the knowledge on what kind of information may be transferred on assignments in those blocks. For both *if* and *while* statements the X will be all variables present in the condition and the X of the enclosing block.

Another information that the context provided by the statement blocks yields are the constraints on the actual values of the variables appearing in those blocks. These constraints also result from the conditions and are useful for determining which policy should apply. The constraints holding at any given statement will be denoted by ϕ_{pc}, which will encompass conditions from all enclosing blocks and results of assignments inside and outside them.

Now, given the knowledge about the context we can define the rule for the assignment. An assignment of form $xv = e$ can only be valid if $\underline{xv}$ is at least as restrictive as $\underline{e}$ (i.e. $\underline{e} \sqsubseteq \underline{xv}$), where $\underline{e}$ is an aggregation of the policies of variables appearing in the e expression. Furthermore, $\underline{xv}$ must also be at least as restrictive as $\underline{pc}$ (i.e. $\underline{pc} \sqsubseteq \underline{xv}$) – the label of the program counter, which results from joining policies of the slots in X indicating the implicit flow of information.

It is also possible that xv is actually a field of some structure, or such fields are present in e. Then, also the policies of the ancestor structures need to be taken into consideration. Let us assume that $\underline{xv}$ encompasses the policies attached to the variable xv and the policies governing the ancestor structures. The same applies to all variables in e so that $\underline{e}$ is joining their policies. This approach differs from DLM, where the logic of combining nested labels is more complex.

Finally, the actual policies will be determined by the conditions attached to them. The constraint environment ϕ_{pc} holding before the assignment statement will determine which policies should be applied to the expression e, while ψ_{pc} holding after the assignment will do the same for xv. We denote this kind of selection of policies by writing the constraint environments in subscripts. The full expression for validating the assignment will be then:

$$\underline{e}_{\phi_{pc}} \sqcup \underline{pc}_{\phi_{pc}} \sqsubseteq \underline{xv}_{\psi_{pc}}$$

Apart from simple assignment, we also have structure initialization and structure assignment. These practically evaluate to multiple assignments executed atomically, which means that they share ϕ_{pc} and ψ_{pc} environments.

The last validated aspect of code are loop invariants, if specified by the programmer. The provided invariant must be true before each execution of the loop, and after it.

4.4.2 Type system

The formalization will closely follow the one given in [29]. In order to capture the contextual information described in the previous section, we will resort to symbolic execution with Hoare logic. The Hoare logic triples are incorporated into the type system judgements and axioms as follows:

$$X \vdash \{\phi\}S\{\phi'\}$$

where X is a set of slots on which execution of the statement S depends (the slots used in the enclosing *if* and *while* statement conditions), while ϕ and ϕ' are the pre- and postconditions of the statement. The pre- and postconditions are specified as logical formulae over the variables present in the program. The type system is depicted in fig. 4.1.

Empty statements and declarations

The first rule regulates an empty statement `;` which is similar in semantics to a classical *skip* – the statement changes nothing.

The second and third rule concern declarations, which have no direct influence on the information flow and do nothing because the policies are constructed in the compilation phase. As defined in section 4.1.1, *decl* can be declaration of a variable or a structure instance, while *stc* is a structure type declaration.

Assignments

The responsibility of the simple assignment rule is triple. First, it needs to ensure that if the assignment changes the policy that applies then the new applying policy should be compared against the old. This is achieved with substitution ($\mathtt{P}[e/xv]\langle\mathcal{A}(xv)/xv\rangle$) and augmentation (here $(\phi \Rightarrow \mathtt{P})\langle X \cup \mathcal{A}(e)/xv\rangle$). The substitution virtually selects the policies that apply to xv after the assignment. The augmentation, on the other hand, selects and assembles policies of the slots that influence xv by the assignment. Here, the ϕ used as the condition for `P` in the augmentation serves the role of a selector of the policies that apply before the assignment. Augmentation attaches these policies to xv so that the two policies – substituted and augmented – are comparable on xv. And this leads us to the second purpose of the assignment rule, which is to ensure that the assignment results in a restriction of security. This is accomplished with comparison of the two afore-mentioned policies. Finally, the rule needs to take care of the change in the context that results from the assignment, which is governed in the postcondition of the Hoare logic triple. There, a fresh slot xv' is instantiated and substitutes the old-valued xv in the right-hand side expression of the assignment and the context ϕ.

Both substitution and augmentation in the assignment rule contain the accessor function $\mathcal{A}$ in their parameters. It retrieves all accessors used in the given expression (or slot), which for simple variables is an identity. For slots being components of structures it also returns all the ancestor structures:

$$\begin{aligned}\mathcal{A}(e_1\ op\ e_2) &= \mathcal{A}(e_1) \cup \mathcal{A}(e_2)\\ \mathcal{A}(x) &= \mathcal{A}_{\mathrm{x}}(x, \epsilon)\\ \mathcal{A}_{\mathrm{x}}(v, pref) &= \{pref + v\}\end{aligned}$$

$$X \vdash \{\phi\}; \{\phi\}$$

$$X \vdash \{\phi\} decl \{\phi\}$$

$$X \vdash \{\phi\} stc \{\phi\}$$

$$X \vdash \{\phi\} xv = e; \{\phi'\} \quad \text{if } \begin{cases} (\phi \Rightarrow \mathtt{P})\langle X \cup \mathcal{A}(e)/xv\rangle \sqsubseteq \mathtt{P}[e/xv]\langle \mathcal{A}(xv)/xv\rangle, \\ \phi' \triangleq \exists_{xv'} : xv = e[xv'/xv] \wedge \phi[xv'/xv] \end{cases}$$

$$X \vdash \{\phi\} stci = init \{\phi'\}$$
$$\text{if } \begin{cases} c_1, c_2 \ldots c_n \in \mathcal{C}(si) \\ cs_1, cs_2 \ldots cs_m \in \mathcal{CS}(si) \\ \phi_1 \triangleq c_1 = e_1 \wedge \phi, \\ \vdots \\ \phi' \triangleq c_n = e_n \wedge \phi_{n-1} \\ \mathtt{P}_{rst} \triangleq (\phi \Rightarrow \mathtt{P})\langle \emptyset/cs_1\rangle \ldots \langle \emptyset/cs_m\rangle, \\ \mathtt{P}_{left} \triangleq \mathtt{P}_{rst}\langle X/si\rangle\langle \mathcal{A}(e_1)/c_1\rangle \ldots \langle \mathcal{A}(e_n)/c_n\rangle, \\ \mathtt{P}_{right} \triangleq \mathtt{P}[e_1/c_1]\langle \mathcal{A}(c_1)/c_1\rangle \ldots [e_n/c_n]\langle \mathcal{A}(c_n)/c_n\rangle, \\ \mathtt{P}_{left} \sqsubseteq \mathtt{P}_{right} \end{cases}$$

$$X \vdash \{\phi\} xs_t = xs_s \{\phi'\}$$
$$\text{if } \begin{cases} c_1, c_2 \ldots c_n \in \mathcal{C}(xs_t) \\ \phi_1 \triangleq \exists_{c_1'} : c_1 = c_1[xs_s/xs_t] \wedge \phi[c_1'/c_1], \\ \vdots \\ \phi' \triangleq \exists_{c_n'} : c_n = c_n[xs_s/xs_t] \wedge \phi_{n-1}[c_n'/c_n] \\ \mathtt{P}_{left} \triangleq (\phi \Rightarrow \mathtt{P})\langle X/xs_t\rangle\langle \mathcal{A}(c_1[xs_s/xs_t])/c_1\rangle \ldots \langle \mathcal{A}(c_n[xs_s/xs_t])/c_n\rangle, \\ \mathtt{P}_{right} \triangleq \mathtt{P}[c_1[xs_s/xs_t]/c_1]\langle \mathcal{A}(c_1)/c_1\rangle \ldots [c_n[xs_s/xs_t]/c_n]\langle \mathcal{A}(c_n)/c_n\rangle, \\ \mathtt{P}_{left} \sqsubseteq \mathtt{P}_{right} \end{cases}$$

$$\frac{X \vdash \{\phi\} S_1 \{\psi\} \quad X \vdash \{\psi\} S_1 \{\phi'\}}{X \vdash \{\phi\} S_1 \ S_2 \{\phi'\}}$$

$$\frac{X \cup \mathcal{A}(b) \vdash \{\phi \wedge b\} S_1 \{\phi'\} \quad X \cup \mathcal{A}(b) \vdash \{\phi \wedge \neg b\} S_2 \{\phi'\}}{X \vdash \{\phi\} \text{if}(b)\ \{S_1\} \text{ else } \{S_2\} \{\phi'\}}$$

$$\frac{X \cup \mathcal{A}(b) \vdash \{\iota \wedge b\} S \{\iota\}}{X \vdash \{\phi\} \text{while}(b)[\psi]\ \{S\} \{\iota \wedge \neg b\}} \quad \text{if } \begin{cases} \phi \Rightarrow \iota, \\ \iota \triangleq \phi \backslash dx(S) \wedge \psi \end{cases}$$

$$\frac{X \vdash \{\psi\} S \{\psi'\}}{X \vdash \{\phi\} S \{\phi'\}} \quad \text{if } (\phi \Rightarrow \psi) \wedge (\psi' \Rightarrow \phi')$$

Figure 4.1: Type system for the non-referential language

$$\mathcal{A}_{\text{x}}(si, pref) = \{pref + si\}$$
$$\mathcal{A}_{\text{x}}(si.x, pref) = \mathcal{A}_{\text{x}}(si, pref) \cup \mathcal{A}_{\text{x}}(x, pref + si.)$$

where + is a concatenation operator.

For example, $\mathcal{A}(s.c1 - s2.c2) = \{s, s.c1, s2, s2.c2\}$.

Thanks to this interpretation of $\mathcal{A}$, and augmentation applied also after substitution, the policies influencing ancestor structures are taken into account whenever their components are used in xv and e (or b in case of control statements). A different viable solution to include them could be flattening of policies attached to structures, i.e. augmenting components with policies of their ancestors, already during the compilation step. Although it would result in the same behaviour (as the global policy is static), such redundant information with every component would negatively impact readability of the policies.

Let us formally define substitution and augmentation. The substitution function changes the conditions of the policies to reflect the new state of the system:

$$\{X : L\}[e/x] = \{X : L\}$$
$$(\phi \Rightarrow \texttt{P})[e/x] = \phi[e/x] \Rightarrow \texttt{P}[e/x]$$
$$(\texttt{P}_1; \texttt{P}_2)[e/x] = \texttt{P}_1[e/x]; \texttt{P}_2[e/x]$$

where $\phi[e/x]$ simply replaces all occurrences of x in ϕ with expression e.

The augmentation function, given a set of slots X and a slot x, reflects that x is augmented with labels of X:

$$\{Y : L\}\langle X/x\rangle = \begin{cases} \{Y \cup \{x\} : L\} & \text{if } Y \cap X \neq \emptyset \\ \{Y \setminus \{x\} : L\} & \text{otherwise} \end{cases}$$
$$(\phi \Rightarrow \texttt{P})\langle X/x\rangle = \phi \Rightarrow \texttt{P}\langle X/x\rangle$$
$$(\texttt{P}_1; \texttt{P}_2)\langle X/x\rangle = \texttt{P}_1\langle X/x\rangle; \texttt{P}_2\langle X/x\rangle$$

The first rule removes x from Y if it does not have any common elements with the influencers set X. This is necessary, as the augmented policy of x should not consider policies of slots that are in fact not influencing its new value, even though that might be x itself. We will call this process *policy erasure*. The most straightforward example displaying the need for this process is a simple assignment, in which the right-hand side expression does not contain the assigned slot, presented in listing 4.4.

```
1 int {(self == 1 => {A->B});(self == 2 => {A->_})} x = 2;
2 x = 1;
```

Listing 4.4: Example of a non-self-influencing assignment

In the second assignment, if x is not removed from the influencers, then the augmented policy remains unchanged. The substituted policy simplifies then to $\{A \to B\}$ (`self` is replaced with 1 and so only policy resulting from $1 == 1$ applies). Then, for the state just before the second assignment where $x == 2$ is true, we have a violation of the assignment rule stating that the substituted

policy must be at least as restrictive as the augmented policy. By induction the same problem happens already in the first assignment, however, the second was chosen to display it for transparency reasons.

One other subject to discuss concerning the assignment rule is usage of the ϕ as the selector for augmentation. An alternative would be using ϕ' instead, which would cause selection of the policy that applies to x after assignment, and thus allow unrestricted self-assignments (eg. `x=x+1`). It would be most convenient for iterations with loops, in case the policy of the incremented slot was not trivial. Unfortunately that alternative introduces a leak which is depicted in listing 4.5.

```
1 int {(self > 0 => {A->_})} x;
2 int {{A->_}} y;
3 ...
4 if(y > 0) {
5         x = y;
6 }
7 x = -x;
```

Listing 4.5: Example of leak if self-assignments are unrestricted

The comparison operators are not allowed in our specification of policies, however, by using it here we do not lose generality and in the same time demonstrate the scale of the problem. After the last self-assignment we get full information about the content of `y` in an unrestricted variable `x` (it is unrestricted for negative values), provided that before the *if* conditional the value of `y` is higher than 0. Otherwise we obtain information that `y` is not positive. One can use this trick twice with some additional checks to gain full knowledge of `y`. That is why also the self-assignments must be restricted and it is ϕ that is used as the selector of the policy for augmentation.

Atomic assignments – structure initializer lists

In the simple language presented in this chapter, interdependency of components of structures is allowed, i.e. slots that influence other slots within the same structure (not only themselves as is the case of variables). This unfortunately introduces a problem whenever a slot that influences the labelling of some other slot is changed. The problem is that in the type system we are only looking at a single assignment, which may only perform a partial structure update. Such assignment will be invalid if the new label of the influenced slot is less restrictive. An example is shown in listing 4.6.

In the example we have an instance `si` of structure `s` where `si.det` determines the policy of `si.c`. The value of `si.det` before it is assigned is not known, which means that for states where $si.det == 1$ is true the reader set of `si.c` for *Alice* will be only *Alice* and *Bob*. However, after `si.det = 2` it will be `*`, which means that this assignment is illegal. But this is too restrictive, since it will forbid even initialisation of structures with interdependent components, which has been depicted in this example.

That is why structure initializers (*stinit*) have to be introduced. An initializer statement makes an assignment to each component and subcomponent of

```
1 struct s {
2         int det;
3         int c;
4 }{
5         (self.det == 1 => self.c={Alice->Bob})
6 } si;
7 si.det = 2;
8 si.c = 3;
```

Listing 4.6: Example of partial structure update problem

some structure *si* (defined in *stci*). In order to retrieve all those components we use the following function:

$$\begin{aligned}
\mathcal{C}(xv) &= \{xv\} \\
\mathcal{C}(xs) &= \mathcal{C}(xs.x_1) \cup \mathcal{C}(xs.x_2) \cup \cdots \cup \mathcal{C}(xs.x_n) \\
&\qquad \text{if} \quad xs \text{ defines } x_1, x_2, \ldots x_n
\end{aligned}$$

For example, for the `si` structure instance from listing 4.6 the application of the $\mathcal{C}$ function would return $\{si.det, si.c\}$.

The expressions $(e_1, \ldots, e_n)$ given by the programmer in the initializer list (*init*) are matched with the components by the order of both those expressions and the components' definitions in the code. This matching is implicit in the type system.

For initializers the policies must hold before them and after all assignments. In order to obtain the left and right policies for the whole block, the effects of all assignments are accumulated by performing augmentation and substitution sequentially.

Unlike in the case of normal assignments, there is no preceding information about the receiving slot. This means that it is not necessary to instantiate fresh slots and substitute the precondition or the mid-conditions during the accumulation process.

Furthermore, in the process of establishing the left-hand side policy, the policies governing the initialized structure instance need to be reset. It is safe to do so, because at the point before initialization there is yet no data in that structure instance to be protected. It has to be done because not all slots are assigned to – the structures are not, their components are – which means that those policies will not be automatically reset by *policy erasure*.

```
1 struct s2 {
2         float c1;
3 };
4 struct s {
5         int det;
6         struct s2 c;
7 } {(self.det == 2 => self.c={A->C})} si = {1, {0.2}};
```

Listing 4.7: Example of structure initialization policy erasure problem

Consider a simple example given in listing 4.7. If the policy is not reset first, then the policies for comparison that we obtain are (braces around the sets of slots omitted):

$$\begin{aligned} \mathtt{P}_{left} &= (si.det = 2 \Rightarrow si.c : \{A \to C\}) \\ \mathtt{P}_{right} &= (1 = 2 \Rightarrow si.c, si.c.c1 : \{A \to C\}) \end{aligned}$$

where the left-hand side policy is the actual policy of the example, unchanged, because it is `si.c.c1` that is considered to be assigned, not `si.c`. Clearly, the left-hand side policy is more restrictive than the right-hand side one.

In order to get the structure instance slot with all its subcomponents, including structure instances, a version of the $\mathcal{C}$ function us used:

$$\begin{aligned} \mathcal{CS}(xv) &= \{xv\} \\ \mathcal{CS}(xs) &= \{xs\} \cup \mathcal{CS}(xs.x_1) \cup \mathcal{CS}(xs.x_2) \cup \dots \cup \mathcal{CS}(xs.x_n) \\ &\quad \text{if} \quad xs \text{ defines } x_1, x_2, \dots x_n \end{aligned}$$

For example, for the `si` structure instance from listing 4.7 the application of the $\mathcal{CS}$ function would return $\{si.det, si.c, si.c.c1\}$.

The solution involving structure initialization is good, however, quite inflexible. I did consider some other solution providing more flexibility, but it is more invasive changing the language syntax, and would cause problems for the extension provided in the next chapter. Nevertheless, I provide it here for whoever might be interested.

The alternative introduces an atomic block, where a number of assignments can be aggregated in order to *update* a structure without breaking the interdependent policies. A type rule for such block of assignments is given in fig. 4.2.

$$X \vdash \{\phi\} \begin{array}{l} \mathtt{atomic\{} \\ \quad xv_1 = e_1; \\ \quad \vdots \\ \quad xv_n = e_n; \\ \mathtt{\}} \end{array} \{\phi'\} \quad \text{if} \begin{cases} \phi_1 \triangleq \exists_{xv'_1} : xv_1 = e_1[xv'_1/xv_1] \wedge \phi[xv'_1/xv_1], \\ \vdots \\ \phi' \triangleq \exists_{xv'_n} : xv_n = e_n[xv'_n/xv_n] \wedge \phi_{n-1}[xv'_n/xv_n] \\ \mathtt{P}_{left} \triangleq (\phi \Rightarrow \mathtt{P})\langle X \cup \mathcal{A}(e_1)/xv_1\rangle \dots \langle X \cup \mathcal{A}(e_n)/xv_n\rangle, \\ \mathtt{P}_{right} \triangleq \mathtt{P}[e_1/xv_1]\langle \mathcal{A}(xv_1)/xv_1\rangle \dots [e_n/xv_n]\langle \mathcal{A}(xv_n)/xv_n\rangle, \\ \mathtt{P}_{left} \sqsubseteq \mathtt{P}_{right}, \left(\bigcup_{1\le i\le n} \{xv_i\}\right) \cap \left(\bigcup_{1\le i\le n} \mathcal{A}(e_i)\right) = \emptyset, \\ \left|\bigcup_{1\le i\le n} \{xv_i\}\right| = n, \end{cases}$$

Figure 4.2: Axiom for block of atomic assignments

Compared to the structure initializer block, there are some additional steps that are required to make this work without breaching security. Let us examine example shown in listing 4.8.

```
1 struct s {
2         int det;
3         int c;
4 }{
5         (self.det == 1 => self.c={A->B});
6         (self.det == 2 => self.c={A->C})
7 } si;
8 int y = 1;
9 int {{A->C}} z;
10 atomic {
11         si.det = y;
12         y = 2;
13         si.c=z;
14 }
```

Listing 4.8: Example of sequential assignment accumulation problem

The example results in a global policy:

$$\begin{aligned} \mathtt{P} = {} & (si.det = 1 \Rightarrow si.c : \{A \to B\}); \\ & (si.det = 2 \Rightarrow si.c : \{A \to C\}); \\ & (z : \{A \to C\}) \end{aligned}$$

And let us apply substitutions (we omit augmentations that normally occur after substitution since no labelling is attached to the structure itself):

$$\begin{aligned} & \mathtt{P}[y/si.det][2/y][z/si.c] \\ = {} & (y = 1 \Rightarrow si.c : \{A \to B\}); \\ & (y = 2 \Rightarrow si.c : \{A \to C\}); \\ & (z : \{A \to C\})[2/y][z/si.c] \\ = {} & (2 = 1 \Rightarrow si.c : \{A \to B\}); \\ & (2 = 2 \Rightarrow si.c : \{A \to C\}); \\ & (z : \{A \to C\})[z/si.c] \\ = {} & (2 = 1 \Rightarrow si.c : \{A \to B\}); \\ & (2 = 2 \Rightarrow si.c : \{A \to C\}); \\ & (z : \{A \to C\}) \end{aligned}$$

The problem here is that the set of assignments is not really atomic, as it is executed according to the C standard, and although `si.det` is assigned with value 1, the type system would interpret it as if it was 2, because the variable `y` is reassigned within the atomic block. The consequence would be that value of `z` could be transferred to `si.c` violating the policy, and it would not be detected. A solution is to require that the two sets, slots assigned in the atomic block, and slots appearing on the right-hand side of the assignments, be disjoint. This is exactly what is specified by the $(\bigcup_{1 \leq i \leq n} \{xv_i\}) \cap (\bigcup_{1 \leq i \leq n} \mathcal{A}(e_i)) = \emptyset$ condition. Unfortunately it does not solve all the problems.

```
1 struct s {
2         int det;
3         int c;
4 }{
5         (self.det == 1 => self.c={A->B});
6         (self.det == 2 => self.c={A->C})
7 } si;
8 int {{A->C}} z;
9 atomic {
10        si.det = 2;
11        si.det = 1;
12        si.c=z;
13 }
```

Listing 4.9: Example of multiple assignment problem

The example in listing 4.9 uses the same data structures with the same policies and the above-mentioned sets are disjoint, however, the problem remains, as the policies are substituted as follows:

$$\begin{aligned}
&\mathtt{P}[2/si.det][1/si.det][z/si.c] \\
=\ &(2 = 1 \Rightarrow si.c : \{A \rightarrow B\}); \\
&(2 = 2 \Rightarrow si.c : \{A \rightarrow C\}); \\
&(z : \{A \rightarrow C\})[1/si.det][z/si.c] \\
=\ &(2 = 1 \Rightarrow si.c : \{A \rightarrow B\}); \\
&(2 = 2 \Rightarrow si.c : \{A \rightarrow C\}); \\
&(z : \{A \rightarrow C\})[z/si.c] \\
=\ &(2 = 1 \Rightarrow si.c : \{A \rightarrow B\}); \\
&(2 = 2 \Rightarrow si.c : \{A \rightarrow C\}); \\
&(z : \{A \rightarrow C\})
\end{aligned}$$

As we can see, the second update of the value of `si.det` is not accounted for. In order to fix this, the atomic assignment axiom requires that each slot is updated only once within the atomic block. This requirement is specified by building a set of all assigned slots and checking that its cardinality is equal to the number of assignments.

Structure assignment

Another type of bulk assignment happens when a structure is assigned to another structure. That case is handled by the type system in the rule that follows the one for structure initialization. The shape of this rule has only a few differences with respect to the structure initialization rule.

Here, the components of one structure instance need to be matched with the counterparts in the other. In order to do that, a slot name substitution ($c[xs_s/xs_t]$) is used, which replaces the name of the *target* structure with the name of the *source* structure. Furthermore, similarly as for singular assignments, the existential quantifier is used in order to relax any previous constraints bounding the components of the target in the precondition. Unlike in the singular assignment axiom, the right-hand side of the expression does not need to

be substituted with the fresh slot since no target component is present there (after the source-target substitution).

Composition and control statements

The judgement for composition ($S1\ S2$) follows classical rules in Hoare logic, and so does the judgement for the *if* conditional, with the difference that in X the type system keeps track of the slots from b that influence execution of the inner statements.

The judgement of the *while* loop differs from the standard Hoare logic interpretation. The reason is to avoid fixed-point analysis that is otherwise necessary in order to establish the strongest loop invariant. Instead, some weak invariant is inferred using the weakening operation (symbolized with a backslash), and the programmer is allowed to strengthen it with ψ that he may provide. The $dx(S)$ function simply extracts all slots that are redefined in S.

The weakening operation is defined as follows:

$$\phi \backslash W = \exists_{x_1',\ldots,x_n'} : \phi[x_1'/x_1]\ldots[x_n'/x_n]$$

where $x_1,\ldots,x_n \in (\mathcal{C}(w_1) \cup \cdots \cup \mathcal{C}(w_n))$, where $w_1,\ldots,w_n \in W$.

As an example, if we have:

$$\phi = ((s.c1 == 1) \wedge (s.c2 == 2) \wedge (s2.c1 == 0) \wedge (s2.c2 == 1))$$

and $W = \{s.c1, s2\}$, then $\phi \backslash W$ would return:

$$\exists_{s.c1',s2.c1',s2.c2'} : ((s.c1' == 1) \wedge (s.c2 == 2) \wedge (s2.c1' == 0) \wedge (s2.c2' == 1))$$

The weakening operation guarantees that $\phi \Rightarrow \phi \backslash dx(S)$.

The consequence rule

The last judgement corresponds to the consequence rule that allows to strengthen preconditions and weaken postconditions. The type system can be refined to remove that rule and directly employ its effects wherever it is necessary. Such system is provided in fig. 4.3.

In the process of refinement, only the judgements for *if* conditionals and *while* loops have changed – other rules were already syntax driven. For the *if* statement it was enough to make a disjunction of the postconditions of its branches in order to get a correct postcondition of the statement. As for the *while* statement, the postcondition of the do-clause has been relaxed with respect to its precondition.

Thanks to removal of the consequence rule, and to the technique in which postconditions are constructed from preconditions the obtained type system is fully syntax driven, where only assignments and the relationships between the pre- and postconditions of loops need to be verified.

$$X \vdash \{\phi\};\{\phi\}$$

$$X \vdash \{\phi\}decl\{\phi\}$$

$$X \vdash \{\phi\}stc\{\phi\}$$

$$X \vdash \{\phi\}xv = e;\{\phi'\} \quad \text{if } \begin{cases} (\phi \Rightarrow \mathtt{P})\langle X \cup \mathcal{A}(e)/xv\rangle \sqsubseteq \mathtt{P}[e/xv]\langle \mathcal{A}(xv)/xv\rangle, \\ \phi' \triangleq \exists_{xv'} : xv = e[xv'/xv] \wedge \phi[xv'/xv] \end{cases}$$

$$X \vdash \{\phi\}stci = init\{\phi'\}$$

$$\text{if } \begin{cases} c_1, c_2 \ldots c_n \in \mathcal{C}(si) \\ cs_1, cs_2 \ldots cs_m \in \mathcal{CS}(si) \\ \phi_1 \triangleq c_1 = e_1 \wedge \phi, \\ \vdots \\ \phi' \triangleq c_n = e_n \wedge \phi_{n-1} \\ \mathtt{P}_{rst} \triangleq (\phi \Rightarrow \mathtt{P})\langle \emptyset/cs_1\rangle \ldots \langle \emptyset/cs_m\rangle, \\ \mathtt{P}_{left} \triangleq \mathtt{P}_{rst}\langle X/si\rangle\langle \mathcal{A}(e_1)/c_1\rangle \ldots \langle \mathcal{A}(e_n)/c_n\rangle, \\ \mathtt{P}_{right} \triangleq \mathtt{P}[e_1/c_1]\langle \mathcal{A}(c_1)/c_1\rangle \ldots [e_n/c_n]\langle \mathcal{A}(c_n)/c_n\rangle, \\ \mathtt{P}_{left} \sqsubseteq \mathtt{P}_{right} \end{cases}$$

$$X \vdash \{\phi\}xs_t = xs_s\{\phi'\}$$

$$\text{if } \begin{cases} c_1, c_2 \ldots c_n \in \mathcal{C}(xs_t) \\ \phi_1 \triangleq \exists_{c_1'} : c_1 = c_1[xs_s/xs_t] \wedge \phi[c_1'/c_1], \\ \vdots \\ \phi' \triangleq \exists_{c_n'} : c_n = c_n[xs_s/xs_t] \wedge \phi_{n-1}[c_n'/c_n] \\ \mathtt{P}_{left} \triangleq (\phi \Rightarrow \mathtt{P})\langle X/xs_t\rangle\langle \mathcal{A}(c_1[xs_s/xs_t])/c_1\rangle \ldots \langle \mathcal{A}(c_n[xs_s/xs_t])/c_n\rangle, \\ \mathtt{P}_{right} \triangleq \mathtt{P}[c_1[xs_s/xs_t]/c_1]\langle \mathcal{A}(c_1)/c_1) \ldots [c_n[xs_s/xs_t]/c_n]\langle \mathcal{A}(c_n)/c_n\rangle, \\ \mathtt{P}_{left} \sqsubseteq \mathtt{P}_{right} \end{cases}$$

$$\frac{X \vdash \{\phi\}S_1\{\psi\} \quad X \vdash \{\psi\}S_2\{\phi'\}}{X \vdash \{\phi\}S_1 \ S_2\{\phi'\}}$$

$$\frac{X \cup \mathcal{A}(b) \vdash \{\phi \wedge b\}S_1\{\psi\} \quad X \cup \mathcal{A}(b) \vdash \{\phi \wedge \neg b\}S_2\{\psi'\}}{X \vdash \{\phi\}\text{if}(b)\ \{S_1\}\ \text{else}\ \{S_2\}\{\psi \vee \psi'\}}$$

$$\frac{X \cup \mathcal{A}(b) \vdash \{\iota \wedge b\}S\{\iota'\}}{X \vdash \{\phi\}\text{while}(b)[\psi]\ \{S\}\{\iota \wedge \neg b\}} \quad \text{if } \begin{cases} (\phi \Rightarrow \iota) \wedge (\iota' \Rightarrow \iota), \\ \iota \triangleq \phi \backslash dx(S) \wedge \psi \end{cases}$$

Figure 4.3: Syntax driven type system for the non-referential language

Chapter 5

Referential C analysis

In this chapter I will extend the language and the type system provided previously with arrays and pointers. Since I am building on previous description, I will not repeat the parts that remain unchanged in this extension.

5.1 Language specification

The language considered in this chapter will contain all features specified in section 4.1 as well as:

- Static and dynamic arrays
- Pointers to simple data types and non-cyclic structures
- Assignments of slot addresses to pointers
- *malloc* and *sizeof* operations

Using these components the solution will be able to support the use case scenario (with abstraction of functions), as well as a more realistic version of the previously defined example.

5.1.1 Syntax

The following provides the syntax of this extension (unchanged rules omitted):

$$
\begin{aligned}
lts &::= lt \mid stc \mid \texttt{struct}\ s \\
ct &::= t \mid \texttt{struct}\ s \\
arrd &::= lts\ arr\texttt{\{}policy\texttt{\}}[n]; \\
ptrd &::= lts\ *\texttt{\{}policy\texttt{\}}ptr; \\
decl &::= lt_a\ x_a; \mid lt_b\ x_b; \mid stci \mid arrd \mid ptrd \\
a &::= n \mid x_a \mid a_1\ op_a\ a_2 \mid \texttt{sizeof(}ct\texttt{)} \\
x_a &::= si.x_a \mid v_a \mid *ptr_a \mid (*ptr).v_a \mid sub_a \\
x_b &::= si.x_b \mid v_b \mid *ptr_b \mid (*ptr).v_b \mid sub_b \\
xp &::= si.xp \mid arr[a].xp \mid (*ptr).xp \mid ptr \\
sub_a &::= arr_a[a] \mid arr[a].x_a \mid ptr_a[a] \mid ptr[a].x_a \\
sub_b &::= arr_b[a] \mid arr[a].x_b \mid ptr_b[a] \mid ptr[a].x_b \\
assign &::= x_b\texttt{=}b; \mid x_a\texttt{=}a; \mid xp\texttt{=\&}x; \mid xp\texttt{=malloc(}a\texttt{);}
\end{aligned}
$$

There are two new symbols which meaning needs to be explained – symbol *arr* stands for an array name, while *ptr* is an identifier of a pointer. Apart from that, there are two standard library functions of C: `malloc` and `sizeof`, as well as dereference `*` and address `&` operators.

Both arrays and pointers have a type with a policy as their base type, and also allow attaching a policy to themselves. This is necessary, as these are types which basic values (like address) may also bear some information. Furthermore, this allows specifying overlay or global policies on top of those that are defined by the base types. In this aspect, the solution presented here differs from the standard DLM, in which for arrays these policies are called *labels of successful array access* and whenever an assignment to an array appears these labels are on the influencing side against the label of the array elements (base type). Here, such additional verification is not introduced and those policies are treated differently (as convenience), because a C program may either successfully evaluate a pointer or array access, or end – a covert channel which is not reasoned about in this work.

Dereferences of pointers and array accesses are also different kinds of slots, which is reflected by including them under the x_a and x_b rules . Moreover, these can be arbitrarily nested, which results from recursive definition of structures. However, note that although pointers or arrays of structures, which also contain pointers or arrays are possible, immediate nesting (e.g. arrays of pointers) is not supported.

Usage of pointers without the dereference operator, in order to access the address referenced by the pointer is distilled in the *xp* rule, and it is used in *pointer assignments* and allocation of memory with `malloc`. The use of memory addresses, resulting from `malloc` calls, the address operator `&` and pointer addresses is restricted only to assignments to the pointers themselves, in order to simplify the model.

The definition of slots containing array accesses is aggregated separately for clarity. The array subscript operator `[]` can be used both on arrays and pointers

(as in C). This, along with the `malloc` function, allows dynamic creation of arrays with their size defined in runtime.

The previously defined slot domains are insufficient as this language also has arrays and pointers. Arrays constitute the **Arr** domain, while pointers form the **Ptr** domain. Because of the fact that arrays and pointers can reference both simple types and structures, their domains are overlapping with both **Var** and **Str**, and are altogether enclosed in **Slot**.

5.1.2 Example

Listing 5.1 presents an example program written in the language defined in this chapter.

```
1  struct s {
2          int {{Alice->Bob,Chuck}} determinant;
3          int *data;
4  }{
5          (self.determinant == 1 => self.data={Alice->Bob});
6          (self.determinant == 2 => self.data={Alice->Chuck})
7  };
8  struct s input;
9
10 int out_chan{
11         (self.index == 0 => self={Alice->Bob});
12         (self.index == 1 => self={Alice->Chuck})
13 } [2];
14 int counter = 0;
15 while(counter < 2)[counter >= 0] {
16         if(input.determinant == counter + 1) {
17                 out_chan[counter] = input.(*data);
18         }
19         counter = counter + 1;
20 }
```

Listing 5.1: Example usage of policies in the extended language

This program is an augmentation of the one provided in Listing 4.1. It takes advantage of the new constructs, arrays in particular. Here we do not initialize the input – it is an example of how a program can be modelled. The actual values present in it can be abstracted away and it should be valid for any `input`. I will elaborate on this example later on in section 6.5.1.

5.2 Policies

In order to facilitate the addition of pointers and arrays in the language, the syntax and semantics of the policies need to be extended. The definition of labels, however, remains unchanged.

5.2.1 Policy syntax

The syntax change is very small, allowing to reference the implicit indexer of a pointer or an array:

$$slot ::= \texttt{self} \mid slot.component \mid slot.\texttt{index}$$

The sole purpose of this change is to allow the programmer to make the policy dependent on the index at which the data reside. An example of an index-dependent policy specification is shown in listing 5.2.

```
1 int x{(self.index == 0 => {Alice->*});
2       (self.index == 1 => {Alice->_})} [2];
3 int{{*->*}} *{(self.index == 3 => {*->Alice});
4               (self.index == 4 => {*->Bob})} y;
```

Listing 5.2: Example policy specification

Here, we have a policy of array `x` that depends on the index that is used to reference a value in this array. In this case, all possible values are governed by the policy (boundary checks are not in the scope of this work).

As for pointer `y`, the values that it stores have their own policy ($\{\{* \to *\}\}$), while it also depends on the index that is used to access the value in combination with the pointer (assuming that `y` may point to a dynamic array).

5.2.2 Policy semantics

The policy semantics do not change by the extension, however there are some additions to the set of translation functions:

$$\begin{aligned}
T_{def}(\texttt{type}\{policy_t\}\ arr\ \{policy_{arr}\}[n], \texttt{P}, \mathcal{P}) &= \texttt{P}; T_P(policy_t, \mathcal{P}, arr); T_P(policy_{arr}, \mathcal{P}, arr)\\
T_{def}(\texttt{type}\{policy_t\}\ *\{policy_{ptr}\}ptr, \texttt{P}, \mathcal{P}) &= \texttt{P}; T_P(policy_t, \mathcal{P}, ptr); T_P(policy_{ptr}, \mathcal{P}, ptr)\\
T_{def}(\texttt{struct}\ s\ arr\ \{policy_{arr}\}[n], \texttt{P}, \mathcal{P}) &= \texttt{P}; T_P(policy_{arr}, \mathcal{P}, arr); \mathcal{P}[s](arr)\\
T_{def}(\texttt{struct}\ s\{policy_{ptr}\}\ *ptr, \texttt{P}, \mathcal{P}) &= \texttt{P}; T_P(policy_{ptr}, \mathcal{P}, ptr); \mathcal{P}[s](ptr)\\
T_{decl}(\texttt{type}\{policy_t\}\ arr\ \{policy_{arr}\}[n], \mathcal{P}, x) &= T_P(policy_t, \mathcal{P}, x.arr); T_P(policy_{arr}, \mathcal{P}, x.arr)\\
T_{decl}(\texttt{type}\{policy_t\}\ *\{policy_{ptr}\}ptr, \mathcal{P}, x) &= T_P(policy_t, \mathcal{P}, x.ptr); T_P(policy_{ptr}, \mathcal{P}, x.ptr)\\
T_{decl}(\texttt{struct}\ s\ arr\ \{policy_{arr}\}[n], \mathcal{P}, x) &= T_P(policy_{arr}, \mathcal{P}, x.arr); \mathcal{P}[s](x.arr)\\
T_{decl}(\texttt{struct}\ s\ *\{policy_{ptr}\}ptr, \mathcal{P}, x) &= T_P(policy_{ptr}, \mathcal{P}, x.ptr); \mathcal{P}[s](x.ptr)\\
T_S(slot.\texttt{index}, x) &= T_S(slot, x).\texttt{index}
\end{aligned}$$

5.3 Validation of programs

In this section I will first provide informal rules for validation of programs written in the language defined in this chapter, by extending and modifying the description provided in section 4.4.1. Then, I will formalize the reasoning behind the verification into a type system by augmenting the one provided in section 4.4.2.

5.3.1 Informal description

We have already seen the notion of influencing slots (X), and the constraint environments (ϕ_{pc} and ψ_{pc}). These remain the same, but note that the constraint environments cannot bind any *volatile* slots – arrays, pointers and slots referenced by pointers. We will also need a set of influencing expressions E, which will keep the conditions used in the enclosing block statements, *preserving the value* of the arithmetic expressions used in subscripts present in those conditions. Then, the validation of simple assignments ($xv = e$) needs to take into account the policies of arrays used in the influencing expressions as well as in e. However, these policies shall be constrained by the subscripts used in E and in e, which we will denote as ρ_E and ρ_e respectively. We will additionally have a selector of the array policy for the slot to which we are assigning, which we will denote ρ_{xv}.

Furthermore, if there are array subscripts used on the left-hand side of the assignment, then they also influence how the data represented by xv is modified. Thus, we will need to ensure that $\underline{sub(xv)} \sqsubseteq \underline{xv}$, where *sub* extracts the expressions used in subscripts. Otherwise, we could learn something about the value of a restrictive slot present in the left-hand side subscript when assigning to a less restrictive slot, by just iterating over it afterwards and checking what has changed. Thus the final rule for validating assignment is:

$$\underline{e}_{\phi_{pc} \wedge \rho_e} \sqcup \underline{pc}_{\phi_{pc} \wedge \rho_E} \sqcup \underline{sub(xv)} \sqsubseteq \underline{xv}_{\psi_{pc} \wedge \rho_{xv}}$$

As for structure initialization and structure assignment, these again evaluate to multiple assignments executed atomically. Likewise, validation of the *while* loop invariants remains unchanged.

We also have pointer assignments that change the address referenced by the pointers. Because a value of a slot that influences the policy of the slot referenced by a pointer can change outside an assignment to that pointer, we cannot rely on that value as a selector of the policy. That is why for pointer assignments ($xp = \&x$) we treat the top level structure containing x and all subcomponents of that structure as volatile, and exclude reasoning about it from all environment constraints. Furthermore, even after the assignment, x remains volatile, as it is referenced by a pointer.

5.3.2 Type system

As mentioned in the informal description, some limitations had to be enforced for the verification. First of all, fine grained reasoning about arrays, such as tracking which indices may have what values, cannot be effectively used. There are several reasons of that. One of them is that the core of the validation

process – policy comparison – works on the slot basis, that is, augmentation and substitution only change the slots. If values in different array indexes were to be involved, then for each possible index a new slot and a policy would have to be introduced. Consider, for example, if $\phi \triangleq s[1].c1 = 1 \wedge s[2].c1 = 2$, and we have a policy where the label of some `s.c2` depends on `s.c1`. It is then necessary to have policies instantiated for each possible index. This breaks the previous model, in which policies were instantiated in a compilation step, while here due to dynamic arrays it would not be possible. Furthermore, by the fact that arrays can be used in structures, they can be arbitrarily nested. That would cause explosion of policy sizes in non-trivial cases making the problem intractable. That is why, although index-dependent policies are allowed, the values of array elements are not maintained. This restriction does not change the expressive power of the language, since any program with use of subscripts can be rewritten to this syntax by introduction of auxiliary variables and conditions. Furthermore, the content of arrays is described by the policies these are governed by, which is their most interesting feature, unlike the actual values.

There is one more problem that is going to be avoided in this work – namely, the reasoning about the slots that share memory they reference with other slots – which results from introduction of pointers. If this simplification was not made here, then an advanced extension to the Hoare logic called *separation logic* [28] would have to be used in order to provide a safe over-approximation of the state. Another solution could be performing a shape analysis, such as the one introduced in [26, p. 104], and employing its results in the type system. Both of those approaches are extremely complex constituting a large separate topic of the program analysis science and therefore are out of the scope of this work.

The final version of the type system is available in figs. 5.1 to 5.3. It is already syntax driven – it is built on the basis of the type system given in fig. 4.3 that was devoid of the consequence rule. In comparison to the previous system, there are two new environment variables: E and V.

E is a set of expressions that influence the execution at a given point. It is used keep track of values used in array subscripts in order to allow fine-grained, index-based reasoning – in case an array access is influencing the given execution point and the policy of that array distinguishes the indexes.

V is a set of *volatile* slots, meaning slots which value may change due to some assignment even if these are not used in that particular assignment. These are, for example, pointers (dereference) and slots referenced by pointers. In this type system, also arrays are deemed to be volatile. This prevents reasoning about the value of their elements, which has been already justified.

I will, as previously, describe the type system rule by rule and define the new and changed auxiliary operations. The details that have been already introduced before and remain unchanged will be left out.

Assignments

The plain assignment rule has changed as the array subscripts usage has to be interpreted and the volatile slots have to be excluded from the content reasoning.

One of the changes concerning those involves ϕ, which is used in the policy selection. A weakening operation is performed using the V set in order to exclude the volatile slots. It is used on demand and does not influence the post-

$$
\begin{array}{l}
X, E, V \vdash \{\phi\}; \{\phi\} \\
X, E, V \vdash \{\phi\} decl \{\phi\} \\
X, E, V \vdash \{\phi\} stc \{\phi\} \\
X, E, V \vdash \{\phi\} xv = e; \{\phi'\} \\
\quad \text{if } \begin{cases}
\mathtt{P}_{left} \triangleq ((\mathcal{SE}(E \cup \{xv, e\}, \phi) \backslash V) \Rightarrow \mathtt{P}) \langle X \cup \mathcal{A}(e) \cup \mathcal{AS}(xv)/xv \rangle, \\
i_1, i_2, \ldots, i_m \in \mathcal{IE}(xv) \\
\mathtt{P}_i \triangleq \mathtt{P}[\mathcal{IE}(xv)[i_1]/i_1] \ldots [\mathcal{IE}(xv)[i_m]/i_m] \\
\mathtt{P}_{right} \triangleq \mathtt{P}_i[e/xv] \langle \mathcal{H}(xv)/xv \rangle, \\
\mathtt{P}_{left} \sqsubseteq \mathtt{P}_{right} \\
\phi' \triangleq \exists_{xv'} : xv = e[xv'/xv] \wedge \phi[xv'/xv]
\end{cases}
\end{array}
$$

Figure 5.1: Skip, declaration and simple assignment axioms

condition not to complicate it. As for array subscripts, the modifications that are introduced to support them should never be included in the postcondition, since it might then cause contradictions in the following assignments, as well as escape the original scope (a conditional or loop).

An auxiliary operation $\mathcal{SE}$ is used in order to modify ϕ to take the array subscripts used in expressions of E into account:

$$\mathcal{SE}(E, \phi) = \phi \wedge T_{bool}(\mathcal{I}(E))$$

The above function introduces two new operations that have to be clarified. The $\mathcal{I}$ function takes a set of expressions, possibly containing array accesses, and calculates a mapping from the accessed arrays to all subscript expressions used on each of them:

$$
\begin{aligned}
\mathcal{I}(E) &= \mathcal{I}_{\mathrm{E}}(E, []) \\
\mathcal{I}_{\mathrm{E}}(e \cup E', ind) &= \mathcal{I}_{\mathrm{E}}(E', merge(ind, \mathcal{I}_{\mathrm{e}}(e))) \\
\mathcal{I}_{\mathrm{E}}(\emptyset, ind) &= ind \\
\mathcal{I}_{\mathrm{e}}(e_1 \ op \ e_2) &= merge(\mathcal{I}_{\mathrm{e}}(e_1), \mathcal{I}_{\mathrm{e}}(e_2)) \\
\mathcal{I}_{\mathrm{e}}(x) &= \mathcal{I}_{\mathrm{x}}(x, \epsilon) \\
\mathcal{I}_{\mathrm{x}}(v, pref) &= \emptyset \\
\mathcal{I}_{\mathrm{x}}(si, pref) &= \emptyset \\
\mathcal{I}_{\mathrm{x}}(*ptr, pref) &= \emptyset \\
\mathcal{I}_{\mathrm{x}}(arr[e], pref) &= merge(\{pref + arr \mapsto e\}, \mathcal{I}_{\mathrm{e}}(e)) \\
\mathcal{I}_{\mathrm{x}}(si.x, pref) &= merge(\mathcal{I}_{\mathrm{x}}(si, pref), \mathcal{I}_{\mathrm{x}}(x, pref + si.))
\end{aligned}
$$

For instance, given the set of expressions

$$E = \{(2 + c[4] - a[3]), (3 * a[a[2]] + c[1])\}$$

this operation would return the following mapping:

$$[c \mapsto \{1,4\}, a \mapsto \{3,2,a[2]\}]$$

The *merge* operation takes two maps and combines them keywise – for each key present in either of the two maps the resulting map will contain that key mapped to the union of the values from both maps referenced by that key:

$$merge(m_1, m_2) = \bigsqcup_{arr \in m_1 \cup m_2} [arr \mapsto (m_1[arr] \cup m_2[arr])]$$

For example, if we take:

$$\begin{aligned} m_1 = [&a \mapsto \{a[4*x]\} \\ &b \mapsto \{3\}] \\ m_2 = [&a \mapsto \{2+c, 3\} \\ &c \mapsto \{3\}] \end{aligned}$$

we get the following mapping:

$$\begin{aligned} [&a \mapsto \{a[4*x], 2+c, 3\} \\ &b \mapsto \{3\} \\ &c \mapsto \{3\}] \end{aligned}$$

Then, the map *ind* that maps arrays to a set of subscripts needs to be translated into a boolean predicate acceptable by the type system. As mentioned before, each array and pointer has an implicit component called `index`, and this is where the values used in subscripts will be registered. For each map entry (i.e. for each array/pointer) an alternative of all possible assignments to the `index` component is built – it is an over-approximation. Then a conjunction of the resulting predicates is taken – each array access must have been using one of the values from the disjunction. This translation is realized by the T_{bool} operation:

$$T_{bool}(ind) = \bigwedge_{entr \in ind} \left(\bigvee_{i \in entr_{val}} entr_{key}.index = i \right)$$

For the map from the first example we would obtain the following expression:

$$(c.index = 4 \lor c.index = 1) \land (a.index = 2 \lor a.index = 3 \lor a.index = a[2])$$

where the subscript of $a[2]$ is actually redundant – values of arrays are not maintained.

Finally, in the $\mathcal{SE}$ operation, the results of this function are conjuncted with the original ϕ, thus, including the information from the postcondition. One might be surprised why not only e but also xv is taken as an input to the $\mathcal{SE}$ function. This is necessary, because only the policies of relevant indexes should compared – in the expression for the right-hand side policy (elaborated later on) the `index` components are substituted, so that if T_{bool} evaluated to *true*, then policies for all indexes on the left-hand side would be compared against the selected (by substitution) on the right-hand side.

Another difference with respect to the previous type system is a changed definition of the $\mathcal{A}$ function, that also covers pointers and array accesses:

$$\begin{aligned}
\mathcal{A}(e_1\ op\ e_2) &= \mathcal{A}(e_1) \cup \mathcal{A}(e_2) \\
\mathcal{A}(x) &= \mathcal{A}_{\mathrm{x}}(x, \epsilon) \\
\mathcal{A}_{\mathrm{x}}(v, pref) &= \{pref + v\} \\
\mathcal{A}_{\mathrm{x}}(si, pref) &= \{pref + si\} \\
\mathcal{A}_{\mathrm{x}}(*ptr, pref) &= \{pref + ptr\} \\
\mathcal{A}_{\mathrm{x}}(arr[e], pref) &= \{pref + arr\} \cup \mathcal{A}(e) \\
\mathcal{A}_{\mathrm{x}}(si.x, pref) &= \mathcal{A}_{\mathrm{x}}(si, pref) \cup \mathcal{A}_{\mathrm{x}}(x, pref + si.)
\end{aligned}$$

If the target slot contains array subscripts, then the expressions used in those subscripts will influence how (under which index) the piece of memory represented by that slot (without subscripts) will change. This means that xv should be augmented with the policies of those expressions. The set of expressions used in subscripts is retrieved using the following function:

$$\begin{aligned}
\mathcal{AS}(x) &= \mathcal{AS}_{\mathrm{x}}(x, \epsilon) \\
\mathcal{AS}_{\mathrm{x}}(v, pref) &= \emptyset \\
\mathcal{AS}_{\mathrm{x}}(si, pref) &= \emptyset \\
\mathcal{AS}_{\mathrm{x}}(*ptr, pref) &= \emptyset \\
\mathcal{AS}_{\mathrm{x}}(arr[e], pref) &= \mathcal{A}(e) \\
\mathcal{AS}_{\mathrm{x}}(si.x, pref) &= \mathcal{AS}_{\mathrm{x}}(si, pref) \cup \mathcal{AS}_{\mathrm{x}}(x, pref + si.)
\end{aligned}$$

The change in the definition of $\mathcal{A}$ also calls for defining another function (for augmentation of the right policy) that ignores the subscripts, and only returns all the parent structures (hierarchy) of the input slot:

$$\begin{aligned}
\mathcal{H}(x) &= \mathcal{H}_{\mathrm{x}}(x, \epsilon) \\
\mathcal{H}_{\mathrm{x}}(v, pref) &= \{pref + v\} \\
\mathcal{H}_{\mathrm{x}}(si, pref) &= \{pref + si\} \\
\mathcal{H}_{\mathrm{x}}(*ptr, pref) &= \{pref + ptr\} \\
\mathcal{H}_{\mathrm{x}}(arr[e], pref) &= \{pref + arr\} \\
\mathcal{H}_{\mathrm{x}}(si.x, pref) &= \mathcal{H}_{\mathrm{x}}(si, pref) \cup \mathcal{H}_{\mathrm{x}}(x, pref + si.)
\end{aligned}$$

Nevertheless, values of the subscripts used on the target slot also need to be taken into account, which is done by taking all those subscripts and substituting the relevant `index` policies in the right-hand side policy.

A helper operation $\mathcal{IE}$ creates a mapping from the (fully qualified) arrays to the expressions used in their subscripts:

$$\begin{aligned}
\mathcal{IE}(x) &= \mathcal{IE}_{\mathrm{x}}(x, \epsilon) \\
\mathcal{IE}_{\mathrm{x}}(v, pref) &= \emptyset \\
\mathcal{IE}_{\mathrm{x}}(si, pref) &= \emptyset \\
\mathcal{IE}_{\mathrm{x}}(*ptr, pref) &= \emptyset \\
\mathcal{IE}_{\mathrm{x}}(arr[e], pref) &= \{pref + arr.index \mapsto e\} \\
\mathcal{IE}_{\mathrm{x}}(si.x, pref) &= \mathcal{IE}_{\mathrm{x}}(si, pref) \sqcup \mathcal{IE}_{\mathrm{x}}(x, pref + si.))
\end{aligned}$$

Memory allocation

Allocation of memory to a pointer using the `malloc` function can fail if, for example, the requested amount of memory is not available. In such case the pointer will be *null*. Hence, by using the `malloc` function one may reveal some information about the slots used in the arithmetic expression passed as an argument to that function. Therefore, the memory allocation validation rule closely resembles the rule for simple assignment. The difference is that here pointers are involved, and as there is no reasoning about their value, the precondition does not have to be modified in order to get the postcondition. Furthermore, the memory allocation operation does not initialize the content of the pointer, and for that reason substitution is not used for construction of the right-hand side policy.

Pointer assignments and structure initializers

The logic behind a pointer assignment is to some extent similar to that of structure assignment – it is perceived as a set of assignments of all individual subcomponents, and their effects are aggregated in order to check that also the policies of the subcomponents match. Aside from the novelties already introduced in case of the simple assignment axiom, there is another helper function:

$$\begin{aligned}
\mathcal{TH}(v) &= v \\
\mathcal{TH}(*ptr) &= ptr \\
\mathcal{TH}(arr[e]) &= arr \\
\mathcal{TH}(si.x) &= si
\end{aligned}$$

This function retrieves the top level slot for the given slot. For example, for `si.c1.c2` it would return `si`. All components of that top level slot are taken and made volatile for purpose of validating that assignment. It is necessary to weaken-out all the determinants and compare virtually unconstrained policies in that case, because the determinants can change, and with them the policy of the referenced data, while the policy of the pointer will not change automatically. An example is shown in listing 5.3.

Here, we have a structure instance `x`, where the policy of the second component at the moment of the last assignment is $\{A \rightarrow B\}$. It would be valid if

$$X, E, V \vdash \{\phi\} xp = \texttt{malloc}(a); \{\phi\}$$
$$\text{if} \begin{cases} \mathtt{P}_{left} \triangleq ((\mathcal{SE}(E \cup \{xp, a\}, \phi)\backslash V) \Rightarrow \mathtt{P})\langle X \cup \mathcal{A}(a) \cup \mathcal{AS}(xp)/xp\rangle, \\ i_1, i_2, \ldots, i_m \in \mathcal{IE}(xp) \\ \mathtt{P}_i \triangleq \mathtt{P}[\mathcal{IE}(xp)[i_1]/i_1] \ldots [\mathcal{IE}(xp)[i_m]/i_m] \\ \mathtt{P}_{right} \triangleq \mathtt{P}_i\langle \mathcal{H}(xp)/xp\rangle, \\ \mathtt{P}_{left} \sqsubseteq \mathtt{P}_{right} \end{cases}$$

$$X, E, V \vdash \{\phi\} xp = \&x; \{\phi\}$$
$$\text{if} \begin{cases} c_1, c_2 \ldots c_n \in \mathcal{C}(xp) \\ \mathtt{P}_{left} \triangleq ((\mathcal{SE}(E \cup \{xp, x\}, \phi)\backslash V \cup \mathcal{C}(\mathcal{TH}(x))) \Rightarrow \mathtt{P}) \\ \quad \langle X \cup \mathcal{AS}(xp)/xp\rangle\langle \mathcal{A}(c_1[x/xp])/c_1\rangle \ldots \langle \mathcal{A}(c_n[x/xp])/c_n\rangle, \\ i_1, i_2, \ldots, i_m \in \mathcal{IE}(xp) \\ \mathtt{P}_i \triangleq \mathtt{P}[\mathcal{IE}(xp)[i_1]/i_1] \ldots [\mathcal{IE}(xp)[i_m]/i_m] \\ \mathtt{P}_{right} \triangleq \mathtt{P}_i[c_1[x/xp]/c_1]\langle \mathcal{H}(c_1)/c_1\rangle \ldots [c_n[x/xp]/c_n]\langle \mathcal{H}(c_n)/c_n\rangle, \\ \mathtt{P}_{left} \sqsubseteq \mathtt{P}_{right} \end{cases}$$

$$X, E, V \vdash \{\phi\} stci = init\{\phi'\}$$
$$\text{if} \begin{cases} c_1, c_2 \ldots c_n \in \mathcal{C}(si) \setminus (\mathbf{Arr} \cup \mathbf{Ptr}) \\ cs_1, cs_2 \ldots cs_m \in \mathcal{CS}(si) \\ \phi_1 \triangleq c_1 = e_1 \wedge \phi, \\ \vdots \\ \phi' \triangleq c_n = e_n \wedge \phi_{n-1} \\ \mathtt{P}_{rst} \triangleq ((\mathcal{SE}(E \cup \{e_1, \ldots, e_n\}, \phi)\backslash V) \Rightarrow \mathtt{P})\langle \emptyset/cs_1\rangle \ldots \langle \emptyset/cs_m\rangle, \\ \mathtt{P}_{left} \triangleq \mathtt{P}_{rst}\langle X \cup \mathcal{A}(e_1)/c_1\rangle \ldots \langle X \cup \mathcal{A}(e_n)/c_n\rangle, \\ \mathtt{P}_{right} \triangleq \mathtt{P}[e_1/c_1]\langle \mathcal{H}(c_1)/c_1\rangle \ldots [e_n/c_n]\langle \mathcal{H}(c_n)/c_n\rangle, \\ \mathtt{P}_{left} \sqsubseteq \mathtt{P}_{right} \end{cases}$$

$$X, E, V \vdash \{\phi\} xs_t = xs_s\{\phi'\}$$
$$\text{if} \begin{cases} c_1, c_2 \ldots c_n \in \mathcal{C}(xs_t) \\ \phi_1 \triangleq \exists_{c_1'} : c_1 = c_1[xs_s/xs_t] \wedge \phi[c_1'/c_1], \\ \vdots \\ \phi' \triangleq \exists_{c_n'} : c_n = c_n[xs_s/xs_t] \wedge \phi_{n-1}[c_n'/c_n] \\ \mathtt{P}_{left} \triangleq ((\mathcal{SE}(E \cup \{xs_t, xs_s\}, \phi)\backslash V) \Rightarrow \mathtt{P}) \\ \quad \langle X \cup \mathcal{AS}(xs_t)/xs_t\rangle\langle \mathcal{A}(c_1[xs_s/xs_t])/c_1\rangle \ldots \langle \mathcal{A}(c_n[xs_s/xs_t])/c_n\rangle, \\ i_1, i_2, \ldots, i_m \in \mathcal{IE}(xs_t) \\ \mathtt{P}_i \triangleq \mathtt{P}[\mathcal{IE}(xs_t)[i_1]/i_1] \ldots [\mathcal{IE}(xs_t)[i_m]/i_m] \\ \mathtt{P}_{right} \triangleq \mathtt{P}_i[c_1[xs_s/xs_t]/c_1]\langle \mathcal{H}(c_1)/c_1\rangle \ldots [c_n[xs_s/xs_t]/c_n]\langle \mathcal{H}(c_n)/c_n\rangle, \\ \mathtt{P}_{left} \sqsubseteq \mathtt{P}_{right} \end{cases}$$

Figure 5.2: Complex assignment axioms

```
1 struct sd {
2         int det;
3         int value;
4 } {(self.det == 1 => self.value={A->B});
5    (self.det == 2 => self.value={A->C})};
6 struct sd x = {1, 2};
7 int {{A->B}} *y;
8 y = &x.value;
```

Listing 5.3: Example of sub-component pointer assignment problem

it was a simple assignment. However, since we are extracting a pointer to that component it cannot be valid. Imagine what would happen if we then change both the determinant (`det`) and the `value` (e.g. using structure assignment) – the value would be accessible by the pointer using an invalid (different) policy.

Another peculiarity of the pointer assignment is that it has no effect on the precondition. This is because pointers are considered volatile anyway, so it would be pointless to introduce equations binding them with some values.

As for the structure initializer, its previously-defined form has been updated in the same way as the normal assignment axiom with one small change. Initialization of pointers and arrays belonging to the structure is not supported – these should be skipped in the initializer list. Hence, the policy erasure for all (even structural) subcomponents of the initialized structure instance is even more useful here. It erases the policies of arrays and pointers to which there are no assignments, and thus, there would be no augmentation on them otherwise.

Structure assignment

There are no surprises or additional constructs as far as the structure assignment rule is concerned. It is upgraded to handle pointers and arrays in the same manner as the rule for simple assignments.

Composition and control statements

The composition and control statement rules use an additional operation $\mathcal{V}$ that retrieves all new volatile slots arising from a given statement:

$$
\begin{aligned}
\mathcal{V}(;) &= \emptyset \\
\mathcal{V}(stc) &= \emptyset \\
\mathcal{V}(decl_x) &= \mathcal{V}_{\mathrm{x}}(x, \epsilon) \\
\mathcal{V}(xv = e;) &= \emptyset \\
\mathcal{V}(xp = \&x;) &= \{x\} \\
\mathcal{V}(stci = init;) &= \emptyset \\
\mathcal{V}(xs_t = xs_s;) &= \emptyset \\
\mathcal{V}(S_1\ S_2) &= \mathcal{V}(S_1) \cup \mathcal{V}(S_2) \\
\mathcal{V}(\text{if}(b)\ \{S_1\}\ \text{else}\ \{S_2\}) &= \mathcal{V}(S_1) \cup \mathcal{V}(S_2) \\
\mathcal{V}(\text{while}(b)[\psi]\ \{S\}) &= \mathcal{V}(S)
\end{aligned}
$$

$$\frac{X, E, V \vdash \{\phi\}S_1\{\psi\} \quad X, E, V \cup \mathcal{V}(S_1) \vdash \{\psi\}S_2\{\phi'\}}{X, E, V \vdash \{\phi\}S_1\ S_2\{\phi'\}}$$

$$\frac{X \cup \mathcal{A}(b), E \cup b', V \vdash \{\phi' \wedge b\}S_1\{\psi\} \quad X \cup \mathcal{A}(b), E \cup b', V \vdash \{\phi' \wedge \neg b\}S_2\{\psi'\}}{X, E, V \vdash \{\phi\}\text{if}(b)\ \{S_1\}\ \text{else}\ \{S_2\}\{\psi \vee \psi'\}}$$

$$\text{if} \begin{cases} x_1, \ldots, x_n \in (\mathcal{S}(b) \setminus \mathcal{C}(V)) \\ \phi_1 \triangleq \exists_{x'_1} x'_1 = x_1 \wedge \phi, b_1 \triangleq b[x'_1/x_1] \\ \vdots \\ \phi' \triangleq \exists_{x'_n} x'_n = x_n \wedge \phi_{n-1}, b' \triangleq b_{n-1}[x'_n/x_n] \end{cases}$$

$$\frac{X \cup \mathcal{A}(b), E \cup b', V \cup \mathcal{V}(S) \vdash \{\iota_n \wedge b\}S\{\iota'\}}{X, E, V \vdash \{\phi\}\text{while}(b)[\psi]\ \{S\}\{\iota \wedge \neg b\}}$$

$$\text{if} \begin{cases} (\phi \Rightarrow \iota) \wedge (\iota' \Rightarrow \iota), \\ \iota \triangleq (\phi \backslash dx(S) \wedge \psi) \\ x_1, \ldots, x_n \in (\mathcal{S}(b) \setminus \mathcal{C}(V \cup \mathcal{V}(S))) \\ \iota_1 \triangleq \exists_{x'_1} x'_1 = x_1 \wedge \iota, b_1 \triangleq b[x'_1/x_1] \\ \vdots \\ \iota_n \triangleq \exists_{x'_n} x'_n = x_n \wedge \iota_{n-1}, b' \triangleq b_{n-1}[x'_n/x_n] \end{cases}$$

Figure 5.3: Composition and control statement judgements

The effect of a pointer assignment is the slot on which the address operator is used. It is necessary to include that slot in the volatile set, as mentioned before, because once a pointer references some slot, its value can change in some assignment that is not assigning to that slot.

In this auxiliary operation, $decl_x$ stands for a declaration of some slot x. The rule concerning it uses also another function – $\mathcal{V}_\text{x}$ – that given a slot will return all volatiles that it declares:

$$\begin{aligned} \mathcal{V}_\text{x}(v, pref) &= \emptyset \\ \mathcal{V}_\text{x}(ptr, pref) &= \{pref + ptr\} \\ \mathcal{V}_\text{x}(arr, pref) &= \{pref + arr\} \\ \mathcal{V}_\text{x}(si, pref) &= \mathcal{V}_\text{x}(x_1, pref + si.) \cup \cdots \cup \mathcal{V}_\text{x}(x_n, pref + si.) \\ &\quad \text{if} \quad (pref + si) \text{ defines } x_1, x_2, \ldots x_n \end{aligned}$$

For example, given structure instance `si` defined like that:

```
1 struct sd {
2         int *p;
3 };
4 struct s {
5         struct sd *c1;
6         struct sd c2;
7 } si;
```

the application of the $\mathcal{V}_{\mathrm{x}}$ operation would return $\{si.c1, si.c2.p\}$.

The composition judgement rule in this type system now modifies the environment of the following statement in order to include the new volatile slots from the preceding statement.

Likewise, a modification has been introduced in the control statement judgements. As mentioned before, E is a set of boolean expressions used to determine which index policies are influencing the given statement. Naturally, that set is modified for the bodies of *if* conditionals and *while* loops in their judgements, because it results from their conditions.

In case of both control statement rules, the b', which is the modified condition, is included in E. b' is obtained from b by creating *aliases* of the slots used in array subscripts present in b (these slots are returned by the $\mathcal{S}$ operation), and substituting occurrences of the original slots with those aliases. Thanks to aliasing, values of the subscript expressions are guaranteed to remain unchanged even if the original slots are assigned in the body of the control statement.

The $\mathcal{S}$ operation is defined as follows:

$$\begin{aligned}
\mathcal{S}(e_1\ op\ e_2) &= \mathcal{S}(e_1) \cup \mathcal{S}(e_2) \\
\mathcal{S}(x) &= \mathcal{S}_{\mathrm{x}}(x) \\
\mathcal{S}_{\mathrm{x}}(si.x) &= \mathcal{S}_{\mathrm{x}}(si) \cup \mathcal{S}_{\mathrm{x}}(x) \\
\mathcal{S}_{\mathrm{x}}(arr[e]) &= \mathcal{S}_{\mathrm{s}}(e) \\
\mathcal{S}_{\mathrm{x}}(x) &= \{\} \\
\mathcal{S}_{\mathrm{s}}(e_1\ op\ e_2) &= \mathcal{S}_{\mathrm{s}}(e_1) \cup \mathcal{S}_{\mathrm{s}}(e_2) \\
\mathcal{S}_{\mathrm{s}}(x) &= \mathcal{S}_{\mathrm{sx}}(x, \epsilon) \\
\mathcal{S}_{\mathrm{sx}}(si.x, pref) &= \mathcal{S}_{\mathrm{sx}}(x, pref + si.) \\
\mathcal{S}_{\mathrm{sx}}(v, pref) &= \{pref + xi\} \\
\mathcal{S}_{\mathrm{sx}}(x, pref) &= \{\}
\end{aligned}$$

There is no reasoning about the values of volatile slots, and thus, aliasing them would have little sense. Therefore, all arrays and pointers used in subscripts, as well as the content of their subscripts, are ignored in the $\mathcal{S}_{\mathrm{sx}}$ operation. For example, if we input $a[x] + (*p).c < b[a[z] + y * (*p).c]$ the result would be $\{x, y\}$. Furthermore, all other volatiles – slots referenced by pointers – are removed (with their subcomponents) from the results of the $\mathcal{S}$ operation in the type system. As volatiles are not aliased, the weakening operation does not have to be applied on the precondition before aliasing.

In the judgement for the *while* loop a modified ι_n is used, while it is ι that is the invariant of the loop. It should pose no problem since ι_n is stronger than ι.

Finally, the volatiles resulting from the body of the loop are incorporated into the environment under which this body will be validated.

5.3.3 Tracking values of static arrays

Although I have decided that the type system would not reason about values of elements of any arrays, here I am going to give an idea of how it could be done for the case of static arrays.

First of all, the elements of an array ($arr[0], \ldots, arr[n-1]$, where n is the size) should be treated as separate slots, and hence, duplicate the policy of the top structure, where the array is defined. The policy of each such slot should be modified with condition $arr.index = i$, where i is the index of each element. Of course, if the arrays were nested in the structure, such nested iteration and nested conditions would have to be made as well.

A more difficult problem is recording information about the array elements in pre- and postconditions. Whenever there is an assignment involving arrays there might be some non-constant arithmetic expression in the subscripts. Even if an SMT solver able to reason about arrays and array logic was used, a problem with simple assignment would still remain – what should be existentially quantified in order to replace the old value? The only possibility is to perform that operation on the whole array, thus losing the old information. Hence, only the information from the enclosing block statement conditions or the last assignment to that array would be retained.

Another solution to that problem could be performing a classic data flow analysis with more accurate approximation for arrays and using information from that step as an input to the type system. A similar approach has been described in [17], where an *interval analysis* on the array subscripts is performed and its results are used as an input to a *reaching definitions* analysis.

Chapter 6

Implementation

In this chapter I will present the C^{2}if validation tool that implements the extended type system. First, I will focus on the overview of the architecture of the application, and then I will provide the details of implementation. Finally, I will conclude with several examples showing the output of the program for some interesting input.

6.1 Requirements analysis

As stated in the introduction, the purpose of this work is not only to provide theoretical foundations for content-dependent information flow analysis for C, but also to engineer a tool, a program, that is capable of carrying out that analysis and validate pieces of actual code. In this section I will provide the functional and non-functional requirements of that tool.

6.1.1 Functional requirements

Console interface
: The tool should provide a console interface, i.e. it should be capable of reading from standard input and writing to standard output.

Augmented subset of C as input
: The tool should accept code written in a subset of the C language, augmented with possibility of declaration of policies and loop invariants. The syntax of the acceptable code has been defined in sections 4.1.1 and 5.1.1.

Validation of the input
: The tool should be able to process the input code and check whether there are flows that violate the policies specified in it. It should also check that the loop invariants provided in the code are correct – these invariants facilitate the validation. The rules for validation have been provided both informally and formally in sections 4.4 and 5.3.

Detailed failure reason description
: In case the validation fails – there exists some flow that violates policies or some loop invariant is incorrect – the tool should identify the offending

statement, state the policies or predicates for which the check has failed and provide further information on the circumstances under which the validation rules are violated.

6.1.2 Non-functional requirements

Robustness
The application should always correctly finish and print relevant output provided that the input code follows the syntax. Otherwise, the application should provide information about where and what has caused the problem.

Performance
The application should take minimal memory resources and time necessary to process the input, validate it and provide feedback. In order to facilitate that, the expressions provided to Z3 for verification should be as simple as possible.

Clear design
The design of the application should comply with the best practices and design patterns of *object-oriented programming* and should be easy to understand and maintain. Clear and concise code should be in general preferred over ultimate performance.

Extensibility
The application architecture should allow additions of new functionalities with minimal effort. In particular, the part representing the syntax of the C language should facilitate extending it.

User-friendly output
The output of the application, particularly in case of validation failure, should be clear, concise and easy to understand.

6.2 Architecture overview

The core idea behind the architectural design of the application is delegation of responsibility to the appropriate nodes of the *abstract syntax tree* (AST), which is output from the parser. This means that most of the code (e.g. the type system's auxiliary functions) processing a kind of node is in the class representing that kind. There are several benefits of taking this approach. First of all, as the code is *encapsulated*, it is easier to navigate through it and find the relevant pieces. Another advantage is that classes are organized into hierarchies using interfaces, which means that extending becomes easier – it is enough to follow interfaces and superclasses. Such way of tree processing also results in the fact that the classes of the AST comprise most of the architecture. These classes are presented in fig. 6.1.

The classes of the AST have been grouped into directories (being their *namespaces*). The directory `C2if` has been named after the project. The diagram shows the whole class hierarchy, and some (the most important) associations. By convention, the associations without the multiplicity specified are of

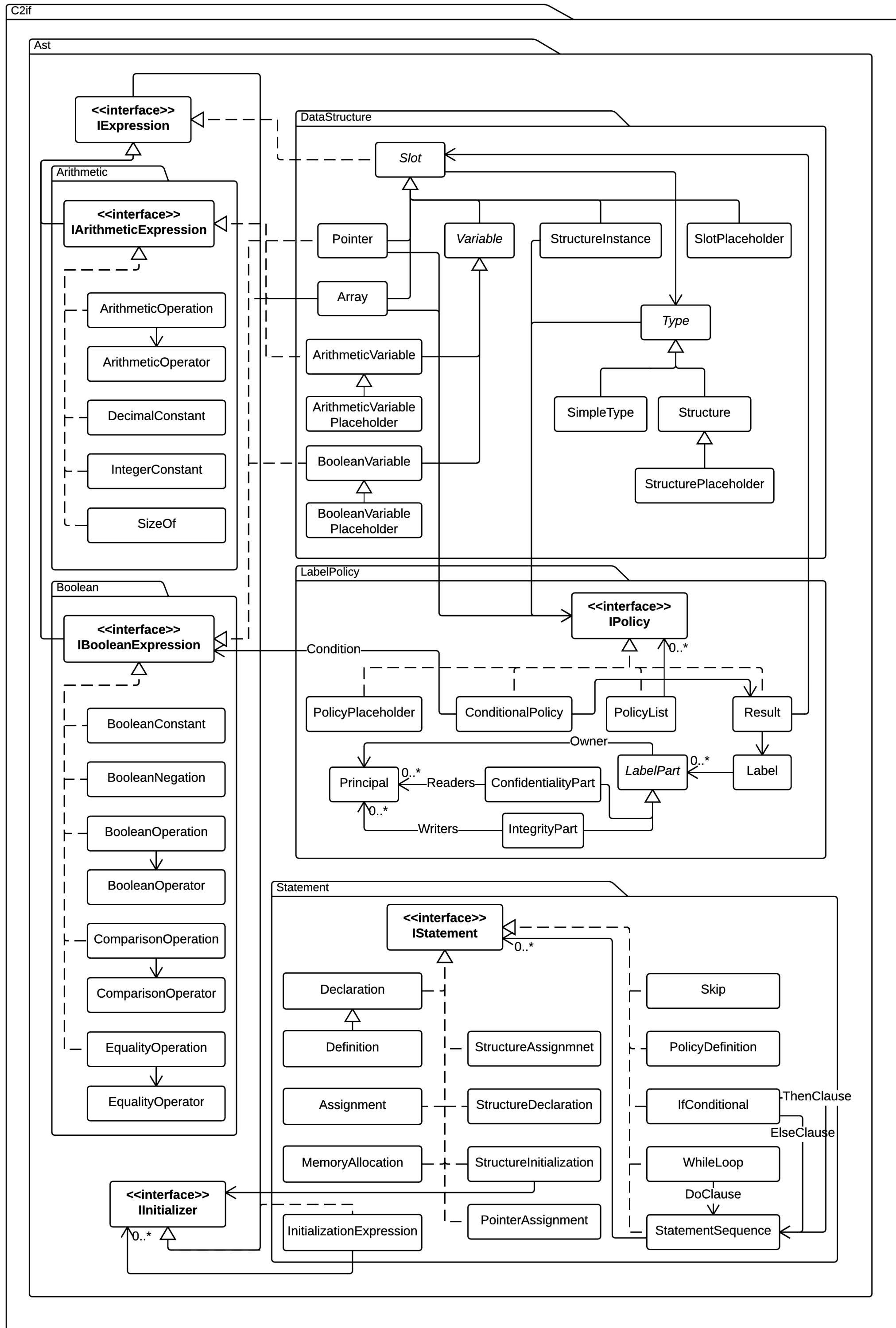

Figure 6.1: Overview of the architecture

cardinality 1. Furthermore, if an association is unnamed then its name is the name of the element that it points to (plural if necessary).

The AST architecture closely follows the syntax given in sections 4.1.1 and 5.1.1. All sequencing recursions from the syntax, or in other words recursions allowing entities to aggregate themselves, have been architecturally translated into the *composite* design pattern.

The expressions are collectively represented by **IExpression**. They are divided into two types: arithmetic expressions (**IArithmeticExpression**) and boolean expressions (**IBooleanExpression**), each residing in its own namespace. For both of these, some *operations*, as well as *operators* related to them, are defined.

The abstract class *Slot* represents all kinds of *slots* – simple *variables*, *structure instances*, *pointers* and *arrays*. The *Variable* class has two subtypes: ArithmeticVariable and BooleanVariable, each implementing different expression interface. Classes Pointer and Array implement both interfaces, which is because these may hold both arithmetic and boolean variables. They, and the StructureInstance also have their own *policy*, although none of them is considered a *type*. The *Type* associated with the slot can be either a SimpleType (integers, floats, booleans) or a Structure. For variables it is always a *simple type*, while for structure instances it has to be a *structure*. There are also some *placeholder* classes. These are instantiated by the parser, whenever some named entity is encountered in the input code, and thereafter replaced with concrete instances during the *compilation* process.

Every policy is an implementation of the **IPolicy** interface. The PolicyList is just an aggregation of policies. There is a ConditionalPolicy class which references a boolean expression as its condition and an instance of Result (which can be a standalone policy as well). The *result* binds a *slot* with a Label. The *label* can consist of many *parts* (*LabelPart*), each having an *owner* (an instance of Principal). The *confidentiality* parts aggregate *readers*, while the *integrity* parts consist of *writers*.

Regarding the *statements*, these are represented by the **IStatement** interface, and their type hierarchy is relatively simple. All statements defined in the syntax and verified in the type system in chapters 4 and 5 are present here as separate classes. The only thing here that deserves explanation is the **IInitializer** interface referenced by the StructureInitialization class. It is extended by **IExpression** and implemented by InitializationExpression, which aggregates the *initializers*. This reflects that lists of expressions, as well as nested initializers, may be part of an initializer used in an aggregate initialization of a structure.

6.3 Helper classes

As already mentioned, not all implementation resides in the classes of the AST – it would not be practical. Instead, a whole set of helper classes under the `Helper` namespace has been introduced. These are depicted in fig. 6.2.

Let us start with C2ifContext – a class that is used throughout the whole solution. It extends the Microsoft.Z3.Context class, which means that all operations creating Z3 expressions usually performed using this class can be executed using C2ifContext. The extension introduces several additional domain specific functionalities. This *context* holds information about the principals and slots

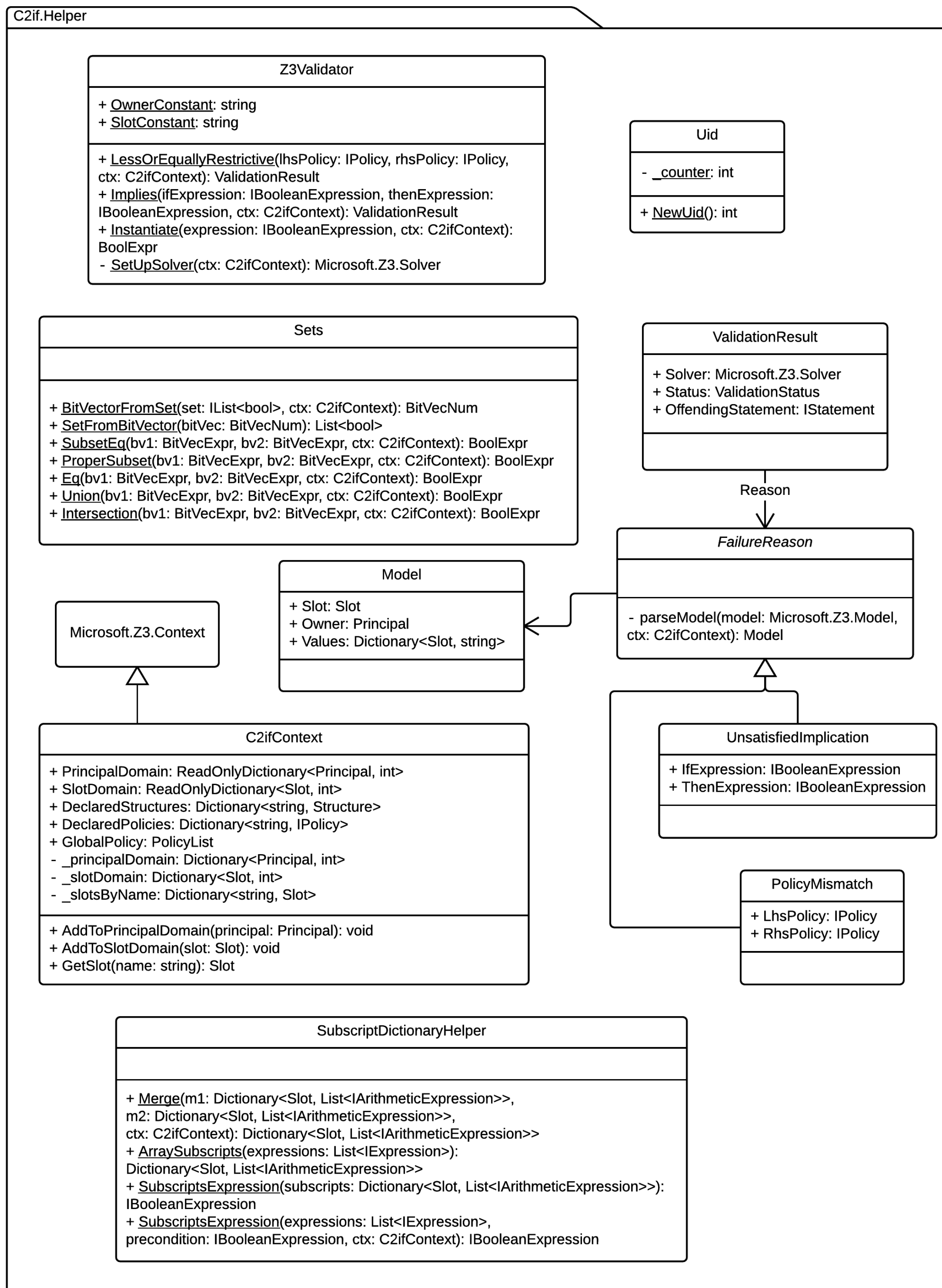

Figure 6.2: Helper classes

used in the input program (direct-modification-protected properties *Principal-Domain* and *SlotDomain*), the declared structures and policies (by their names in *DeclaredStructures* and *DeclaredPolicies* respectively), and the *global policy* that is obtained as the result of the compilation of the input program. The principal and slot domain properties are dictionaries from either Principal or *Slot* to integers informing about the corresponding bit in a Z3 bit vector. C2ifContext provides methods for limited manipulation (addition only) of the domains, and a method for retrieving a slot using its name.

Another important class here is the Z3Validator. It implements the comparison of policies (as in $\mathsf{P}_1 \sqsubseteq \mathsf{P}_2$) in the *LessOrEquallyRestrictive* method and the verification of implication (whether one predicate is stronger than some other) in the *Implies* method. The *Instantiate* method returns a Z3 boolean expression that existentially quantifies all fresh slots in the input expression. It is used in order to obtain Z3 expressions in the *Implies* method, as well as in the ConditionalPolicy using which the global policy receives the precondition (policies cannot contain fresh slots in their condition, so only the precondition is quantified). The *SetUpSolver* method is simply a common code that has to be executed for both validation methods. The *OwnerConstant* and the *SlotConstant* provide names for the v and p defined in sections 4.2 and 4.3, over which the SMT solver will validate the policies.

The formula provided to Z3 is negated, so that the validation succeeds if Z3 responds that it is *unsatisfiable*, and fails otherwise. Thanks to this trick, a model is obtained (it is only available for a satisfiable formula) indicating what went wrong. The model is translated back to the domain of the AST (method *parseModel* of the *FailureReason* class) into an instance of the Model class, which becomes part of the failure reason explanation. The reason can be either an *unsatisfied implication*, or a *policy mismatch*. Naturally, different kind of information is provided in the two subclasses of the *FailureReason*. The reason, together with the *offending statement*, *status* and the Z3 Solver instance (for in-depth information), forms the *validation result*, which is returned from the Z3Validator. Of course, the former two are present only if the status is *incorrect*.

Another widely used helper class is Sets. Its purpose is to provide a common implementation of translation between sets (list of booleans) and Z3 bit vectors, as well as set operations on those bit vectors. Thus we have the *BitVectorFrom-Set* and *SetFromBitVector* methods for translation in both directions. Likewise, we have *SubsetEq*, *ProperSubset*, *Eq*, *Union* and *Intersection* implementing the set operations $\subseteq$, $\subset$, $=$, $\cup$ and $\cap$ respectively.

The SubscriptDictionaryHelper class implements a range of auxiliary operations defined in section 5.3.2. Hence, the *Merge* method obviously implements the *merge* operation, *ArraySubscripts* covers part of the $\mathcal{I}$ operation up to $\mathcal{I}_{\mathrm{E}}$, and the *SubscriptsExpression* overload that accepts a dictionary as an argument realizes the T_{bool} operation. The other *SubscriptsExpression* method combines them all together and makes a conjunction with the input precondition just as the $\mathcal{SE}$ operation does.

Finally, the Uid class merely implements the functionality of generating unique identifiers that are then appended to the slot names in order to obtain fresh slots. These identifiers are incrementing natural numbers, so that the names of the fresh slots do not become to long.

6.4 Implementation details

Now that we have seen the overall design of the solution as well as all the helper classes and methods necessary for understanding the validation process, I will describe the details of the implementation of the type system, including where in the AST classes and how are the auxiliary operations implemented. The base of the description will be the interfaces and the *Slot* abstract class presented in fig. 6.3 – all their implementations and subclasses simply implement them in order to provide the validation functionality. *Slot* implements the **IExpression** interface, and the inherited methods have not been repeated in *Slot* for brevity and clarity.

All the presented interfaces contain a variation of the *Compile* method, which is used in the compilation process. **IExpression** has an overload with an additional *prefix* argument, which is used for subcomponents of structure instances (e.g. when compiling their policy). **IPolicy** has only one version of that method containing the *prefix* argument (which is following the translation presented in section 4.3.2), as the concrete policies always belong to some slot. Statements, on the other hand, only need the compilation method without the *prefix*. As for *Slot*, it has an additional function – *CompileSubscripts*, which is responsible for compilation of array subscripts present in the particular usage of a slot that is represented by its instance.

Another property that the expressions, slots and policies share is that they declare methods for translating their instances into the Z3 domain. For expressions it is the *ToExpr* method. In case of slots, there are two static helper methods *ToBitVector* and *FromBitVector*, which together can perform the translation both ways. The reverse translation is useful for decoding the model that is output from Z3. As for policies, there are two separate methods for forward translation: *ReadersBv* and *WritersBv*. Obviously, the former is for retrieving a Z3 bit vector expression for readers and the latter does the same for writers. Their implementations closely follow the definition of R_{eff} and W_{eff}, where the case differentiation using *iff* is translated into *ite* (if-then-else) expressions of Z3. The state is not passed as an argument in those, as it is implicit in Z3.

As mentioned before, all statements are represented by the **IStatement** interface. Furthermore, as each axiom and judgement of the type system considers some statement, their logic is realized by the implementations of that interface. Most of it – the whole *if* part – is contained in the *Validate* method, where arguments *influencingSlots*, *influencingExpressions* and *volatileSlots* stand for X, E and V respectively. In addition to that, some auxiliary operations, such as $\mathcal{AS}$ and $\mathcal{IE}$ are directly implemented in the body of *Validate* since it was easier to do so using all other AST processing functions, rather than introducing new ones. Regarding other operations that **IStatement** realizes, we have *GetAssignedSlots* that implements dx, and *GetVolatiles* that performs the $\mathcal{V}$ function.

Let us now focus on the auxiliary operations defined for the type system in previous chapters. The **IExpression** interface implements the weakening operation using the *Weaken* method that takes a dictionary as an argument. That dictionary declares by its keys which slots should be weakened out and by the values it provides the fresh slots for replacement. This means that unlike in the definition of the weakening operation, the $\mathcal{C}$ operation for retrieving subcomponents of the weakened slots must be executed before calling this method.

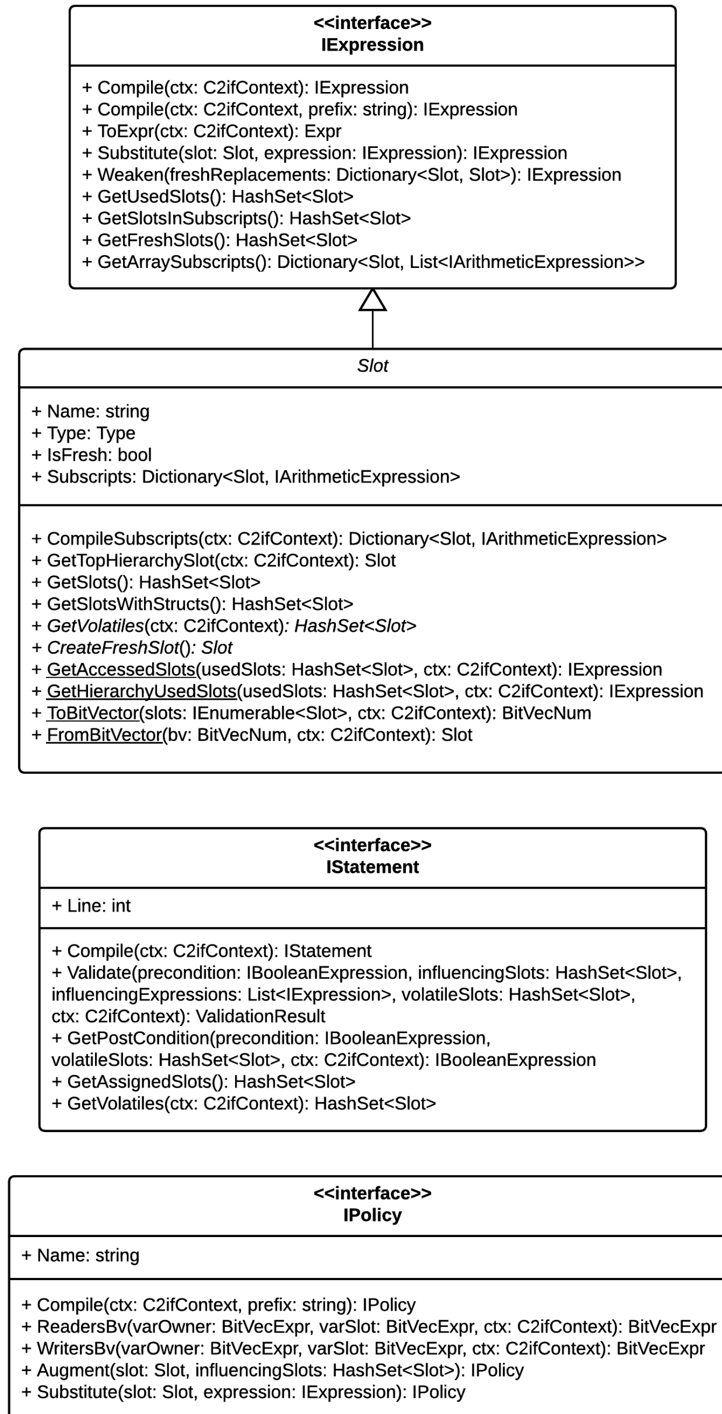

Figure 6.3: Interfaces and abstract classes

The interface also implements the substitution operation with the *Substitute* method. The $\mathcal{S}$ auxiliary function is encoded in *GetSlotsInSubscripts*. The *GetArraySubscripts* method returns a mapping from arrays (of type Array for static, and Pointer for dynamic arrays) to the list of expressions used in their subscripts in the expression represented by **IExpression** instance – it corresponds to the $\mathcal{I}_\mathrm{x}$ and $\mathcal{I}_\mathrm{e}$ operations. Apart from that, there are two other helper methods which are used in other places. These are: *GetUsedSlots* that retrieves all slots present in the expression; *GetFreshSlots* that returns all slots that are fresh, or have been weakened out (which results in them being fresh).

The abstract class *Slot* declares a *Name* of the slot, its Type, an indicator of whether it is *fresh* and a *Subscripts* dictionary. The dictionary holds a mapping from arrays (may be many due to structure nesting) used in the dot-separated accessor of the slot (which is also the name of that slot if the subscripts are omitted) to the expressions used in the subscripts.

The *Slot* implements several auxiliary operations defined for the type system. The *GetAccessedSlots* method, in combination with previously defined *GetUsedSlots*, implements the $\mathcal{A}$ function, while *GetHierarchyUsedSlots*, also with an input from *GetUsedSlots*, has the functionality of the $\mathcal{H}$ operation. The *GetTopHierarchySlot* method realizes the $\mathcal{TH}$ operation, *GetSlots* is equivalent to $\mathcal{C}$ and *GetSlotsWithStructs* implements $\mathcal{CS}$. Finally, the Slot class defines the *GetVolatiles* method which is responsible for performing the $\mathcal{V}_\mathrm{x}$ function. There is one more interesting method, namely *CreateFreshSlot*, which creates a copy of the slot with the *IsFresh* field set to true.

The **IPolicy** interface defines the *Name* property that is used when declaring named, and thus reusable, policies. As for the type-system-defined operations that it implements we have augmentation (*Augment*) and substitution (*Substitute*).

As we already raised the subject of policies, the labels, which are their *results*, and the principals, which the labels are built of, are quite interesting as well. They are presented in fig. 6.4.

All three, labels, their parts and principals provide methods for translation into the Z3 domain. In case of the Label class, the methods *ReadersExpr* and *WritersExpr* correspond to the readers and writers operations on labels defined in section 4.2. These delegate some implementation to the same-named methods of the ConfidentialityPart and IntegrityPart classes respectively. These, in turn, use their internal methods (*WritersBv* and *ReadersBv*) to get the Z3 bit vectors representing the appropriate set of principals. The Principal class has methods for translation in both directions, which is (same as for slots) useful for decoding the Z3 model. It also declares two commodity static methods – *FullBv* and *EmptyBv* – that return Z3 bit vector for the top and the bottom principals respectively. All forward translation methods have a common denominator in the form of the *ToBoolArray* method that transforms a set of principals into a bit vector encoding, which is later on used as an input to the *BitVectorFromSet* method of the Sets helper class to obtain the final result.

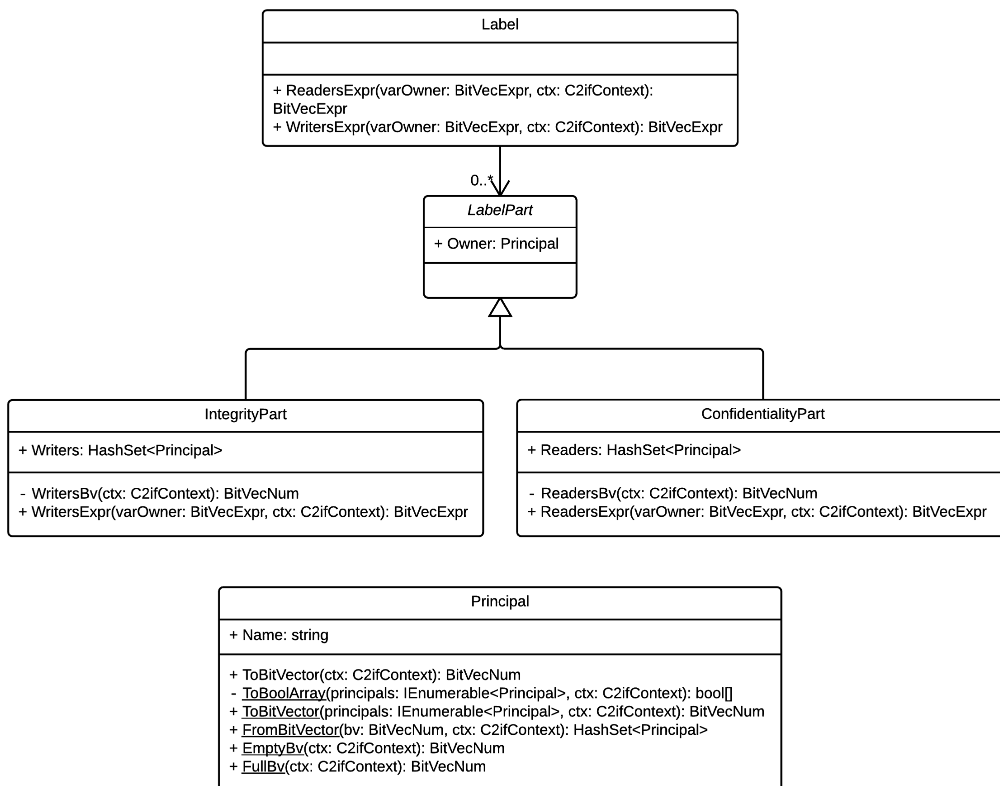

Figure 6.4: The architecture of labels

6.5 Benchmarks

In this section I will show several interesting examples of what the C^{2}if validation tool is capable of. I will start with the example provided in listing 5.1 and modify it, invalidating the information flow in order to see what will be the output of the tool. Next, I will discuss the use case scenario that is the motivation of this work. Finally, I will present other capabilities and aspects of the verification which are not apparent from the previous examples.

6.5.1 Verification of the extended language example

The listing 5.1 is repeated here for quick reference. The name of the *determinant* has been shortened for brevity:

```
1 struct s {
2         int {{Alice->Bob,Chuck}} det;
3         int *data;
4 }{
5         (self.det == 1 => self.data={Alice->Bob});
6         (self.det == 2 => self.data={Alice->Chuck})
7 };
8 struct s input;
9
10 int out_chan{
11         (self.index == 0 => self={Alice->Bob});
12         (self.index == 1 => self={Alice->Chuck})
13 } [2];
14 int counter = 0;
15 while(counter < 2)[counter >= 0] {
16         if(input.det == counter + 1) {
17                 out_chan[counter] = input.(*data);
18         }
19         counter = counter + 1;
20 }
```

When the unchanged code from the listing is provided as the input for the tool, the output is the following:

```
1 Validation has been completed successfully. There are no illegal flows
      ↪ in the program.
```

Let us now change the policy in line 12, so that it looks as follows:

```
1         (self.index == 1 => self={Alice->Bob})
```

This should invalidate the flow, as for *input.det* $==$ 2 the label of `input.data` is $\{Alice \to Chuck\}$, while now for the index equal to `counter` (which is equal to 1), paired with `input.det` by the *if* conditional, the label of `out_chan` is $\{Alice \to Bob\}$.

The output of the tool is presented in listing 6.1, where the policies have been reformatted for better legibility. The first two lines identify the statement which causes the problem. Line 5 gives the reason for which the validation has failed, and the lines that follow provide details. Lines 8 and 9 are the (modified) precondition, where line 9 informs about constraints imposed by usage of subscripts. What follows is the rest of the augmented policy, the *LHS* policy, while the *RHS* policy is the substituted one. For clarification, in

```
1 Validation failed. Offending statement (line 17):
2 out_chan[counter] = input.data;
3
4 Reson:
5 LHS policy is more restrictive than RHS policy
6
7 LHS policy:
8 ((((((counter >= 0) && ((counter:5 == 0) && true)) && (counter < 2)) &&
    ↪ (input.det == (counter + 1))) &&
9   (true && (false || (out_chan.index == counter)))) =>
10    input.det|out_chan={Alice->Bob,Chuck};
11    ((input.det == 2) => input.data|out_chan={Alice->Chuck});
12    ((input.det == 1) => input.data|out_chan={Alice->Bob});
13    ((out_chan.index == 1) => ={Alice->Bob});
14    ((out_chan.index == 0) => ={Alice->Bob}))
15
16 RHS policy:
17 input.det={Alice->Bob,Chuck};
18 ((input.det == 2) => input.data={Alice->Chuck});
19 ((input.det == 1) => input.data={Alice->Bob});
20 ((counter == 1) => out_chan={Alice->Bob});
21 ((counter == 0) => out_chan={Alice->Bob})
22
23 Model:
24 out_chan:{Alice} ->[input.det=2, out_chan.index=1, counter=1]
```

Listing 6.1: Output of the tool for flow validation failure

lines 13 and 14 we can see effects of policy erasure – labels assigned to no slot. The most interesting information, however, comes at line 24, where we have the model for which the policy comparison validation fails. First is the slot, followed by the principal and then the state in the form of mapping from variables to their values. This information allows us to pinpoint the problem in the policy comparison. In line 11 we have that *Alice* designates *Chuck* for `out_chan`, while in line 20 with the matching `counter` it is *Bob* that is designated.

Now, what if we modify the original example so that the `counter` is initialized with -1, instead of 0? Then we get a validation error presented in listing 6.2, which indicates that the invariant is not met. Here, again the first lines of the output identify the *source* of the problem. The most useful information for identifying what *is* the problem, is provided by the two expressions in lines 13 and 16, as well as the model, which this time does not specify any particular slot nor principal. The expression from line 13 does not imply the one from line 16, because the latter rules out the possibility of `counter` being equal to -1 (`counter:5` is a fresh variable).

One might wonder why do we need the invariant at all. What if we remove it? Then we get another information flow validation error, this time stating the following (full output not included for brevity):

```
1 Validation failed. Offending statement (line 17):
2 out_chan[counter] = input.data;
3 (...)
4 input.det|out_chan={Alice->Bob,Chuck};
5 (...)
6 Model:
7 out_chan:{Alice} ->[input.det=0, out_chan.index=-1, counter=-1]
```

```
1 Validation failed. Offending statement (line 15):
2 while ((counter < 2))[(counter >= 0)] {
3         if ((input.det == (counter + 1))) {
4                 out_chan[counter] = input.data;
5         }
6         counter = (counter + 1);
7 }
8
9 Reason:
10 Precondition does not match the postcondition
11
12 Expression:
13 ((counter == (0 - 1)) && true)
14
15 ...does not imply:
16 ((counter >= 0) && ((counter:5 == (0 - 1)) && true))
17
18 Model:
19 n/a:{n/a} ->[counter=-1]
```

Listing 6.2: Output of the tool for invariant validation failure

As we can see, the states for which the policies are compared include one where `counter` is equal to -1. This is because as `counter` is assigned in the loop, and no invariant is provided, the pre-existing information about the counter is weakened out on entering the loop. The problem here again is the policy of `out_chan`, which receives label $\{Alice \rightarrow Bob, Chuck\}$ in augmentation from `input.det`, as this variable is present in the *if* condition.

6.5.2 Verification of the use case scenario

Listing 6.3 shows the code of the use case scenario, which is based on the code of the Receiver Component in Enhanced DLM provided in [19]. The code has been adapted to the policy syntax and limited language discussed in this work. The adaptation process, among other things, excluded the configuration part, where the `handler` was initialized, which is not necessary for the validation – enough information is provided by the policies. Moreover, the function pointer has been replaced by an integer pointer and the function call by an assignment. From the information flow security point of view the code is equivalent, unfortunately the magnitude of changes makes it a modelling case rather than automatic verification.

The code performs a similar functionality and envisions similar validation problems as the code from listing 5.1. The difference is that it concerns both confidentiality and integrity, and re-uses policies declared by name. The verification process for this code completes successfully.

An interesting part here is that we need the assignment to a local slot `DeMux` in line 20, and then use that slot, instead of the `handler` array directly, in the *if* conditional that follows. This is only because the type system does not reason about the values of elements of arrays. If the `handler` array was to be used directly in the condition, then such reasoning would be unavoidable.

It is also possible to model a reverse scenario – a multiplexer that receives some data from multiple input channels and modulates them into one output

```
1 policy GatewayHandler =
2 {(self.protocol==6 =>self.func={TCP->_;TCP<-_});
3  (self.protocol==11 =>self.func={UDP->_;UDP<-_})};
4 policy Gateway =
5 {(self.u.protocol==6 =>self={TCP->_;TCP<-_});
6  (self.u.protocol==11 =>self={UDP->_;UDP<-_})};
7 struct DeMuxType {
8         int protocol;
9         int* func;
10 };
11 struct DeMuxType handler {GatewayHandler}[3];
12 struct inputType {
13         struct info {
14                 int protocol;
15         } u;
16         int buf[65535];
17 } {Gateway} INPUT;
18 int counter = 0;
19 while(counter < 3) {
20         struct DeMuxType {GatewayHandler} DeMux = handler[counter];
21         if(DeMux.protocol==INPUT.u.protocol) {
22                 DeMux.func=INPUT.buf;
23         }
24 }
```

Listing 6.3: The use case scenario code

```
1 policy GatewayHandler =
2 {(self.u.protocol==6 =>self.buf={TCP->_;TCP<-_});
3  (self.u.protocol==11 =>self.buf={UDP->_;UDP<-_})};
4 policy Gateway =
5 {(self.protocol==6 =>self.dataBuf={TCP->_;TCP<-_});
6  (self.protocol==11 =>self.dataBuf={UDP->_;UDP<-_})};
7 struct MuxType {
8         struct info {
9                 int protocol;
10         } u;
11         int buf[65535];
12 };
13 struct MuxType {GatewayHandler} handler;
14 struct inputType {
15         int protocol;
16         int dataBuf[65535];
17 } {Gateway};
18 struct inputType INPUT[3];
19 int counter = 0;
20 while(counter < 3) {
21         struct inputType {Gateway} in = INPUT[counter];
22         struct MuxType {GatewayHandler} Mux = {{in.protocol}};
23         Mux.buf = in.dataBuf;
24         handler = Mux;
25 }
```

Listing 6.4: The multiplexer – the reversed use case scenario code

channel. A valid code for that scenario is presented in listing 6.4. The input channels are modelled as an array, and the output channel handler function is represented as a structure, same as the one being the input for the demultiplexer.

The most important part of this code is translation from the input data structure to the output data structure. This translation needs to be done using structure initialization list – atomically with creation of the structure. However, what is interesting here is that `in.dataBuf` is not part of that initialization. It cannot be as initializing arrays and pointers is not supported. However, it can be assigned afterwards causing no invalid information flow, because `Mux.buf` has a compatible policy, which does not change due to that assignment. Nevertheless, we would not be able to change the value of the `Mux.protocol` that way, as this change would have a side effect of changing the policy of the dependent slot, which is `Mux.buf`. That is why atomic structure initialization is needed.

6.5.3 Other examples

Let us now examine some other examples of the C^{2}if tool execution. We will start with another example of structure initialization provided in listing 6.5, where we have a simple structure with a determinant that decides about the policy for the other component, and we try to incorrectly initialize an instance of that structure. The output from the validation tool for that example is shown in listing 6.6.

```
1 int x {(self.index == 0 => {A->B});(self.index == 1 => {A->C})}[2];
2 struct s {
3         int det;
4         int c;
5 }{
6         (self.det == 1 => self.c={A->B});
7         (self.det == 2 => self.c={A->C})
8 } si = {1, x[1]};
```

Listing 6.5: Example of wrong structure initialization

The offending statement presented in the output has all the components of the structure and their uses in its policy concretely instantiated – it is a by-product of the compilation process that also helps to reason about the policies. The model indicates that the problem concerns the `si.c` variable, and indeed for $x.index == 1$ it receives label $\{A \rightarrow C\}$ in the LHS policy, while in the RHS policy its label can only be $\{A \rightarrow B\}$. There would be no problem if `x[0]` was used in the initialization instead of `x[1]`.

The next example, presented in listing 6.7, shows how array subscripts influence policies concerned in assignments. There are two arrays: `x` with a detailed, disjunctive policy, and `y` with merely a label. There are also two other variables `z` and `z2`, where the policy of `z` is chosen to be no more restrictive than the one of `y`. Then, in the *if* conditional we have a condition involving `x` at the specific index of `z`. The program has some information flow problems that make the assignments inside the conditional invalid.

The output of the validation tool is given in listing 6.8 (only the interesting parts shown). It states that the problem arises already for the `z=0` assignment.

```
Validation failed. Offending statement (line 2):
struct s{
        int si.det;
        int si.c;
}
{((si.det == 2) => si.c={A->C}); ((si.det == 1) => si.c={A->B})}  si = {
        1, x[1]
};

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
((true && (true && (false || (x.index == 1)))) =>
  ((x.index == 1) => x|si.c={A->C});
  ((x.index == 0) => x|si.c={A->B});
  ((si.det == 2) => ={A->C});
  ((si.det == 1) => ={A->B}))

RHS policy:
((x.index == 1) => x={A->C});
((x.index == 0) => x={A->B});
((1 == 2) => si.c={A->C});
((1 == 1) => si.c={A->B})

Model:
si.c:{A} ->[x.index=1]
```

Listing 6.6: Output for wrong structure initialization example

```
int x {(self.index == 0 => {A->B});(self.index == 1 => {A->B,C})}[2];
int y {{A->C}}[1];
int {{A->C}} z = 0;
int {{B->_}} z2 = 0;
if (x[z] == 1) {
        z = 0;
        y[z2] = x[1];
}
```

Listing 6.7: Example of array subscripts influence

```
1 Validation failed. Offending statement (line 6):
2 z = 0;
3
4 Reason:
5 LHS policy is more restrictive than RHS policy
6
7 LHS policy:
8 ((((((z2 == 0) && ((z == 0) && true)) && (z:7 == z)) && (x:9 == 1)) &&
    ↪ (true && (false || (x.index == z:7)))) =>
9 ((x.index == 1) => x|z={A->B,C});
10 ((x.index == 0) => x|z={A->B});
11 (...)
12 RHS policy:
13 (...)
14 z={A->C};
15 (...)
16 Model:
17 z:{A} ->[z=0, x.index=0, z2=0]
```

Listing 6.8: Output for the array subscripts influence example

```
1 Validation failed. Offending statement (line 7):
2 y[z2] = x[1];
3
4 Reason:
5 LHS policy is more restrictive than RHS policy
6
7 LHS policy:
8 ((((z == 0) && ((((z2 == 0) && ((z:10 == 1) && true)) && (z:7 == z:10))
    ↪ && (x:12 == 1))) &&
9  ((true && (false || (y.index == z2))) &&((false || (x.index == 1)) ||
     ↪ (x.index == z:7)))) =>
10   ((x.index == 1) => x|y={A->B,C});
11   ((x.index == 0) => x|y={A->B});
12   ={A->C};
13   z|y={A->C};
14   z2|y={B->_})
15
16 RHS policy:
17 ((x.index == 1) => x={A->B,C});
18 ((x.index == 0) => x={A->B});
19 y={A->C};
20 z={A->C};
21 z2={B->_}
22
23 Model:
24 y:{B} ->[y.index=0, z=0, z2=0, x.index=1]
```

Listing 6.9: Output for the partially corrected array subscripts influence example

Following the model and policies, this is due to the fact that the value of `x` under index 0 influences `z`, and it does, as it is used in the condition. To fix that, let us change the initial value of `z` to 1. Unfortunately, even after doing that the tool complains, which is presented in listing 6.9.

The problem now involves assignment to `y`, which is influenced by `z2` as the LHS policy shows in line 14. It receives the label of `z2`, because `z2` is used in the subscript. In order to ultimately fix the example, the policy of `y` would have to be modified to $\{A \rightarrow C; B \rightarrow _\}$. One last interesting remark about this example is that the assignment `z=0` in line 6 has no effect on the `x[z]` used in the condition, which is also reflected in the output of the validation tool ($x.index == z : 7$).

The last example that I would like to discuss concerns pointers and how they make other variables become volatile. Let us look at the example provided in listing 6.10. Here, we again have a structure with some determinant that decides about the policy for the other component, and a pointer, which after line 7 references the determinant. Then in the last line we have an assignment which does not cause any information flow problems. Unfortunately it is not successfully validated by the C^{2}if tool, which is depicted in listing 6.11.

```
1 struct sd {
2         int det;
3         float c;
4 } {(self.det == 1 => self.c={A->B});(self.det == 2 => self.c={A->C})};
5 struct sd x = {1, 2};
6 int {{A->A}} *y;
7 y = &x.det;
8 int {{A->B}} z;
9 z = x.c;
```

Listing 6.10: Example of volatility of pointers

As we can see, the problem is that `z` is augmented with label $\{A \rightarrow C\}$ (line 9). But why is the state given in the model possible? It is because `x.det` has been weakened out in the precondition, which is visible in line 8. This happened because `x.det` became volatile when its address was assigned to `y` in line 7 of the example code. This is a shortcoming of the type system, in which pointers and all slots they ever reference are not reasoned about, and both are made volatile as an over-approximation.

6.6 Unit tests

A separate project called `C2ifTest` has been created to contain all unit tests for the validation tool. The tests are further split into *test classes* which group them thematically (i.e. which part of the system they do test). There are over 100 tests in the `C2ifTest` project altogether thoroughly testing every aspect of the type system implementation.

The validation tool has been developed according to the *test-driven development* technique – whenever a new functionality was added, a number of tests were also added to the test project and the code of the validation tool was

```
Validation failed. Offending statement (line 9):
z = x.c;

Reason:
LHS policy is more restrictive than RHS policy

LHS policy:
((((x.c == 2) && ((x.det:5 == 1) && true)) && true) =>
 ((x.det == 2) => x.c|z={A->C});
 ((x.det == 1) => x.c|z={A->B});
 y={A->A};
 ={A->B})

RHS policy:
((x.det == 2) => x.c={A->C});
((x.det == 1) => x.c={A->B});
y={A->A};
z={A->B}

Model:
z:{A} ->[x.c=2, x.det=2]
```

Listing 6.11: Output for the volatility of pointers example

corrected until all tests were passed. The tests were the last verification step confirming that the type system works. Some of them were actually useful in identifying problems with the type system itself. Another purpose of the unit tests was ensuring that whenever some new functionality is added it does not break any other that had been implemented previously.

The unit tests are also a great source of examples on how the validation tool works, what is an acceptable input and what are its capabilities. The most interesting examples can be found in classes: IfWhileTest, StructuresTest and PointerArrayTest.

Chapter 7

Conclusions

In this thesis I have suggested an augmentation for the syntax of a subset of the C language that introduces content-dependent policies and constructs necessary to provide analysis for them. I have built a type system, based on which programs written in that language can be processed to determine whether the policies specified in them are obeyed, that is, whether the information flow during their execution will be secure. The kind of analysis presented in this work may be useful not only in the avionics industry, but in all safety-critical systems, where the information flow security plays a major role.

The other outcome of this thesis is the C^{2}if verification tool that implements the type system and meets all the requirements specified in section 6.1. The benchmarks of the tool show that it can be useful in finding flaws in programs that violate their policies. Its architecture is clear and provides straightforward possibilities of extension for the future. The tool, along with the type system, is a proof of concept that a secure content-based verification is attainable even for complex imperative languages, such as C. It also shows, however, that the task is challenging and it is hard to achieve a safe over-approximation in presence of arrays and pointers.

The technologies selected for the implementation, in particular Z3, proved to be appropriate for the task. Although the C# API of Z3 lacks a detailed documentation, the tutorials and manuals for the SMT-LIB interface were enough to grasp the main principles and apply them in practice. Also the performance of Z3 was above expectations – for most of the simple programs provided in listings and used in unit tests the answers were delivered in milliseconds. As for ANTLR, it was surprisingly easy to write rules for parsing the policies and the C-like code, as well as embed the code for creating AST in them.

The chosen iterative, test-driven development method was invaluable in ensuring correctness of the implementation. Moreover, on numerous occasions this method helped to identify problems with the type system itself. For example, no sooner than during the tests of implementation was it discovered that some sort of atomic assignments, or initialization at least, is unavoidable for programs to actually perform something useful.

Future work

Although the type system and the tool are fully operational, they are incomplete in a way. There are some parts of the C language that are not included, and many features that would be useful in the analysis and in terms of automation are missing. Without them, the verification applications are limited to simple case study programs or to modelling, which is not enough for the avionics industry or any other businesses concerned with safety-critical systems.

Functions

The most noticeable part of C that is not included in this work are functions. These have been left out due to unexpected complexity of the type system, resulting from possibility of nesting structures, also with arrays and pointers, and changes in it and in the implementation that would be required otherwise. Let me here explain what should be done in order to support the functions of the C language.

A call of a function assigns *actual arguments*, used in that call, to the *formal arguments*, specified in the function declaration and used in its body. Of course, it should be possible to specify policies for the arguments in the signature of a function, just as for any other slots. In order to verify the flow, the argument assignments should be type-checked using appropriate assignment rules from the type system presented in section 5.3.1. Thus, either function calls should be split into several assignments for type checking, or the type system should be rewritten, further extracting logic governing assignments into auxiliary operations so that it could be reused for function calls.

The function declaration should include a *begin policy*, that would be assumed as the initial program counter policy inside the body of the function. This would ensure that there is no indirect information flow violation inside the function, particularly concerning global variables. In order to simplify typing for the local variable declaration, their policy should be automatically joined using the augmentation operation with the begin policy (similarly to what is done in Jif). To encapsulate verification of functions, the scope of the begin policies should be limited to the formal arguments, or instead a begin *label* (an unconditional policy) could be required.

Although Jif also introduces *end labels*, these will not be necessary for C, since there are no exceptions in this language or any other ways in which a function may indirectly influence the program counter of the caller. The application may of course crash, but it is a covert channel and poses a different kind of problem, not discussed in this work.

Multiple return statements, possible in functions, redirect the information flow in a way that only some execution paths will go through the part of the code after the return statement if it is enclosed in a conditional or a loop. Thus, the program counter for a statement that follows should be augmented with the policies of program counters of all return statements that precede it. Note that the end of a function is an implicit return statement if the function is void.

Functions can contain recursive calls, which usually require interprocedural analysis to approximate values on which they operate. In order to avoid the problem, the programmer should be required to provide pre- and postconditions.

Other parts of C

Some minor parts of the C language, such as other primitive data types and control statements, should also be incorporated to allow verification of programs without making changes in their code other than specification of policies. Inclusion of primitive data types is straightforward and does not change the logic of the type system, likewise, the control statements similar to *if* conditionals and *while* loops. However, the *goto*-like statements are a bit more challenging.

The simplest cases of the *goto*-like statements are *continue* and *break*. These should be treated in a similar way as multiple return statements, though their influence should be limited to the control block to which they refer, as these blocks enclose the execution paths on which they operate.

The *goto* statement itself, which unlike in Java is supported in C, changes the paths of execution of the program in the least predictable way – it can actually simulate both loops and functions. This means that, similarly to functions, the *goto labels* would need begin policies, and following the solution for *while* loops, also specification of an invariant would be necessary. In case of the latter, the invariant would constitute the full precondition of the *goto* target statement since there is no possibility to infer even part of the state without fixed-point analysis (imagine *goto* calls in multiple places).

Downgrading

Downgrading for DLM has been described in details in [25]. It is the part of DLM that could be introduced with the least effort, though the prerequisite of doing so would be adding a way to express the principals on whose behalf the code is executed (the authority), so that downgrading could be controlled. Just as in DLM, it should be possible to not only downgrade results of expressions (used in assignments), but also the indirect information flow, also known as program counter downgrading.

The fact that, in contrast to labels, policies in the type system introduced in this work contain conditions does not make downgrading more difficult than it already is for assignments. The restriction of the part of the policy that can be downgraded by an authority (a principal) can be done in the same way as in DLM.

The last part of downgrading that could be useful from the perspective of the industry are checked endorses, which have been introduced in Jif. These allow specifying a boolean expression, an integrity check, for a variable endorsement. The block of code that follows is only executed if the integrity check succeeds, and the endorsed variable cannot be modified in that block. Although this scenario seems to be a perfect opportunity to take advantage of information about the values gained in the content-based analysis, it is not. The reason behind this concept is to allow integrity checks with automatic endorsement on any input data, and not to match some content-dependent policies. Checking those expressions statically would defy the purpose for which they were introduced.

Policy inference

A feature that could be the selling point for Airbus and other companies in the safety-critical systems industry is policy inference. This is because it increases

automation of verification in a way that only minimal modifications to the code are needed. It is, in the same time, the most challenging feature to implement.

In DLM [24] labels are inferred by binding all label comparisons resulting from the code into a label constraint system. However, in the simple scenario without policy inference, discussed in this work, I take advantage of the fact that policies governing statements are fixed and thus the statements are independent. This allows checking their correctness one by one, which not only facilitates understanding the output of the verification and pinpointing a potential problem, but also enhances performance. If we entangle the constraints, then these two benefits would be much harder to achieve. Furthermore, verification could become infeasible for larger programs.

A solution could be pruning of the policies, for use in comparisons, to only those parts that are absolutely necessary. The indicator, in deciding whether a part is necessary or not, should be checking whether its removal would change the result of policy comparison for any state. Pruning could be also applied even if policy inference is not introduced to improve on performance and understandability of the current solution.

However, even if the policies are pruned to minimum, there are other issues. The label parts of the inferred policies could be described in the same way as they are now (with if-then-else expressions of Z3), but using existentially quantified variables, so that any set of those variables that satisfies all the constraints would be acceptable as solution. Unfortunately, the conditional parts introduce a huge complexity. In order to allow inferring them, total functions returning a boolean value would have to be introduced. Z3 and other SMT solvers do support this kind of functions and are capable of devising them based on constraints. However, to allow any kind of condition they would have to be defined over the whole state — all slots present in the code. This could make the analysis intractable especially for larger programs, hence, I would recommend supporting only partial inference, that is, inference of simple unconditional policies — labels.

Polymorphism

In terms of interference with the code, probably the second most desired feature of verification is policy polymorphism. As mentioned in chapter 2 it allows defining methods without specifying the return policy, nor the policies of the arguments. The concept is quite simple, yet powerful, as polymorphic methods can be used in different contexts with different policies.

In the content-dependent framework, the return policy can be derived from the arguments by augmenting the return value with their labels. This approach would work even if the arguments were structures and had incompatible types with the return value, because by augmentation the policy of the return value would "reference" the policies of the arguments.

Cryptography

To further facilitate integration of static information flow verification with existing software, one could introduce automatic declassification on encryption. This idea was proposed in a DLM case-study by Askarov and Sabelfeld [11],

and assumed that the cryptographic function is perfect – no information about the plaintext can be derived from the ciphertext.

Hicks et al. [18] demonstrated a solution on a simple functional language and implemented it as an extension to Jif. The solution is to introduce a way of delegating authority to the cipher module so that it can declassify the input data on behalf of the principal that calls the encryption function. Yet better solution could be a deeper integration with the cryptography libraries so that this trust relationship would be automatic. The implementation of the encryption function should, of course, ensure that no information can be derived even for low entropy input, which can be achieved using padding and nonces or salts.

External integration

The final step for automation of verification of safety-critical software could be pushing the policy specification outside of the code.

In a MILS-based architecture this could be done by declaring the interfaces of the components against which the software inside those components would be validated. Of course, the input and output of the programs would also have to be well-defined and annotated with policies, and possibly restricted to the IPC channels only.

On the security gateway level, the policies of interconnecting components should be checked and only communication that is a restriction should be allowed. The gateway could examine the content of the messages to decide what policy applies on both ends.

This way we would ensure that there are no illegal flows due to interface incompatibility, which might happen if the components disagreed about the policy.

Fixed-point analysis

The last point of future work worth of noting with respect to this thesis is taking advantage of classic fixed-point analysis, e.g. interval analysis, and using it as an input to the information flow security analysis. A similar approach has been taken for improving performance of a SAT-based program verification in [15].

The most apparent benefit of this development would be avoiding the invariants and the pre- and postconditions that have to be manually specified by the programmer for loops and functions respectively. Secondly, the input provided to the SMT solver would be simplified, as no existential quantifiers would be necessary – these are only used to ignore part of the value reasoning and to switch between states.

A disadvantage is that the fixed-point analysis might not be as precise as the manually provided invariants and conditions. What is more, it might not capture logical interdependencies between variables that are the basis for policy matching.

Index of notation

$\bot$, 17
$\top$, 17
_, 17
*, 17

■, 22

a, 14
$\mathcal{A}$, 25, 41
Arr, 35
arr, 34
$\mathcal{AS}$, 41

b, 14

$\mathcal{C}$, 27
$\mathcal{CS}$, 28

$decl$, 14
dx, 31

E, 37, 38
e, 14

$\mathcal{H}$, 41

$\mathcal{I}$, 39
$\mathcal{IE}$, 42
$init$, 14

L, 17
$L_1 \sqsubseteq L_2$, 17

$merge$, 40

n, 14

$O \rightarrow R$, 17
$O \leftarrow W$, 17

p, 17

ϕ, 20, 23
ϕ', 23
$\phi \Rightarrow \mathtt{P}$, 20
ϕ_{pc}, 22
$\phi \backslash W$, 31
$\mathtt{P}$, 20
$\mathtt{P}_1 \sqsubseteq \mathtt{P}_2$, 21
$\mathtt{P}_1 ; \mathtt{P}_2$, 20
$\mathtt{P}\langle X/x \rangle$, 23, 25
$\mathtt{P}[e/x]$, 23, 25
ψ, 15
ψ_{pc}, 22
Ptr, 35
ptr, 34

readers, 17
R_{eff}, 21

S, 14
$\mathcal{S}$, 46
$\mathcal{SE}$, 39
si, 14
σ, 21
Slot, 15, 35
stc, 14
$stci$, 14
Str, 15, 35

T, 20
T_{bool}, 40
$\mathcal{TH}$, 42

V, 38
$\mathcal{V}$, 45
v, 14
Var, 15, 35

writers, 17
W_{eff}, 21

X, 20, 22, 23
$X : L$, 20
x, 15, 34
xp, 34
xs, 14
xv, 14

Index

abstract syntax tree, 12, 50
address operator, 34
aliasing, 46
ANTLR, 12
ARINC 811, 2
atomic assignment, 28
augmentation, 23, 25

begin label, 8

CAP certification, 3
CCDB certification, 3
C^2if, 6
Common Criteria, 2
compilation, 21, 55

Decentralized Label Model, 5, 7
declassification, 5
demultiplexer, 4
dereference operator, 34
downgrading, 5, 9

end label, 8
endorsement, 5
Evaluation Assurance Level, 2

fresh slot, 23
fully qualified name, 14

global policy, 20

influencing expressions, 37, 38
influencing slots, 22, 23
Integrated Modular Avionics, 2
IPC channel, 3

Jif, 5, 7

label, 5, 7, 17
loop invariant, 15

memory allocation, 42
Multiple Independent Levels of Security, 2

owner, 7, 17

partition, 2
PikeOS, 3
pointer assignment, 34, 42
policy, 4, 19
policy condition, 19
policy erasure, 25
policy inference, 8
policy result, 19
policy scope, 15, 19
polymorphism, 8
port, 2
principal, 7, 17
program counter label, 8

reader, 7, 17

secure gateway, 3
security domain, 2
security policy, 3
separation kernel, 3
simple assignment, 23
slot, 14, 34
SMT solver, 12
structure assignment, 30
structure initialization, 15, 26, 44
structure instance, 14
subscript, 34
substitution, 23, 25

variable, 14
volatile slot, 37, 38

weakening, 31
writer, 7, 17

Z3, 12

Bibliography

[1] 10th International Satisfiability Modulo Theories Competition, SMT-COMP 2015. Summary. http://smtcomp.sourceforge.net/2015/results-summary.shtml?v=1435577347.

[2] CVC4 SMT solver. http://cvc4.cs.nyu.edu/.

[3] GOLD Parsing System. http://www.goldparser.org/.

[4] Jif: Java + information flow. http://www.cs.cornell.edu/jif/.

[5] PikeOS Hypervisor. https://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/.

[6] Yices 2 SMT Solver. http://yices.csl.sri.com/.

[7] Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[8] Commercial Aircraft Information Security Concepts of Operation and Process Framework, ARINC Report 811. Technical report, Airlines Electronic Engineering Committee, 2005.

[9] *Common Criteria for Information Technology Security Evaluation, ISO/IEC 15408-1:2009.* International Standards Organization and International Electro-technical Commission, 2009.

[10] *Programming Languages — C, ISO/IEC 9899:2011.* International Standards Organization and International Electro-technical Commission, 2011.

[11] Asian Askarov and Andrei Sabelfeld. Security-typed languages for implementation of cryptographic protocols: A case study. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 3679 LNCS:197–221, 2005.

[12] Clark Barrett, Aaron Stump, and Cesare Tinelli. The SMT-LIB Standard: Version 2.0. Technical report, Department of Computer Science, The University of Iowa, 2010.

[13] D. Elliott Bell and Leonard J. LaPadula. Secure Computer Systems: A Mathematical Model. Technical report, MITRE Corporation, 1973.

[14] Kenneth J. Biba. Integrity Considerations for Secure Computer Systems. *Proceedings of the 4th annual symposium on Computer architecture*, 5:66, 1977.

[15] Bruno Cuervo Parrino, Juan Pablo Galeotti, Diego Garbervetsky, and Marcelo F. Frias. A dataflow analysis to improve SAT-based bounded program verification. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7041 LNCS:138–154, 2011.

[16] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 20:504–513, 1977.

[17] Thomas Gross and Peter Steenkiste. Structured dataflow analysis for arrays and its use in an optimizing compiler. *Software: Practice and Experience*, 155(June 1989):133–155, 1990.

[18] Boniface Hicks, David King, and Patrick Mcdaniel. Declassification with Cryptographic Functions in a Security-Typed Language. *Science*, 2004.

[19] Kevin Müller, Ximeng Li, Flemming Nielson, Hanne Riis Nielson, and Georg Sigl. Secure Information Flow Control in Safety-Critical Systems. 2014.

[20] Kevin Muller, Michael Paulitsch, Reinhard Schwarz, Sergey Tverdyshev, and Holger Blasum. Mils-based information flow control in the avionic domain: A case study on compositional architecture and verification. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, 2012.

[21] Kevin Muller, Michael Paulitsch, Sergey Tverdyshev, and Holger Blasum. MILS-related information flow control in the avionic domain: A view on security-enhancing software architectures. *Proceedings of the International Conference on Dependable Systems and Networks*, 2012.

[22] Andrew C. Myers. Jif Reference Manual. http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html.

[23] Andrew C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '99*, pages 228–241, 1999.

[24] Andrew C. Myers and Barbara Liskov. A Decentralized Model for Information Flow Control. (October), 1997.

[25] Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9, 2000.

[26] Flemming Nielson, Hanne Nielson Riis, and Chris Hankin. *Principles of Program Analysis*. Springer Science & Business Media, 1999.

[27] Michael L. Olive, Roy T. Oishi, and Stephen Arentz. Commercial aircraft information security-an overview of ARINC report 811. *AIAA/IEEE Digital Avionics Systems Conference - Proceedings*, pages 1–12, 2006.

[28] John C. Reynolds. Separation logic: a logic for shared mutable data structures. *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, 2002.

[29] Hanne Riis Nielson and Flemming Nielson. Content-Dependent Information Flow Control. (Draft), 2015.

[30] Hanne Riis Nielson, Flemming Nielson, and Ximeng Li. Disjunctive Information Flow. 2014.

[31] John Rushby. Separation and Integration in MILS (The MILS Constitution). 2008.

[32] Dennis Volpano, Geoffrey Smith, and Cynthia Irvine. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167 – 187, 1996.