Alexander Jesner, Bakk. rer. nat.

Secure Program Partitioning for a Trusted Execution Environment

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Roderick Bloem, Univ.-Prof. M.Sc. Ph.D.

Name of the institute

Institute for Applied Information Processing and Communications

Second Supervisor

Daniel Hein, Dipl.-Ing.

Graz, April 2015

Alexander Jesner, Bakk. rer. nat.

Sichere Programmpartitionierung für eine vertrauenswürdige Laufzeitumgebung

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Informatik

eingereicht an der

Technischen Universität Graz

Betreuer

Roderick Bloem, Univ.-Prof. M.Sc. Ph.D.

Institutsname

Institut für Angewandte Informationsverarbeitung und Kommunikationstechnologie

Zweitbetreuer

Daniel Hein, Dipl.-Ing.

Graz, April 2015

ABSTRACT

Classical programming paradigms revolve around running both code operating on confidential data and code operating on „don't-care-if-public" data in a single execution environment. This may lead to security issues and information leakage.

Adding information flow policies to a program and applying certifying compilation with respect to a lattice-based information flow system as described by Denning [Den76], allows for writing programs with strong guarantees in terms of data flow. Nevertheless the code is still executed in a single environment. One possibility to increase the security is to separate a program into two cooperative parts, where confidential operations are executed in a special *trusted environment* and other operations are executed in a *normal environment*. Partitioning a program in such a way is usually a manual process.

This thesis aims to automate the separation of programs annotated with information flow policies. We automatically partition a program into two cooperative parts, with confidential statements residing in a partial program that is only available on a trusted environment. Thus, the burden of writing distributed code is taken off the shoulders of the developer and shifted to the compiler.

In this thesis we introduce the newly developed *Bebop* Compiler that takes a security-annotated JIF [Mye99a] program as input, automatically determines a secure partitioning from information flow policies, and produces two partial programs that communicate via calls to a runtime. We show in detail how the *Bebop* Compiler and the *Bebop* Runtime Environment work. With the help of two examples we show that the *Bebop* Compiler can be used to securely partition real world programs.

KURZFASSUNG

Klassische Programmierparadigmen unterscheiden nicht zwischen Code welcher mit sensiblen Daten arbeitet, und Code der auf unkritischen Daten operiert. Beides wird in der selben Laufzeitumgebung ausgeführt. Dies kann Sicherheitsbedenken aufwerfen und zu Informationslecks führen.

Erweitert man ein Programm um Informationsflussrichtlinien und verwendet einen Compiler der das Informationsflussmodell nach Denning [Den76] berücksichtigt, kann man starke Garantien für den Datenfluss im Programm geben. Der Programmcode wird dennoch in ein und derselben Laufzeitumgebung ausgeführt. Um mehr Sicherheit zu gewährleisten bietet es sich an, das Programm in zwei voneinander abhängige Teile zu aufzuteilen. Kritische Operationen können nun in einer *vertrauenswürdigen Laufzeitumgebung* ausgeführt werden, die restlichen Operationen können in einer *regulären Laufzeitumgebung* ausgeführt werden. Üblicherweise muss diese Aufspaltung manuell vorgenommen werden.

Diese Arbeit beschäftigt sich mit der automatisierten Aufspaltung von Programmen die mit Informationsflussrichtlinien versehen wurden. Wir teilen Programme automatisch in zwei kooperative Teile, wobei das Teilprogramm für kritische Operationen nur in einer vertrauenswürdigen Laufzeitumgebung verfügbar ist. Das Schreiben von verteiltem Programmcode muss nun nicht mehr vom Entwickler übernommen werden sondern kann auf den Compiler abgewälzt werden.

Wir stellen in dieser Arbeit den neu entwickelten *Bebop* Compiler vor. Als Eingabe dienen mit Informationsflussrichtlinien versehene JIF [Mye99a] Programme. Der Compiler bestimmt automatisch eine Programmpartitionierung welche die angegebenen Informationsflussrichtlinien berücksichtigt und produziert zwei Teilapplikationen die mit Hilfe der Bebop Runtime kommunizieren. Wir gehen in dieser Arbeit näher auf die Funktionsweise des *Bebop* Compilers ein. An Hand von zwei praxisnahen Beispielen zeigen wir, dass der *Bebop* Compiler dazu verwendet werden kann, Programme sicher aufzuspalten.

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Place

Date

Signature

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Ort

Datum

Unterschrift

ALEXANDER JESNER

SECURE PROGRAM PARTITIONING

FOR A TRUSTED EXECUTION ENVIRONMENT

CONTENTS

ACKNOWLEDGEMENTS

I want to thank my supervisors Roderick Bloem and Daniel Hein for their guidance and their help, and for supporting me in writing this thesis.

Further thanks go to Johannes Winter and Bernd Prünster for their feedback, when I had the need to discuss technical problems. I also want to thank Christof Sirk for convincing me to study at the Graz University of Technology.

Last but not least, I want to thank my family for their support throughout the years, and I want to thank my parents for buying me my first computer.

Alexander Jesner
Graz, Austria, April 2015

PUBLICATION NOTICE

Parts of this thesis have also been submitted to ESORICS 2015[1]. This includes (but is not limited to) images and descriptions used in chapter 1, the placement labelling mechanism described in chapter 4 and the Time-Based One-Time Password (TOTP) example presented in chapter 6.

[1] http://esorics2015.sba-research.org/

CHAPTER 1

INTRODUCTION

1.1. Background

This thesis is about automatically partitioning a program into two separate parts that execute in different execution environments. A program is partitioned such that confidential data and critical statements are only executed in an execution environment that offers stronger security guarantees regarding confidentiality and integrity.

In recent years, security incidents have become a more and more severe problem. In an all-connected world, password leaks and authentication breaches can easily lead to severe damage: average users often reuse a password for different services [FH07] and a broken login of a photo sharing website may thus threaten this user's online banking account. Leaked passwords may cause a *domino effect* [IWS04; Hon12], where not only the end user may experience financial loss, studies suggest that announcements of security breaches also have a negative effect on the stock value of companies [CMR04]. This is especially true if the breach includes leakage of confidential data [Cam+03].

When talking about security in this thesis, we focus on *data confidentiality* and *data integrity*.

Security is hard to implement and must already be considered while designing software [DS00]. In theory, software systems claim to offer a certain kind of security; in practice, the implementation is often faulty or unintentionally leaks information. For example, the *NIST National Vulnerability Database* (NVD) reports 2,044 incidents of type „Software Flaws“ in the period of May 2014 to August 2014 [NISa]. Such flaws can be grouped in categories, e.g. *denial of service*, *unauthorised destruction of data*, *unauthorised disclosure of data*, etc. [Lan+94]. Recent trends suggest, that information leakage is increasing in its prevalence, as Figure 1.1 shows.

If information is leaked—either directly or through covert channels—security measures may be worked around, and the system must be considered broken. As already described

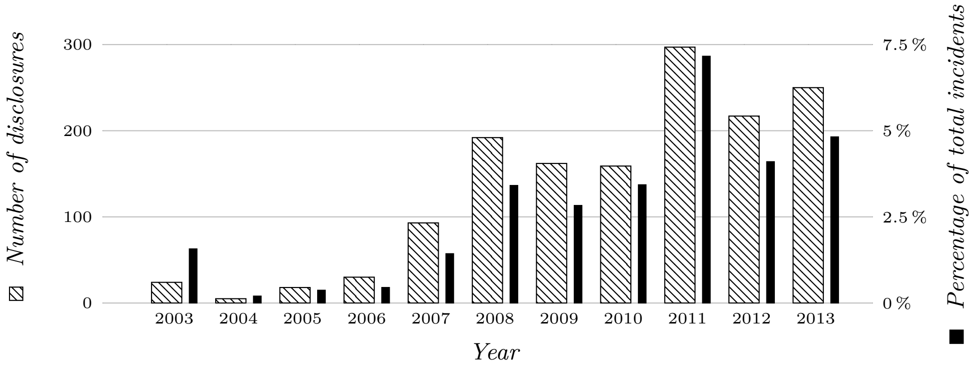


Figure 1.1.: Software vulnerabilities of type *Information Leak / Disclosure* as reported by the NVD [NISb]. Both, absolute and relative values show an increasing trend.

above, information leaks may lead to disclosure of confidential information and in turn may lead to a domino effect.

One possibility to mitigate information leakage in implementations is to avoid leakage at all, by refusing to compile code that discloses too much information. This process of *certified compilation* is achieved with the help of static analysis of the program source code. The compiler calculates the *secure information flow* [DD77] over all data in the program and ensures that an executable output is only produced, if the information flow is *non-interfering* [GM82], with respect to annotated security policies which define the readers and writers of stored information. Both properties are described in detail in chapter 2.

An extension to calculating the information flow during compilation, is to *a-priori* define boundaries in the program that define how much information may be leaked. This can be achieved by extending the type system to not only store the data type of a variable, but also a set of security policies that define how much information may flow into (or flow out of) a variable [VIS96]. The JIF language [Mye99a] is an example of certified compilation that incorporates a type system that supports security policies and tracks secure information flow throughout a whole program. JIF itself is an extension to the Java language.

The correct and secure behaviour of a program also depends—at least partially—on the environment in which the program is executed. Not all environments offer the same security guarantees. For example, a server that is locked in a room that is watched by armed guards is less likely to undergo physical manipulation, than a personal computer in an average home. Another approach to offer stronger guarantees regarding a program's execution is to use special hardware to mitigate manipulation attempts. One example is the *ARM TrustZone* [ARM] that physically splits the CPU in a „normal environment“ and a „trusted environment“, where the trusted environment offers stronger security guarantees.

In this thesis we assume that we have access to a normal environment and to some kind of trusted environment satisfying the following constraints: Both environments are capable of running a Java execution environment. The trusted environment offers higher guarantees in terms of averting data manipulation or control flow disturbance. For this thesis we do not specify *how* the trusted environment ensures these guarantees; we assume that it is possible to execute code in a secure fashion. Furthermore, these two environments are interconnected and capable of doing secure communication. Examples are given in section 2.9.

1.2. Problem Description

We claim that many programs are written for a single execution environment only, and have confidential or critical data interleaved with uncritical data. Attackers might take advantage of a certain program structure or covert channel stemming from mixing confidential with uncritical data in one program. Paying attention to the flow of information from variable to variable (*explicit* flow) and information leaked by conditionally changing the program flow (*implicit* flow) must be a number-one goal when designing a secure system [DD77]. Implicit flow may create a *covert channel* [Lam73], i.e. information not intended to be disclosed can be learnt by an attacker. Secure programs should be identified already at compile time; ideally, insecure programs are rejected by the compiler.

As discussed in section 1.1, not every execution environment offers the same security guarantees. Running a whole program in a weak environment may raise security concerns. Having the possibility to run parts of the program code in a trusted environment, the question arises: *Is it possible to only run uncritical parts of a program in the normal environment?*

Critical parts should run in a trusted environment and only specific information should be disclosed to the normal environment. In fact, such techniques are well established and common for client-server applications. The server is usually considered a trusted environment, the client is regarded as the normal environment. Unfortunately, the separation of a program into a client and server part must be done manually. This may be complicated and time-consuming. Since tracking of the information flow across program boundaries is hard when done manually, the resulting split program may also disclose too much information.

Foundations

If we have the possibility to track secure information flow in a single-environment-program and the possibility to define security policies inside this program, can this information be used to guide program partitioning? A popular language that incorporates these features and provides certified compilation is the JIF language [Mye99a]. The underlying *Decentralized Label Model* [ML00] defines a lattice of policies and principals; both are used to define reader and writer sets for variables in a program—this allows the programmer to explicitly state policies that define who may read and who may write information. Information in such a *security lattice* can naturally only become more confidential when manipulated [Den76; DD77; BL73; SM06; ML97]. If the property of *integrity* is also considered, data can naturally only lose integrity, because this property is dual to confidentiality [Bib77]. Detailed explanations are given in chapter 2.

Data that has a strict security policy but should be disclosed to a greater set of readers must undergo a special *declassification* mechanism [ZM01]. An example would be a ciphertext that has a strong security policy attached, because its content is influenced by a secret key. Despite having a strong policy, a ciphertext can be made public because it is computationally infeasible (given a reasonable cipher) for an attacker to reveal the plaintext. The strong confidentiality policy must therefore be replaced by a weaker policy to make the program usable. A declassification mechanism works against what is called *label creep*—the problem that after enough operations all variables end up with having the strongest policy [SM06]. This is due to the nature of the „one-way lattice“ described above.

Since data can only flow against the natural direction of a security lattice at some, well-defined places in the code (declassification points), is it possible to exploit this property and *automatically partition* a program into two or more pieces, such that critical parts can fully be moved to a trusted environment? In fact, this is possible as projects like *Swift* [Cho+07], *Jif/split* [Zda+02] and others show. Both projects are built on top of the JIF language; Swift focuses on web applications and uses the *Google Web Toolkit* to distribute programs across the web server and the browser. Jif/split focuses on distribution of programs to multiple hosts with different levels of trust. Both tools emit Java code when adding necessary runtime calls to a split program.

A detailed introduction to these concepts is given in chapter 2.

Terminology

Throughout this thesis we are using the following terms:

- *Partitioning*:
  1. The partitioning of a program describing what will be in the trusted environment partial program and what will be in the normal environment partial program.
  2. The process of separating the trusted environment partial program and normal environment partial program, given an unpartitioned input program
- *Original Program/Class/...*: the unpartitioned JIF program/class/...as it was written by a developer.
- *Trusted Environment Partial Program*: after the original program was partitioned, we use this term to refer to the parts of the program that will run in the trusted environment.
- *Normal Environment Partial Program*: after the original program was partitioned, we use this term to refer to the parts of the program that will run in the normal environment.

- *Split Program*: we use this term to refer to the combination of a trusted environment partial program and the accompanying normal environment partial program.

Goals for this Thesis

Hardware solutions like the ARM TrustZone are mostly targeted by low-level C programs. Assuming there is a Java runtime that runs in the TrustZone, can a high-level Java program be partitioned to run on a device with limited resources? The supporting framework should be lightweight and partitioning should focus on only two levels of trust.

The emission of Java code when weaving runtime calls into a split program may raise security concerns. The runtime is something that must be trusted by the user and is therefore part of the trusted computing base. A larger system implies a higher risk for the user, that there is a defect or an information leak. Some proof (or at least a guarantee) that the runtime is secure might be asked for.

This thesis aims for the following goals:

- *(Objective A)* **Automatic** *Bebop* takes as input a JIF program and produces as output two partial programs—one to be run in a trusted environment, one to be run in a normal environment—that cooperatively rebuild the functionality of the original program.

- *(Objective B)* **Secure** Code emitted by the newly developed compiler should not leak information via covert channels as defined by Denning. Ideally, the tool also considers security concerns for generated code, that is responsible for communication between two partial programs.

- *(Objective C)* **Minimal Trusted Computing Base** The compiler should strive for an optimal partitioning with respect to the amount of code placed in one execution environment, i.e. the trusted environment partial program should be as large as necessary, but as small as possible.

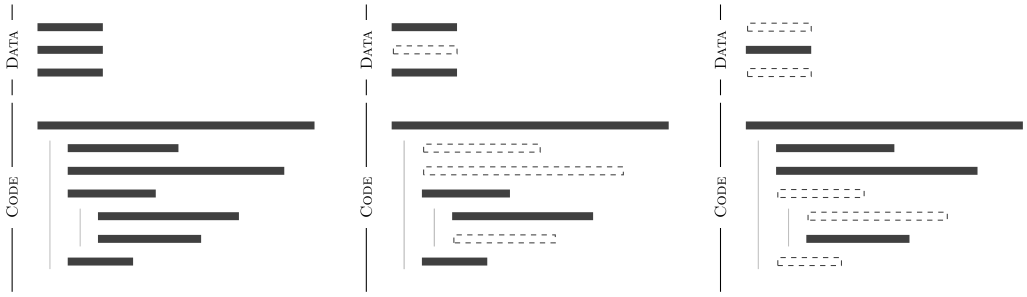
- *(Objective D.1)* **Confidentiality** Confidentiality policies defined by the user shall be respected when determining a program partitioning.

- *(Objective D.2)* **Integrity** Integrity policies defined by the user shall be respected when determining a program partitioning.

1.3. Contribution

Figure 1.2: The *Bebop* projects's logo.

Building on the JIF language and its policy system, we developed a compiler that automatically partitions a given program into two parts that will execute in environments



(a) Original class. (b) Trusted partial class. (c) Normal partial class.

Figure 1.3.: Classes consist of data and methods. During splitting, code and data is distributed between a trusted partial class and a normal partial class.

with different levels of trust. The project—named *Bebop*—consists of a compiler for a language extension of JIF and builds on the Polyglot compiler infrastructure [NCM03]. Additionally, we developed an accompanying runtime that hosts split programs.

The primary design principles of *Bebop* build on the assumption that the partitioning can be achieved with simple remote method invocation—systems like Jif/split employ a *continuation passing* approach [SS75b] and thus must rewrite the control flow of the complete program. We assumed that every method in a class can be assigned a definitive placement, that is all of its statements either „run in trusted environment“ or „run in normal environment“. Furthermore, we assumed that methods with indefinite placement—i.e. a method consists of both statements that should run in the trusted environment and statements that should run in the normal environment—can be partitioned into a set of smaller methods, all of which have definite placement.

For each class written by the user, both environments have a partial class that contains a subset of the methods and fields of the original class. As shown in Figure 1.3, the data (fields) and code (methods) of a class are distributed to two separate class files. The sets of fields and „implemented“ methods in the partial classes are disjoint. For each method that is implemented in the opposite environment, the compiler creates a method stub that allows to invoke a remote call.

Compilation Pipeline

Figure 1.4 depicts a typical compile cycle for source code that is to be partitioned. The system takes one or more source code files as input. The source language is a small extension to the JIF language as it has one built-in principal that is used when annotating policies. The JIF compiler is executed in a regular fashion, except the target code generation is disabled in the first step. The resulting JIF code—if the source code passes the JIF compilation, i.e. information flow in the program does not violate the specified policies—is passed to the *Bebop* Compiler whose functionality is described below. The *Bebop* Compiler again emits JIF code annotated with security policies to ensure the generated code does not leak information. In the next pass, the JIF compiler

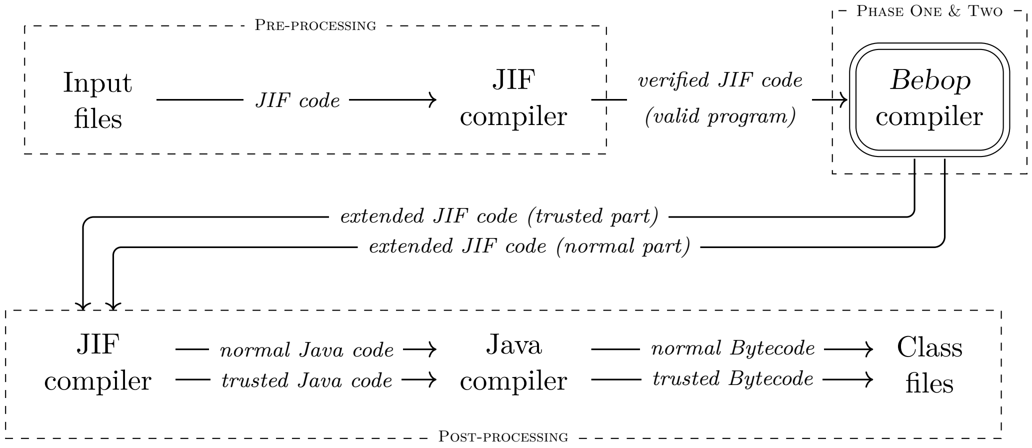


Figure 1.4.: Compile cycle.

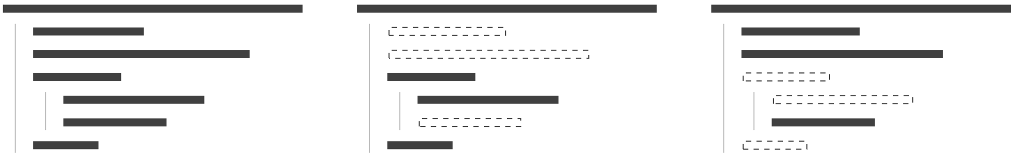
is executed again; this time with enabled target code generation. The resulting Java code is sent to the standard `javac` compiler and class files are created. The output consists of two directories, one containing target Java sources and class files for the normal environment partial program, the other contains the respective trusted environment files.

The *Bebop* Compiler

Taking a set of JIF source files as input, the *Bebop* Compiler first invokes the JIF compiler to ensure the information flow of the program adheres to the defined security policies. If this pass succeeds, the single class files are duplicated and renamed with a suffix of either `_N` for classes that will be on the normal environment, or `_T` for classes that will be on the trusted environment. This is done to have „physically“ different names for the new classes and therefore have a real separation. Also, keeping two different classes of the same name would have raised the need to rewrite parts of the underlying Polyglot compiler infrastructure [NCM03] to support this feature.

In the next pass, a definite placement—that is either Trusted Environment or Normal Environment or Both—is assigned to each statement and field. The decision is based on comparing the labels of each node in the Abstract Syntax Tree (AST) (cf. section 2.8 to a *threshold policy* that is defined in the compiler. If the policy of the node is less restrictive than (cf. section 2.6) the threshold policy, then the placement will be Normal Environment, otherwise the placement is Trusted Environment. This algorithm is executed for each sub-tree of the AST that makes up a single statement, or field respectively. Statement sub-trees with mixed placements are either split into multiple statements (introducing new temporary variables) or are bound to Trusted Environment if no sub-expressions can be extracted. A detailed description of the algorithm is given in subsection 4.5.8. After this pass, every statement has a definite placement and only exists in one environment.

The main idea of *Bebop* is to always start execution of a split program in a defined



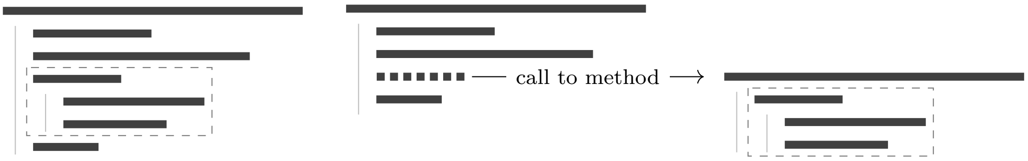
(a) Original. (b) Trusted environment. (c) Normal environment.

Figure 1.5.: Consecutive statements in a method may be affected by different policies and can end up having different placement. A method can therefore be seen as an interleaving of statement groups that execute in different environments.

entry point in the normal environment, and to invoke remote calls to trusted methods when needed. Optionally, the trusted environment can also call to the normal environment during the execution of a program; this is e.g. the case if secure information is *declassified* during execution of trusted statements and needs to be passed back to the normal environment. This requires that a method can be entirely placed in one environment, because single statements cannot exist without a surrounding method. Since a method may consist of many statements not necessarily sharing the same placement (see Figure 1.5) a useful definite placement for a method can not be easily derived. A trivial solution would be to move a method to the trusted environment if at least one trusted statement is found. The disadvantage is, that this may lead to a non-optimal partitioning where everything is on the trusted environment. Such a partition is not desired as normal environments may have limited resources.

A more optimal solution should partition a program with respect to how much code is moved to one execution environment. To construct an optimal solution, *Bebop* searches for groups of consecutive statements sharing the same placement in one method. Such a group is then moved to a new, synthetic, method and the original statements are replaced by a call to this new method. Figure 1.6 illustrates how statement groups are moved out of a method. The compiler makes sure that data dependencies are synchronised. If a group contains a statement that manipulates the control flow—for example a `continue` is moved out of its parenting `while` statement—code is emitted to synchronise the control flow change back to the appropriate call site. In subsection 4.5.8 we describe in detail, how the extraction of statement groups works and how it is made sure that data dependencies are not broken. After this pass, every method of a class has a definite placement and all statements in the method's body share this placement. New additional methods may be present in the class.

The *Bebop* Compiler is designed to emit a `_N` and `_T` target class for each input class. Those two classes cooperatively re-build the functionality present in the original class. Class fields are distributed across the two environments, with fields having the placement Trusted Environment only being present in the trusted environment and vice versa. Class methods are also distributed in terms of the statement making up their bodies. A method keeps its statements if the parenting class has the same placement as the method. A *method stub*, initiating a remote call to the other environment, is emitted if the class



(a) Original method.

(b) Statements with same placement moved out to a new method.

Figure 1.6.: Replacing statement groups by remote calls to synthetic methods allows assigning a definite placement to each method. Methods can then be distributed between the two environments and called with a remote calling mechanism.

placement does not match the method placement. The *Bebop* Runtime Environment will take care of executing the call, marshalling of the arguments and return value, and exceptions thrown in the other environment.

Having partitioned a class into a trusted and normal part, the two pieces must somehow be identified to make up a *pair*, and it must be ensured that a corresponding object in the opposite environment is always present. This is needed for remote method invocation, because it must be determined on which remote instance the method should be called on. During compilation, an additional instance field is introduced that holds a UUID to uniquely identify an instance. Additionally, code is emitted that intercepts object creation and makes sure that a remote object is always activated when a local object is constructed. The runtime makes sure that the remote object is assigned the same ID and keeps track of objects in a look-up table.

The *Bebop* Runtime Environment

After successful compilation of a program, both the trusted and normal parts have turned from a (typically) active to a reactive program. Since execution always starts in the normal environment, the trusted part only offers a set of remote methods that can be called. The normal part offers a special entry point that is invoked by the runtime. Both partial programs must therefore be *hosted* on a suitable container that enables communication between the two environments. Additionally, the container on the normal environment is responsible for bootstrapping the application. A graphical overview of the system is given in Figure 1.7.

Since one of the goals for the supporting runtime was to be scalable and extensible, the whole system works with an asynchronous message passing system. The only requirement (besides a working Java runtime environment) imposed by the *Bebop* Runtime Environment is, that the two runtime instances can be connected by a Java `InputStream`/ `OutputStream` pair. It does not matter if the underlying stream is a TCP connection (as in the reference implementation), a Pipe or a special JNI-driven stream that abstracts the low-level TrustZone TEE API. This allows us to use *Bebop* for many scenarios, e.g. in a network where a normal client is connected to a trusted server, or on special

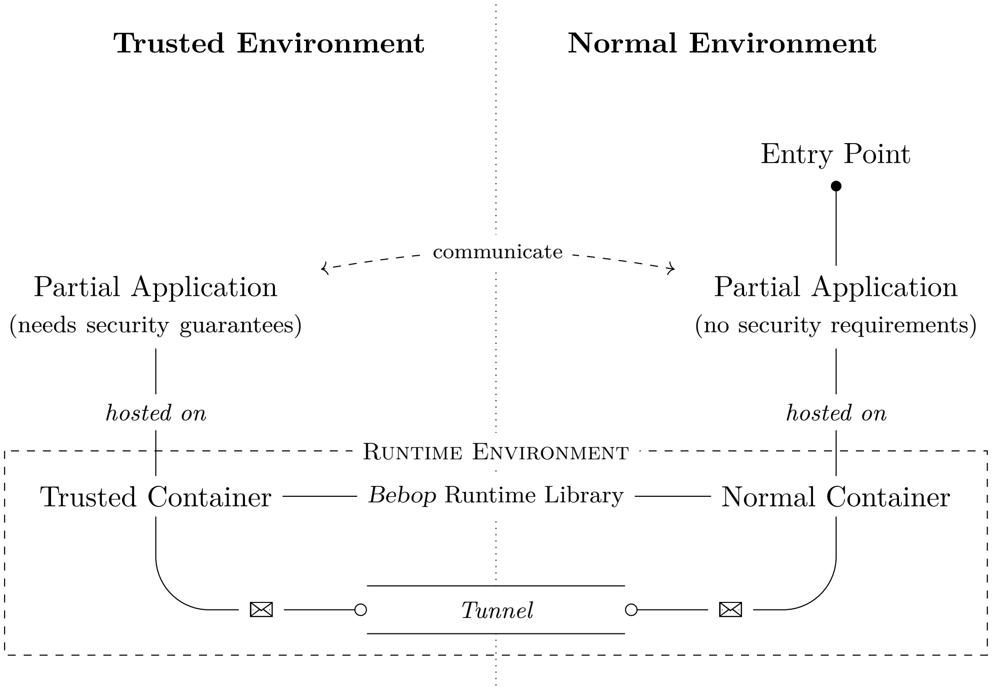


Figure 1.7.: *Bebop* Runtime Environment overview.

hardware like the ARM TrustZone.

The runtime also provides an API for remote object activation and remote method invocation as described in chapter 5. These methods are used by the compiler during the partitioning process when it emits code that connects two partial programs. Additional functionality includes helper methods for the compiler to allow easy (un)boxing of primitive objects, cast and serialisation utilities.

When a remote object is activated—i.e. an instance of an object must be constructed—the runtime is responsible for translating between normal/trusted class names, instantiation of the object, and execution of the appropriate constructor (if necessary). It also makes sure that the object's ID is correctly set. The newly created object, together with its unique ID, is stored in a look-up table. When methods on a remote object are invoked, the runtime looks up the instance and dispatches the call. It also handles marshalling of arguments and return values. On invocation of a remote method, the runtime also handles possible exceptions and transports it back to the call site.

Evaluation

We developed a compiler infrastructure that allows to partition a program into two cooperative parts that execute in different environments. We provide an accompanying runtime—as defined in *(Objective E)*—that hosts the partial programs and manages communication between the two parts. The runtime API is annotated with JIF policies and emitted glue-code is JIF compliant. Furthermore, the *Bebop* Runtime Environment is lightweight and does not depend on additional libraries or toolkits. Although initially targeted, support for integrity policies *(Objective D.2)* was neglected because this would exceed the scope of this thesis. Future work may fill this gap.

We evaluated the compiler on the basis of two examples: a key derivation example that employs a time-based one-time password algorithm, and a log-in example that reads a password from a trusted channel and compares the password in a trusted environment. The *Bebop* Compiler produced a partitioning *(Objective A)* that moved critical code like the password compare or the key derivation to a trusted environment while leaving uncritical code like printing results to the user on the normal environment *(Objective C)*. The example programs were annotated with confidentiality policies that were respected by the *Bebop* Compiler *(Objective D.1)*. Intermediate code (before post-processing) shows that calls to the *Bebop* Runtime Environment are annotated with security policies *(Objective B)*; the intermediate code is printed in Appendix A.

Evaluation of the *Bebop* Runtime Environment was done by means of message throughput and shows that the *Bebop* Runtime Environment is suited well to the kind of workloads that we think are to be expected for programs the *Bebop* Compiler is able to partition. Properties like extensibility were achieved by software design decisions: the use of message passing enables us to easily extend the *Bebop* Runtime Environment. The *Bebop* Runtime Environment shows good portability, has a small footprint, is extensible and scales well *(Objective F)*.

A detailed evaluation of the developed tools is given in chapter 6.

Limitations and Future Work

- Integrity properties are neglected.
- The *Bebop* Language does not support method calls, and `declassify` statements and expressions in constructors.
- JIF's polymorphic label mechanism (cf. section 2.6) is not supported.
- Method parameters used may be only of primitive types.
- Recursive methods are not supported.
- The `RemoteCallingContext` may not be synchronised correctly if an exception is encountered during execution of a remote method.
- The `RemoteCallingContext` may grow very large, if many parameters must be synchronised.
- The runtime's `ObjectStore` currently lacks a mechanism to clear out destroyed (or orphaned) objects.
- The control flow of a program is currently not secured against manipulation.
- There is currently no native TEE support due to lack of a Java port.

A detailed listing and explanation of all limitations is given in chapter 4 and chapter 5.

1.4. Structure of this Thesis

After this introduction, chapter 2 will give a presentation of the theoretical backgrounds of secure information flow and the *Decentralized Label Model*. Additionally, the policy model of JIF is described in detail and examples are given. Related work is discussed in chapter 3. In chapter 4, we give a detailed description of the implemented *Bebop* Compiler. It is shown how the compiler determines the placement of partitioned code, how it partitions classes, and what runtime glue-code is emitted. The *Bebop* Runtime Environment and the remote method invocation architecture is explained in chapter 5. Afterwards, an evaluation of the implemented system is done on basis of two sample programs; results are printed in chapter 6. The final chapter 7 draws conclusions based on the preceding chapters and gives an outlook on future work.

CHAPTER 2

PRELIMINARIES

2.1. Covert Channels

A computer program usually processes some kind of input information and transforms it to output information. By doing so, the program creates a channel, such that said information can flow out to be read by the user. If the flow of information is intended, the channel is called *legitimate* [Lam73]. On the other hand, the behaviour of the program may reveal information about its internals, not intended to be read by the user. Such *covert channels* [Lam73] may have severe impact on the security. There are many different types of covert channels, depending on on what an attacker is able to observe. Having physical access to a computation device allows for a wider range of attacks.

2.2. Information Flow

Looking at a sequence of statements in a computer program, one can find that information *flows* through the program. Information, for instance, enters a function through its parameters, is manipulated, and a value derived from this information is returned by the function. A flow occurs if at least *some*—not necessarily all—information is transferred. Information flow can either be determined statically—i.e. by looking at the program code—or dynamically—i.e. by executing the program and tracking memory access.

For static information flow, two kinds of information flow are distinguished [DD77]. For a statement $b = f(a)$ the flow is called *explicit*, because the **execution** of f happens in any case. For a statement $if(c)b = f(a)$ the flow is called *implicit*, because the **execution** of f depends on the value of a condition c. Since the value of c is in general not statically available, it cannot be a-priori decided if f will execute or not.

Implicit flow expresses the property that all values depending on the evaluation of a conditional expression *at runtime*, can reveal information about what the value of the condition was. In the example in Listing 2.2, the variable `granted` is only assigned constant values, although, information about the variable `password` can be learnt. Listing 2.1 showcases explicit flow. Intuitively, information flow is *explicitly* visible, because information stored in the variable `password` is transferred to `x`.

Listing (2.1) Explicit flow

```
password = "tiger";
x = password;
```

Listing (2.2) Implicit flow

```
granted = false;
if (password == "tiger")
    granted = true;
```

Figure 2.1.: Comparison of explicit and implicit information flow.

Implicit information flow can be a covert channel, if no information is intended to be transferred (cf. section 2.1).

2.3. The Non-Interference Property

Non-interference states that any high-confidential data processed by a program must not interfere with low confidentiality data [GM82; SM06]. This means that if a variable with low confidentiality policy (cf. section 2.4) is influenced by a high-confidential value, the contents of the low variable must not be displayed to a principal with only low clearance.

2.4. Security Policies

Policies are a basic concept, used to describe security requirements of a system. Goguen and Meseguer define a security policy as: *„The purpose of a security policy is to declare which information flows are not to be permitted. Giving such a security policy can be reduced to giving a set of noninterference assertions.“* [GM82]. Policies therefore control what changes to the state of a system can be observed by a specific user.

2.5. Secure Information Flow

Secure information flow as defined by Denning [Den76] means that no unauthorized flow of information is possible. If a variable with low security policies depends on a high security condition, an attacker may learn confidential information. This is due to implicit information flow, cf. section 2.2.

A program is considered *secure* if no unintended information flows occur (*information flow control*) and if the program can be certified statically by a compiler [DD77].

Example

Considering a system with a highly confidential variable `account_balance`. When checking if a customer is eligible for a credit, information about the balance implicitly flows to the low security variable `creditworthiness`. A clerk with only low clearance learns information about a customer's financial status.

```
if (account_balance > 100000) {
  creditworthiness = "good";
} else if (account_balance > 50000) {
  creditworthiness = "average";
} else {
  creditworthiness = "bad";
}
```

2.6. JIF

This section gives an introduction to the security model of the JIF language. The JIF language [Mye99b; Mye99a] is an extension to the Java language and aims provide mechanisms to certify programs in terms of secure information flow. It is based on the *Decentralized Label Model* developed by Myers and Liskov [ML97; ML98; ML00]. JIF derives from approximately Java 1.4. All concurrency related features of the Java language (threading) are disabled.

Policies for confidentiality and integrity are defined and ordered in terms of restrictiveness. Labels as a compound type containing confidentiality and integrity policies are introduced. All entities—that are principals, policies and labels—can be placed on security lattices that allow easy comparison and combination of those entities.

2.6.1. Principals

In JIF, a principal is defined as: *„an entity with some power to observe and change certain aspects of the system“* [Cho+09]. It is JIF's primary mechanism to express users, groups, processes, etc. that are present in a system. Principals can delegate their power to other principals via the *acts-for relation.* Thus, principals in a system form a hierarchy and can be ordered in terms of authoritative power.

2.6.2. The acts-for Relation

A principal `a` is said to *act for* principal `b`, `a` $\succeq$ `b`, if `b` empowers `a` to act on their behalf [ML00]. If `a` $\succeq$ `b`, `a`'s set of privileges is extended by the set of privileges of `b`. The acts-for relation either delegates *all* privileges, or none at all.

Definition 1 (Poset) *A* partial ordering *is a binary relation* $\sqsubseteq : L \times L \to \{\text{true}, \text{false}\}$. *A* partially ordered set (poset) *is a set* L *with a partial ordering* $\sqsubseteq$*,* $(L, \sqsubseteq)$*. [NNH99]*

The acts-for relation fulfils the following properties:

- The relation is reflexive: $\mathtt{a} \succeq \mathtt{a}$.
- The relation is antisymmertic: if $\mathtt{a} \succeq \mathtt{b}$ and $\mathtt{b} \succeq \mathtt{a}$ it follows that $\mathtt{a} = \mathtt{b}$, and
- The relation is transitive: if $\mathtt{a} \succeq \mathtt{b}$ and $\mathtt{b} \succeq \mathtt{c}$ it follows that $\mathtt{a} \succeq \mathtt{c}$.

The acts-for relation $\succeq$ in combination with a set of principals $\mathtt{P}$ induces a partial order in terms of authoritative power and $\mathtt{P}$ is therefore a *partially ordered set*, $(\mathtt{P}, \succeq)$.

Note: this definition of the acts-for relation excludes cyclic acts-for relations between principals, that is $\mathtt{a} \succeq \mathtt{b} \land \mathtt{b} \succeq \mathtt{a}$ with $b \neq a$, as this violates the antisymmetric property of the partial order. Although, it is possible to allow cycles of this kind—the definition of the relation must then be adapted to work on equivalence classes of principals [Mye99b].

Definition 2 (Bounds) *A subset $K \subseteq L$ has an* upper bound $u \in L$ *if* $\forall k \in K\colon k \sqsubseteq u$. *A subset $K \subseteq L$ has a* lower bound $l \in L$ *if* $\forall k \in K\colon l \sqsubseteq k$. *Let U be a set of upper bounds for K and let L be a set of lower bounds for K. The* least upper bound (LUB, join, $\sqcup$)*, if it exists, is defined as $u_0 \in U\colon \forall u \in U\colon u_0 \sqsubseteq u$. The* greatest lower bound (GLB, meet, $\sqcap$)*, if it exists, is defined as $l_0 \in L\colon \forall l \in L\colon l \sqsubseteq l_0$. [NNH99]*

The acts-for relation can be used to model groups of principals with different authorisations. Figure 2.2 gives an example of a hierarchical principal structure modelled with the acts-for relation. In a principal hierarchy, there is a top principal $\top$ that acts for any principal, $\forall \mathtt{p} \in \mathtt{P}\colon \top \succeq \mathtt{p}$. Likewise there is a bottom principal $\bot$ that any principal can act for, $\forall \mathtt{p} \in \mathtt{P}\colon \mathtt{p} \succeq \bot$ [ML00].

For any subset of principals $\mathtt{S} \subseteq \mathtt{P}$ a principal $\mathtt{u} \in \mathtt{P}$ is an *upper bound* of $\mathtt{S}$ if $\forall \mathtt{s} \in \mathtt{S}\colon \mathtt{u} \succeq \mathtt{s}$. Such an upper bound always exists because of the top principal $\top$. Analogous, there is a *lower bound* $\mathtt{l} \in \mathtt{P}$ for all $\mathtt{s} \in \mathtt{S}$ such that $\mathtt{s} \succeq \mathtt{l}$. This is always satisfied by the bottom principal $\bot$. The least upper bound for a set $\mathtt{G} \subseteq \mathtt{P}$ is denoted by the *join* operator: $\sqcup \mathtt{G} = \text{lub}(\mathtt{G})$. The greatest lower bound is denoted by the *meet* operator: $\sqcap \mathtt{G} = \text{glb}(\mathtt{G})$.

Definition 3 (Lattice) *A partially ordered set $(L, \sqsubseteq)$ where all subsets have a least upper bound and a greatest lower bound is called a* (complete) lattice $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. *The* greatest element *is $\top = \sqcup L$ and the* least element *is $\bot = \sqcap L$. [NNH99]*

The least element $\bot$ is defined by $\bot = \sqcap \mathtt{P} = \text{glb}(\mathtt{P})$. Using the definition of the GLB, $\forall \mathtt{p} \in \mathtt{P}\colon \mathtt{p} \succeq \bot$. Similarly, the top element $\top = \sqcup \mathtt{P} = \text{lub}(\mathtt{P})$ that is: $\forall \mathtt{p} \in \mathtt{P}\colon \top \succeq \mathtt{p}$. This is consistent with the definition of the bottom and top principal given above.

Examples

1. In Figure 2.2 the CEO `carl` can act for all `executives`, $\mathtt{carl} \succeq \mathtt{executives}$, and can therefore also act for the group of all employees, $\mathtt{carl} \succeq \mathtt{employees}$. In terms

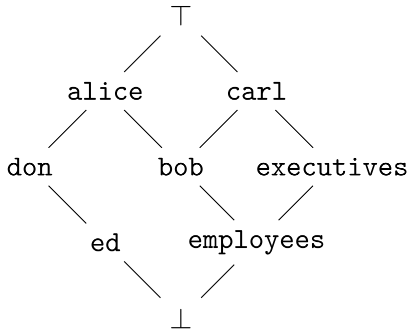


Figure 2.2: A model of a privilege hierarchy with the acts-for relation ($\succeq$). In this example, $\mathtt{carl} \succeq \mathtt{bob}$, and $\mathtt{don} \succeq \mathtt{ed}$, and so forth.

of access management with user groups, every principal is a group and a singleton at the same time. Principal `carl`'s privileges are extended by all privileges of `executives` and all privileges of `employees`.

2. The least upper bound for the principals `employees` and `executives` in Figure 2.2 is given by $\text{lub}(\{\mathtt{employees}, \mathtt{executives}\}) = \mathtt{employees} \sqcup \mathtt{executives}$. Using the definition of the $\sqcup$ operator we get $\forall \mathtt{p} \in \mathtt{P}\colon \mathtt{u} \succeq \mathtt{p}$ with $\mathtt{u}$ being the smallest upper bound. Let $\mathtt{U}$ be the set of upper bounds for `employees` and `executives`, $\mathtt{U} = \{\mathtt{p}\colon \mathtt{p} \succeq \mathtt{employees} \land \mathtt{p} \succeq \mathtt{executives}\} = \{\mathtt{executives}, \mathtt{carl}, \top\}$. Looking at the ordering $\top \succeq \mathtt{carl} \succeq \mathtt{executives}$, the least element is given by the principal `executives`.

3. The greatest lower bound for `bob` and `don` is given by the greatest element that both `bob` and `don` can act for: $\mathtt{bob} \sqcap \mathtt{don} = \bot$.

2.6.3. Confidentiality Policies

Concerns over confidentiality are expressed in terms of *reader policies*. The notation $\langle \mathtt{owner} \rightarrow \mathtt{readers} \rangle$ states that the principal `owner` grants all principals in the set `readers` the right to read certain values. The policy $\mathsf{pol} = \langle \cdot \rightarrow \cdot \rangle$ is called a *confidentiality policy*, the set of all defined confidentiality policies is denoted by C. It is convenient to construct a set of reading principals for a given policy:

Definition 4 (Reader set) *Let* $(\mathtt{P}, \succeq)$ *be a partially ordered set of principals, let* $\mathtt{o} \in \mathtt{P}$ *be the owner of a read policy, let* $\mathtt{R} \subseteq \mathtt{P}$ *be the set of principals entitled by* $\mathtt{o}$ *to read a value,* $\langle \mathtt{o} \rightarrow \mathtt{R} \rangle \in \mathsf{C}$. *Furthermore let* $\mathtt{r}$ *be a member of* $\mathtt{R}$, *and let* $\mathcal{P}(\mathtt{P})$ *be the power set of* $\mathtt{P}$. *The function* $\text{readers}\colon \mathtt{P} \times \mathsf{C} \rightarrow \mathcal{P}(\mathtt{P})$*—if evaluated by a principal* $\mathtt{p}$*—yields a set of all principals that are allowed to read a value under a certain policy.* $\text{readers}(\mathtt{p}, \langle \mathtt{o} \rightarrow \mathtt{R} \rangle) = \{\mathtt{x} \in \mathtt{P}\colon \text{if } \mathtt{o} \succeq \mathtt{p} \text{ then } (\mathtt{x} \succeq \mathtt{o} \text{ or } \exists r \in \mathtt{R}\colon \mathtt{x} \succeq \mathtt{r}) \text{ else } (\text{true})\}$.

A principal `p` evaluating a confidentiality policy adheres to the following rules:

1. If the owner `o` of the policy does not act for `p`, the principal `p` ignores the policy and access to the value is granted to all principals, $\mathtt{o} \not\succeq \mathtt{p} \Rightarrow \text{readers}(\mathtt{p}, \mathsf{pol}) = \mathtt{P}$. A

Table 2.1.: Various reader sets of policies owned by $\top$. The principal hierarchy is taken from Figure 2.2. Since $\forall \mathtt{p} \in \mathtt{P}\colon \top \succeq \mathtt{p}$, the policy is respected by every principal and the reader sets are thus equal for all principals.

	Policy	readers(p, policy)
1	$\langle \top \to \bot \rangle$	$\mathtt{P}$
2	$\langle \top \to \mathtt{ed} \rangle$	$\{\mathtt{ed}, \mathtt{don}, \mathtt{alice}, \top\}$
3	$\langle \top \to \mathtt{don} \rangle$	$\{\mathtt{don}, \mathtt{alice}, \top\}$
4	$\langle \top \to \mathtt{alice} \rangle$	$\{\mathtt{alice}, \top\}$
5	$\langle \top \to \mathtt{bob} \rangle$	$\{\mathtt{bob}, \mathtt{alice}, \mathtt{carl}, \top\}$
6	$\langle \top \to \mathtt{carl} \rangle$	$\{\mathtt{carl}, \top\}$
7	$\langle \top \to \mathtt{employees} \rangle$	$\{\mathtt{employees}, \mathtt{executives}, \mathtt{bob}, \mathtt{alice}, \mathtt{carl}, \top\}$
8	$\langle \top \to \mathtt{executives} \rangle$	$\{\mathtt{executives}, \mathtt{carl}, \top\}$
9	$\langle \top \to \top \rangle$	$\{\top\}$

policy defined by $\mathtt{o}$ is only valid for principals „below $\mathtt{o}$“ in the principal hierarchy. Another principal $\mathtt{q}$ that $\mathtt{o}$ does not act for, may specify different confidentiality concerns (or none at all).

2. If $\mathtt{o} \succeq \mathtt{p}$ the reader set consists of all principals that can act for the owner of the policy, or for one of the specified readers, $\mathtt{o} \succeq \mathtt{p} \Rightarrow \text{readers}(\mathsf{p}, \mathsf{pol}) = \{\mathtt{x} \in \mathtt{P}\colon \mathtt{x} \succeq \mathtt{o} \text{ or } \mathtt{x} \succeq \mathtt{r}\}$.

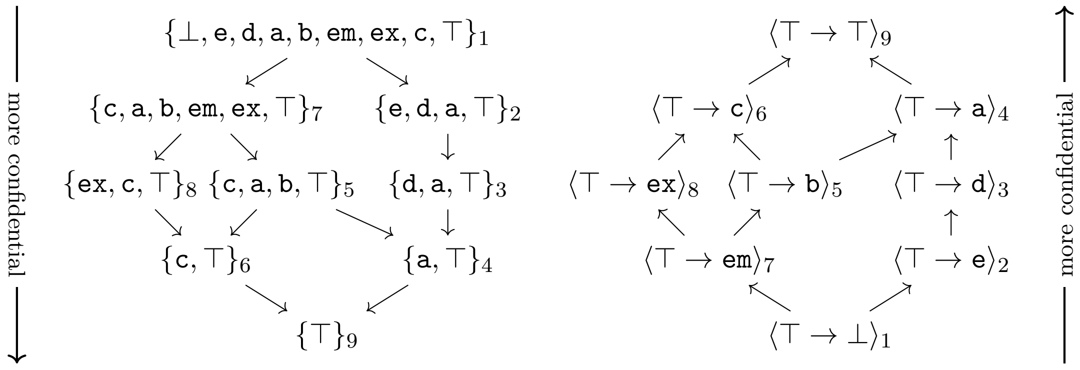
Confidentiality policies can be compared in terms of being „at most as restrictive as“ [ML00]. A confidentiality policy p_1 is less restrictive than policy p_2, if p_1 allows more readers than p_2. An example for reader sets is given in Table 2.1.

Definition 5 ($\sqsubseteq_C$) *Let p_1 and p_2 be confidentiality policies.* $\mathsf{p}_1 \sqsubseteq_C \mathsf{p}_2 \iff \forall \mathtt{p} \in \mathtt{P}\colon \text{readers}(\mathsf{p}, \mathsf{p}_1) \supseteq \text{readers}(\mathsf{p}, \mathsf{p}_2)$. *[ML00]*

The policy p_1 allows more readers—when evaluated for each p—than p_2 and is thus less confidential than p_2. An example of reader sets and the corresponding policy lattice is given in Figure 2.3.

Definition 6 (Preorder) *A* preorder *is a binary relation* $\sqsubseteq\colon L \times L \to \{\text{true}, \text{false}\}$ *that is reflexive and transitive. [Ok10]*

The relation $\sqsubseteq_C$ is reflexive, that is $\mathsf{a} \sqsubseteq_C \mathsf{a}$, and transitive, from $\mathsf{a} \sqsubseteq_C \mathsf{b} \wedge \mathsf{b} \sqsubseteq_C \mathsf{c}$ follows $\mathsf{a} \sqsubseteq_C \mathsf{c}$. $\sqsubseteq_C$ is hence a preorder. It is not a partial order, because the anti-symmetric property does not hold: Let $\mathsf{a} = \langle \mathtt{bob} \to \mathtt{alice} \rangle$ and let $\mathsf{b} = \langle \mathtt{bob} \to \mathtt{bob} \rangle$. Enumerating the reader sets for all principals (see Table 2.2) it holds that $\mathsf{a} \sqsubseteq_C \mathsf{b}$ and $\mathsf{b} \sqsubseteq_C \mathsf{a}$ because the reader sets are equal for all principals. However it does not follow from this that $\mathsf{a} = \mathsf{b}$, because the policies a and b are two different objects. By defining the equivalence $\mathsf{a} \sim \mathsf{b} \iff \mathsf{a} \sqsubseteq_C \mathsf{b} \wedge \mathsf{b} \sqsubseteq_C \mathsf{a}$, a partial order over the equivalence classes can be obtained.


more confidential
$\{\bot, \mathtt{e}, \mathtt{d}, \mathtt{a}, \mathtt{b}, \mathtt{em}, \mathtt{ex}, \mathtt{c}, \top\}_1$
$\{\mathtt{c}, \mathtt{a}, \mathtt{b}, \mathtt{em}, \mathtt{ex}, \top\}_7$
$\{\mathtt{e}, \mathtt{d}, \mathtt{a}, \top\}_2$
$\{\mathtt{ex}, \mathtt{c}, \top\}_8$
$\{\mathtt{c}, \mathtt{a}, \mathtt{b}, \top\}_5$
$\{\mathtt{d}, \mathtt{a}, \top\}_3$
$\{\mathtt{c}, \top\}_6$
$\{\mathtt{a}, \top\}_4$
$\{\top\}_9$
$\langle \top \to \top \rangle_9$
$\langle \top \to \mathtt{c} \rangle_6$
$\langle \top \to \mathtt{a} \rangle_4$
$\langle \top \to \mathtt{ex} \rangle_8$
$\langle \top \to \mathtt{b} \rangle_5$
$\langle \top \to \mathtt{d} \rangle_3$
$\langle \top \to \mathtt{em} \rangle_7$
$\langle \top \to \mathtt{e} \rangle_2$
$\langle \top \to \bot \rangle_1$
more confidential

(a) Reader set lattice. **(b)** Policy lattice.

Figure 2.3.: Lattice of reader sets ordered by $\supseteq$ and the corresponding policy lattice under the order $\sqsubseteq_{\mathsf{C}}$, for policies defined in Table 2.1.

Table 2.2.: Different policies with equivalent reader sets for all principals.

$\langle \mathtt{bob} \to \mathtt{alice} \rangle$		$\langle \mathtt{bob} \to \mathtt{bob} \rangle$		$\langle \mathtt{bob} \to \mathtt{carl} \rangle$	
$\top$	P	$\top$	P	$\top$	P
alice	P	alice	P	alice	P
carl	P	carl	P	carl	P
don	P	don	P	don	P
bob	$\{\top, \mathtt{alice}, \mathtt{carl}, \mathtt{bob}\}$	bob	$\{\top, \mathtt{alice}, \mathtt{carl}, \mathtt{bob}\}$	bob	$\{\top, \mathtt{alice}, \mathtt{carl}, \mathtt{bob}\}$
ed	P	ed	P	ed	P
$\bot$	$\{\top, \mathtt{alice}, \mathtt{carl}, \mathtt{bob}\}$	bob	$\{\top, \mathtt{alice}, \mathtt{carl}, \mathtt{bob}\}$	bob	$\{\top, \mathtt{alice}, \mathtt{carl}, \mathtt{bob}\}$
emps	$\{\top, \mathtt{alice}, \mathtt{carl}, \mathtt{bob}\}$	bob	$\{\top, \mathtt{alice}, \mathtt{carl}, \mathtt{bob}\}$	bob	$\{\top, \mathtt{alice}, \mathtt{carl}, \mathtt{bob}\}$
execs	P	execs	P	execs	P

The equivalent policies can be seen as being equal under the relation $\sim$, $[\mathsf{a}]_\sim = [\mathsf{b}]_\sim$, and a partially ordered set of confidentiality policies can be formed, $(\mathsf{C}/\!\sim, \sqsubseteq_{\mathsf{C}})$. Policies are therefore equal, if the reader sets are equal for all principals. An example of a policy lattice over $(\mathsf{C}/\!\sim)$ is given in Figure 2.4.

It holds for all policies p, that p is not more restrictive than the strictest policy $\langle \top \to \top \rangle$ and that the weakest policy $\langle \bot \to \bot \rangle$ is not more restrictive than p: $\langle \bot \to \bot \rangle \sqsubseteq_{\mathsf{C}} \mathsf{p} \sqsubseteq_{\mathsf{C}} \langle \top \to \top \rangle$.

Conjunction and Disjunction

Definition 7 **($\sqcup$, $\sqcap$)** *Let* p_1 *and* p_2 *be confidentiality policies.* $\mathsf{p}_1 \sqcup \mathsf{p}_2 = \mathrm{readers}(\mathrm{p}, \mathsf{p}_1) \cap \mathrm{readers}(\mathrm{p}, \mathsf{p}_2)$ *and* $\mathsf{p}_1 \sqcap \mathsf{p}_2 = \mathrm{readers}(\mathrm{p}, \mathsf{p}_1) \cup \mathrm{readers}(\mathrm{p}, \mathsf{p}_2)$

Having a strictest and weakest reader policy, there is always an upper and lower bound for any subset of policies $\mathsf{S} \subseteq \mathsf{C}/\!\sim$. The least upper bound $\sqcup$ of two confidentiality policies p_1 and p_2 is the *conjunction* of confidentiality policies, $\mathsf{p}_3 = \mathsf{p}_1 \sqcup \mathsf{p}_2$. The policy

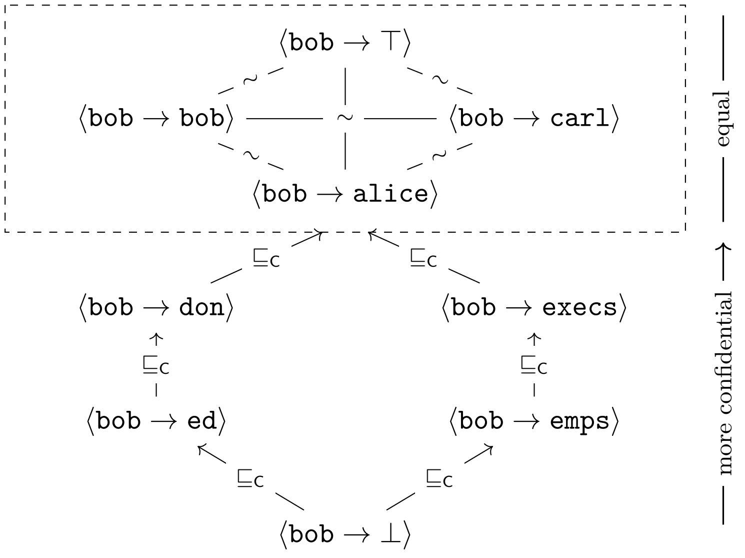


Figure 2.4: A lattice of confidentiality policies. The topmost diamond can be seen as single lattice element since those policies form an equivalence class.

p_3 allows only readers that are defined in both p_1 and p_2. The reader set of p_3 is the greatest lower bound of the reader sets of p_1 and p_2 in the lattice of all possible reader combinations, that is the power set of $\mathtt{P}$ ordered by subset inclusion, $(\mathcal{P}(\mathtt{P}), \subseteq)$. Since the policy lattice and the reader set lattice are duals, there is a dual element in $(\mathsf{C}/{\sim}, \sqsubseteq_{\mathsf{C}})$ with $p_3 = [p_1 \sqcup p_2]_{\sim} = \text{readers}(p, p_1) \sqcap \text{readers}(p, p_2) = \text{readers}(p, p_1) \cap \text{readers}(p, p_2)$.

The greatest lower bound operator $\sqcap$ expresses *disjunction* of confidentiality policies. A policy $p_4 = p_1 \sqcap p_2$ allows all readers either defined in the reader set of p_1 or the reader set of p_2. This corresponds to the least upper bound in the reader set lattice $(\mathcal{P}(\mathtt{P}), \subseteq)$, hence: $p_4 = [p_1 \sqcap p_2]_{\sim} = \text{readers}(p, p_1) \sqcup \text{readers}(p, p_2) = \text{readers}(p, p_1) \cup \text{readers}(p, p_2)$.

The partially ordered set of confidentiality policies $(\mathsf{C}/{\sim}, \sqsubseteq_{\mathsf{C}})$, the conjunctive and disjunctive operators, and the weakest and strictest policy, form a lattice $(\mathsf{C}/{\sim}, \sqsubseteq_{\mathsf{C}}, \sqcap, \sqcup, \langle \bot \rightarrow \bot \rangle, \langle \top \rightarrow \top \rangle)$.

Examples

1. Comparing the policies $p_1 = \langle \mathtt{executives} \rightarrow \mathtt{bob} \rangle$ and $p_2 = \langle \top \rightarrow \mathtt{carl} \rangle$, it holds that for each principal $p \in \mathtt{P}$ the reader sets of p_1 (see Table 2.3) are larger ($\supseteq$) than the reader sets of p_2 (see Table 2.1). The policy p_1 is in every case not more restrictive than p_2 and relation $p_1 \sqsubseteq_{\mathsf{C}} p_2$ is therefore valid. Having the policy $p_3 = \langle \top \rightarrow \mathtt{employees} \rangle$, the reader sets of p_1 are not in every case larger than the reader sets of p_3, thus $p_1 \not\sqsubseteq_{\mathsf{C}} p_3$.

2. Let $p_1 = \langle \top \rightarrow \mathtt{executives} \rangle$ and $p_2 = \langle \top \rightarrow \mathtt{alice} \rangle$ be reader policies in the lattice given in Figure 2.3. The conjunction $p_1 \sqcup p_2$ is given by the intersection of reader sets, $\text{readers}(p, p_1) \cap \text{readers}(p, p_2)$ which is the equivalence class $[\langle \top \rightarrow \top \rangle]_{\sim}$. The disjunction is given by $p_1 \sqcap p_2 = \text{readers}(p, p_1) \cup \text{readers}(p, p_2) = [\langle \top \rightarrow \mathtt{executives} \rangle \sqcap \langle \top \rightarrow \mathtt{alice} \rangle]_{\sim}$. As defined above, $[\cdot]_{\sim}$ denotes the equival-

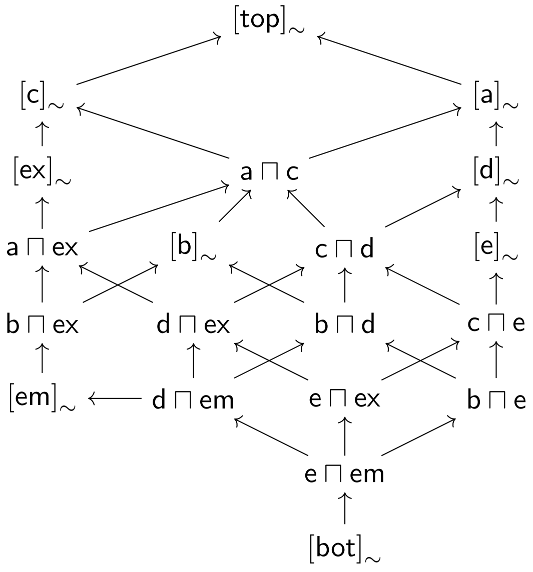


Figure 2.5: The lattice of Figure 2.3 extended by $\bigsqcup_{\mathsf{i,j} \in C/\sim}\{\mathsf{i},\mathsf{j}\}$ and $\bigsqcap_{\mathsf{i,j} \in C/\sim}\{\mathsf{i},\mathsf{j}\}$. For legibility, the policies are assigned names: $\mathsf{a} = \langle \top \to \texttt{alice} \rangle$, $\mathsf{b} = \langle \top \to \texttt{bob} \rangle$, $\mathsf{top} = \langle \top \to \top \rangle$, and so on.

Table 2.3.: Reader sets for the policy $\langle \texttt{executives} \to \texttt{bob} \rangle$.

Principal $\mathtt{p} \in \mathtt{P}$	Respects Policy ($\mathtt{o} \succeq \mathtt{p}$)	readers($\mathtt{p}$, policy)
`alice`	no	$\mathtt{P}$
`bob`	no	$\mathtt{P}$
`carl`	no	$\mathtt{P}$
`don`	no	$\mathtt{P}$
`ed`	no	$\mathtt{P}$
`employees`	yes	$\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{ex}, \top\}$
`executives`	yes	$\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{ex}, \top\}$
$\top$	no	$\mathtt{P}$
$\bot$	yes	$\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{ex}, \top\}$

ence class of all policies of the same restrictiveness. Figure 2.5 shows the ordering of policies in Figure 2.3 extended by mutual conjunction and disjunction of the policies.

2.6.4. Integrity Policies

Principals writing to variables may change the trustworthiness—or integrity—of the value. Concerns over writers are expressed in *integrity policies.* The mapping $\langle \texttt{owner} \leftarrow \texttt{writers} \rangle$ states that `owner` grants all principals in the set `writers` to influence the value of a variable.

Definition 8 (Writer Set) *Let* $\mathtt{o}$ *be the owner of a write policy, let* $\mathtt{W}$ *be the set of principals entitled by* $\mathtt{o}$ *to write a value,* $\langle \mathtt{o} \leftarrow \mathtt{W} \rangle \in \mathsf{I}$*, where* I *is the set of integrity policies.*

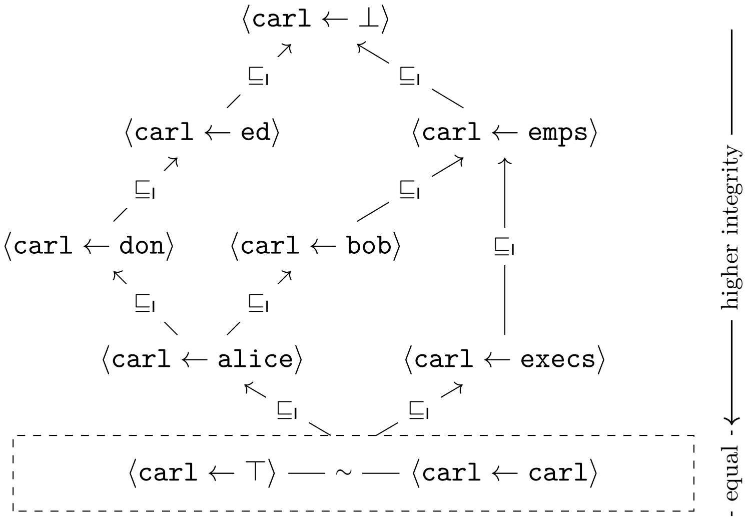


Figure 2.6: A lattice of integrity policies in $\mathsf{I}/{\sim}$ ordered by $\sqsubseteq_{\mathsf{I}}$.

Furthermore, let $\mathtt{w}$ *be a member of the writer set* $\mathtt{W}$*. The function* $\text{writers}\colon \mathtt{P} \times \mathsf{I} \to \mathcal{P}(\mathtt{P})$ *yields a set of all principals that are allowed to change a value under a certain policy.* $\text{writers}(\mathsf{p}, \langle \mathtt{o} \leftarrow \mathtt{W}\rangle) = \{\mathtt{x} \in \mathtt{P}\colon \text{if } \mathtt{o} \succeq \mathtt{p} \text{ then } (\mathtt{x} \succeq \mathtt{o} \text{ or } \exists \mathtt{w}\colon \mathtt{x} \succeq \mathtt{w}) \text{ else } (\text{true})\}$.

Similar to confidentiality policies, integrity policies are only respected by a principal $\mathtt{p}$ evaluating the writer set, iff the owner of the policy is higher up or equal in the hierarchy than $\mathtt{p}$, in terms of the acts-for relation. Integrity policies can be ordered by the reflexive and transitive relation $\sqsubseteq_{\mathsf{I}}$. The ordering is done in terms of „is more trustworthy than“. For higher integrity information fewer restrictions apply when being used in computation, hence this ordering can also be seen as „not more restrictive than“.

Definition 9 ($\sqsubseteq_{\mathsf{I}}$) *Let* p_1 *and* p_2 *be integrity policies.* $\mathsf{p}_1 \sqsubseteq_{\mathsf{I}} \mathsf{p}_2 \iff \forall \mathtt{p} \in \mathtt{P}\colon \text{writers}(\mathtt{p}, \mathsf{p}_1) \subseteq \text{writers}(\mathtt{p}, \mathsf{p}_2)$. *[ML00]*

The weakest—in terms of restrictiveness—integrity policy $\langle \top \leftarrow \top\rangle$ forms a lower bound on the set of integrity policies I. Restrictiveness for use in computation, and integrity of a value, are dual properties. $\langle \top \leftarrow \top\rangle$ is therefore the strictest policy in terms of write-access to the value, because it limits write-access to only $\{\top\}$. Analogous, the most restrictive integrity policy in terms of „restrictions when using the value“, $\langle \bot \leftarrow \bot\rangle$, allows every principal to change a value; more restrictions will apply when the value is used in computation. The weakest and strictest integrity policies form a lower and upper bound on the set of integrity policies I and it holds for all policies $\mathsf{p} \in \mathsf{I}$: $\langle \top \leftarrow \top\rangle \sqsubseteq_{\mathsf{I}} \mathsf{p} \sqsubseteq_{\mathsf{I}} \langle \bot \leftarrow \bot\rangle$. As with confidentiality policies, the preorder $\sqsubseteq_{\mathsf{I}}$ can be turned into a partial order by defining integrity policies to be equivalent iff their writer sets are equivalent for all principals. $\mathsf{a} \sim \mathsf{b} \iff \forall \mathtt{p} \in \mathtt{P}\colon \text{writers}(\mathtt{p}, \mathsf{a}) = \text{writers}(\mathtt{p}, \mathsf{b})$. Integrity policies are thus defined over the partial order $(\mathsf{I}/{\sim}, \sqsubseteq_{\mathsf{I}})$. Figure 2.6 gives an example of equivalence classes of integrity policies, ordered by $\sqsubseteq_{\mathsf{I}}$.

Conjunction and Disjunction

Definition 10 **($\sqcup$, $\sqcap$)** *Let* p_1 *and* p_2 *be integrity policies.* $\mathsf{p}_1 \sqcap \mathsf{p}_2 \iff \forall \mathtt{p} \in \mathtt{P}\colon \text{writers}(\mathtt{p}, \mathsf{p}_1) \cap \text{writers}(\mathtt{p}, \mathsf{p}_2)$ *and* $\mathsf{p}_1 \sqcup \mathsf{p}_2 \iff \forall \mathtt{p} \in \mathtt{P}\colon \text{writers}(\mathtt{p}, \mathsf{p}_1) \cup \text{writers}(\mathtt{p}, \mathsf{p}_2)$.

The $\sqcap$ operator denotes *conjunction* of integrity policies, $\mathsf{p}_3 = \mathsf{p}_1 \sqcap \mathsf{p}_2$. Only writers defined in both p_1 and p_2 are allowed to influence a value under the policy p_3. This corresponds to the greatest lower bound in the dual lattice of all possible writer sets, $(\mathcal{P}(\mathtt{P}), \subseteq)$. The *disjunction* of integrity policies is given by $\mathsf{p}_4 = \mathsf{p}_1 \sqcup \mathsf{p}_2$; a principal allowed to influence a value must be either allowed by p_1 or by p_2. This is equivalent to the least upper bound in $(\mathcal{P}(\mathtt{P}), \subseteq)$.

Examples

1. Let $\mathsf{p}_1 = \langle \mathtt{carl} \leftarrow \mathtt{ed} \rangle$ and let $\mathsf{p}_2 = \langle \mathtt{carl} \leftarrow \mathtt{carl} \rangle$. The writer sets of p_2 are in every case—that is when evaluated for each principal—smaller ($\subseteq$) than the writer sets of p_1, therefore $\mathsf{p}_2 \sqsubseteq_\mathsf{I} \mathsf{p}_1$. The policy $\mathsf{p}_3 = \langle \mathtt{carl} \leftarrow \top \rangle$ has the same writer sets as p_2, thus $\mathsf{p}_2 \sqsubseteq_\mathsf{I} \mathsf{p}_3 \wedge \mathsf{p}_3 \sqsubseteq_\mathsf{I} \mathsf{p}_2 \implies \mathsf{p}_2 \sim \mathsf{p}_3$. The policies p_2 and p_3 express the same restrictiveness in terms of allowed writers and can be seen as equal under the relation $\sim$. A lattice reflecting this ordering is given in Figure 2.6.

2. The conjunction of the integrity policies p_1 and p_2 reduces the authorised principals to principals defined in both p_1 and p_2, and is given by the intersection of their writer sets, $\mathsf{p}_1 \sqcap \mathsf{p}_2 = [\langle \mathtt{carl} \leftarrow \mathtt{carl} \rangle]_\sim$. The disjunction allows principals authorised by at least one of the policies and is given by the union of the corresponding writer sets, $\mathsf{p}_1 \sqcup \mathsf{p}_2 = [\langle \mathtt{carl} \leftarrow \mathtt{ed} \rangle]_\sim$.

2.6.5. Labels

Having both confidentiality and integrity concerns, it is useful to enforce them simultaneously. A *label* expresses both reader and writer policies. The pair $\mathbf{l} = \{\mathsf{c} \in \mathsf{C}; \mathsf{i} \in \mathsf{I}\}$ states that for a given value labelled with the label $\mathbf{l}$, the confidentiality policy c and the integrity policy i must be enforced. The reader set of a label is the reader set of this label's confidentiality policy. The writer set is the writer set of the label's integrity policy.

As labels express restrictiveness in confidentiality and integrity, they can be ordered accordingly. A label $\mathbf{l_1}$ is not more restrictive than a label $\mathbf{l_2}$, if the reader sets of $\mathbf{l_1}$ are always larger than the reader sets of $\mathbf{l_2}$ and if the writer sets of $\mathbf{l_1}$ are always smaller than the writer sets of $\mathbf{l_2}$.

Definition 11 **($\sqsubseteq$)** *Let* $\mathbf{l_1} = \{\mathsf{c}_1; \mathsf{i}_1\}$ *and* $\mathbf{l_2} = \{\mathsf{c}_2; \mathsf{i}_2\}$ *be labels.* $\mathbf{l_1} \sqsubseteq \mathbf{l_2} \iff \mathsf{c}_1 \sqsubseteq_\mathsf{C} \mathsf{c}_2 \wedge \mathsf{i}_1 \sqsubseteq_\mathsf{I} \mathsf{i}_2$.

The ordering $\sqsubseteq$ is reflexive and transitive on the set of labels $\mathbf{L}$. Again, equivalence of labels can be defined as equivalence of reader and writer sets. $\{\mathsf{c}_1; \mathsf{i}_1\} \sim \{\mathsf{c}_2;$

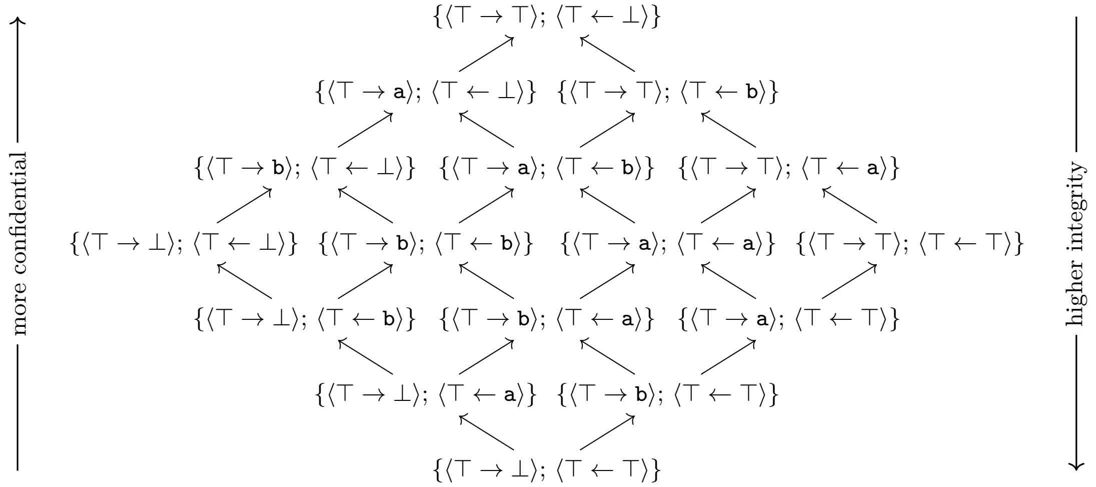

more confidential
higher integrity
$\{\langle\top \rightarrow \top\rangle;\ \langle\top \leftarrow \bot\rangle\}$
$\{\langle\top \rightarrow \mathtt{a}\rangle;\ \langle\top \leftarrow \bot\rangle\}$
$\{\langle\top \rightarrow \top\rangle;\ \langle\top \leftarrow \mathtt{b}\rangle\}$
$\{\langle\top \rightarrow \mathtt{b}\rangle;\ \langle\top \leftarrow \bot\rangle\}$
$\{\langle\top \rightarrow \mathtt{a}\rangle;\ \langle\top \leftarrow \mathtt{b}\rangle\}$
$\{\langle\top \rightarrow \top\rangle;\ \langle\top \leftarrow \mathtt{a}\rangle\}$
$\{\langle\top \rightarrow \bot\rangle;\ \langle\top \leftarrow \bot\rangle\}$
$\{\langle\top \rightarrow \mathtt{b}\rangle;\ \langle\top \leftarrow \mathtt{b}\rangle\}$
$\{\langle\top \rightarrow \mathtt{a}\rangle;\ \langle\top \leftarrow \mathtt{a}\rangle\}$
$\{\langle\top \rightarrow \top\rangle;\ \langle\top \leftarrow \top\rangle\}$
$\{\langle\top \rightarrow \bot\rangle;\ \langle\top \leftarrow \mathtt{b}\rangle\}$
$\{\langle\top \rightarrow \mathtt{b}\rangle;\ \langle\top \leftarrow \mathtt{a}\rangle\}$
$\{\langle\top \rightarrow \mathtt{a}\rangle;\ \langle\top \leftarrow \top\rangle\}$
$\{\langle\top \rightarrow \bot\rangle;\ \langle\top \leftarrow \mathtt{a}\rangle\}$
$\{\langle\top \rightarrow \mathtt{b}\rangle;\ \langle\top \leftarrow \top\rangle\}$
$\{\langle\top \rightarrow \bot\rangle;\ \langle\top \leftarrow \top\rangle\}$

Figure 2.7.: Labels ordered by $\sqsubseteq$.

$\mathsf{i}_2\} \iff ([\mathsf{c}_1]_\sim = [\mathsf{c}_2]_\sim) \wedge ([\mathsf{i}_1]_\sim = [\mathsf{i}_2]_\sim)$. The conjunction and disjunction of labels are given by conjunction and disjunction of the corresponding confidentiality and integrity policies. $\{\mathsf{c}_1;\ \mathsf{i}_1\} \sqcap \{\mathsf{c}_2;\ \mathsf{i}_2\} = \{\mathsf{c}_1 \sqcap \mathsf{c}_2;\ \mathsf{i}_1 \sqcap \mathsf{i}_2\}$ and $\{\mathsf{c}_1;\ \mathsf{i}_1\} \sqcup \{\mathsf{c}_2;\ \mathsf{i}_2\} = \{\mathsf{c}_1 \sqcup \mathsf{c}_2;\ \mathsf{i}_1 \sqcup \mathsf{i}_2\}$. The ordering $\sqsubseteq$ can thus be lifted to a partial order over the equivalence classes. Since the corresponding reader and writer sets do have a greatest lower and least upper bound, and a greatest and smallest element, a lattice can be formed, $(\mathbf{L}/\sim, \sqsubseteq, \{\langle\bot \rightarrow \bot\rangle;\ \langle\top \leftarrow \top\rangle\}, \{\langle\top \rightarrow \top\rangle;\ \langle\bot \leftarrow \bot\rangle\}, \sqcup, \sqcap)$.

Examples

1. Figure 2.7 shows how various labels containing policies for principals $\top$, $\bot$, `alice`, and `bob` from Figure 2.2, are ordered by $\sqsubseteq$.

2.6.6. A Security Enhanced Type System

In traditional programming languages, a value is associated with a type that determines the meaning of the stored bit sequence. This *data type* determines, how a value may be used in further computation and how to apply certain operations. Not only the data type of a value needs to be taken care of, also concerns about confidentiality and integrity must be taken into account when operating on a value or with values derived from it. Especially the *flow* of information must be considered, when building a secure system [DD77]. A security enhanced type system extends the type of a value to incorporate not only the value's data type, but also the associated security policies. Type checking then allows for statical reasoning about secure information flow in a computer program [VIS96].

Definition 12 (Labelled Type) *A* labelled type *for a value* $\mathtt{x}$ *consists of a data type* $\mathtt{T}$ *and a policy label* $\{\mathsf{c};\ \mathsf{i}\}$*:* $\mathtt{T}\{\mathsf{c};\ \mathsf{i}\}\ \mathtt{x}$*.*

Assignment Typing Rule

An expression of labelled type $\mathtt{S}\{\mathsf{c_s};\ \mathsf{i_s}\}$ is assignable to a variable of type $\mathtt{T}\{\mathsf{c_t};\ \mathsf{i_t}\}$, only if $\mathtt{S}$ is a subtype of $\mathtt{T}$, $\mathtt{S} \leq \mathtt{T}$, and if the security policies $\mathsf{c_s}, \mathsf{i_s}$ are not more restrictive than the policies $\mathsf{c_t}, \mathsf{i_t}$, that is $\{\mathsf{c_s};\ \mathsf{i_s}\} \sqsubseteq \{\mathsf{c_t};\ \mathsf{i_t}\}$, for the current principal hierarchy $\mathtt{P}$.

Given an environment γ: `varname` → `type`, that defines a mapping from a variable name to its associated labelled type, assignment can more formally be expressed as:

$$\frac{\begin{array}{l}\gamma, \mathtt{P}\ \vdash\ \mathtt{x}\colon \mathtt{T}\{\mathsf{c_t};\ \mathsf{i_t}\}\\ \gamma, \mathtt{P}\ \vdash\ e\colon \mathtt{S}\{\mathsf{c_s};\ \mathsf{i_s}\}\end{array}}{\gamma, \mathtt{P}\ \vdash\ \mathtt{x} \leftarrow e} \quad \{\mathsf{c_s};\ \mathsf{i_s}\} \sqsubseteq \{\mathsf{c_t};\ \mathsf{i_t}\} \land \mathtt{S} \leq \mathtt{T}$$

The data type part and the label part of a labelled type can be treated independently during type checking. The condition $\mathtt{S} \leq \mathtt{T}$ needs little explanation, since this typing rule is incorporated in every object-oriented language. The $\sqsubseteq$ relation decays into two relations, one for confidentiality and one for integrity. The condition $\mathsf{c_s} \sqsubseteq_{\mathsf{C}} \mathsf{c_t}$ states that it is valid to store public information under policy $\mathsf{c_s}$ to a more secret variable under policy $\mathsf{c_t}$. No information is leaked by assigning a higher confidentiality classification to a piece of information. Similarly, $\mathsf{i_s} \sqsubseteq_{\mathsf{I}} \mathsf{i_t}$ states that it is valid to store trusted information under policy $\mathsf{i_s}$ to an untrusted assignment target under policy $\mathsf{i_t}$. Trusted values can always be used and operated on in an untrusted environment. This is commonly referred to as *write up* in the integrity lattice and *read down* in the confidentiality lattice [SS94].

The process of storing a value with a certain label to a variable (or another value) with a label at least as restrictive, is called *relabelling.* For a relabelling to be successful, the policies must not become less restrictive, otherwise type checking fails. A set of rules describing allowed iterative relabellings can be found [Mye99b]:

Confidentiality Relabelling Rule

1. *Removing a reader from a policy.* The reader sets become smaller and the policy is thus more restrictive.

2. *Adding a policy for another owner.* More confidentiality policies must be enforced. This results in a join (least upper bound) of the policies and is thus more restrictive.

3. *Adding a reader that acts for an already defined reader.* This does not change the reader set as an implicitly defined reader becomes defined explicitly. The reader being acted for can also be removed which results in a replacement by a principal with a higher clearance.

4. *Replacing the owner by a principal that acts for the owner.* All principals acting for the owner are already implicitly readers. Replacing the owner by a principal higher up in the hierarchy removes the old owner from the reader set; the new owner was already a reader. The policy becomes more restrictive.

5. *Adding a principal that acts for the owner to the reader set.* This does not change the reader set as an implicitly defined reader becomes defined explicitly.

Integrity Relabelling Rule

1. *Adding a writer to a policy.* Adding a principal to the writer set lowers the integrity and is therefore more restrictive (in terms of $\sqsubseteq_{\mathsf{I}}$).

2. *Removing a policy.* An owner no longer imposes restrictions on the integrity of a value, hence the writer set become larger and the value requires more restrictions in future computation.

3. *Adding identical policies with a weaker owner.* A policy $\langle \mathtt{o} \leftarrow \mathtt{W} \rangle$ can be added if there already exists a policy $\langle \mathtt{p} \leftarrow \mathtt{W} \rangle$ with $\mathtt{p} \succeq \mathtt{o}$. The newly added policy allows more writers (because of the weaker owner) and is thus more restrictive.

4. *Replacing a writer by a principal the writer acts for.* This adds a new principal to the writer set. The old writer is now implicitly defined.

5. *Removing principals that act for the owner from the writer set.* Explicitly defined writers become defined implicitly.

When a value is computationally derived from one or more operands, the assignment target must ensure, that the policy of each operand is enforced correctly. The target must therefore enforce the security concerns of *all* operands.

Binary Operations Typing Rule

$$\frac{\begin{array}{l}\gamma, \mathsf{P} \ \vdash \ e_1 \colon \mathtt{E}\{\mathsf{c_s};\ \mathsf{i_s}\} \\ \gamma, \mathsf{P} \ \vdash \ e_2 \colon \mathtt{E}\{\mathsf{c_u};\ \mathsf{i_u}\}\end{array}}{\gamma, \mathsf{P} \ \vdash \ (e_1 \ \mathtt{op} \ e_2) \colon \mathtt{E}\{\mathsf{c_s} \sqcup \mathsf{c_u};\ \mathsf{i_s} \sqcup \mathsf{i_u}\}}$$

The weakest policy enforcing both the policies of e_1 and e_2, is the least upper bound (or join) of those policies in their corresponding lattice. Any label that is stricter than the join could be legally used for the intermediate result of the binary operation, but this is not useful in practice. Similar rules apply to operations involving more operands.

2.6.7. Tracking Implicit Flow

Implicit information flow, as described in section 2.2, can lead to severe information leaks. It is therefore important to correctly enforce security policies for values affected by implicit flow. JIF employs a technique called *program counter labelling.*

Every statement (and every intermediate expression) in a program is associated with a program counter (PC) label. Every time the control flow is changed depending on a conditional expression, the PC label for all statements inside of all branches is extended to also enforce the policies imposed on the conditional value, $\mathbf{pc} = \mathbf{pc} \sqcup \mathbf{e_c}$. Then, for every statement in the program, the label of the involved expression is joined ($\sqcup$) with the current PC label. This assigns the least possible label enforcing all, the PC policies and all policies of all operands, to the intermediate value of the expression. Since constant values and literals do not define policies—this is they are labelled with the least possible policy $\{\langle\bot \rightarrow \bot\rangle; \langle\top \leftarrow \top\rangle\}$—their effective label after the join is always the PC label. Listing 2.3 gives an example of implicit flow tracking.

Listing 2.3: Implicit flow tracking with PC labels.

```
Boolean{⟨⊤ → ⊥⟩; ⟨⊤ ← ⊥⟩} granted = false;
String{⟨⊤ → ⊤⟩; ⟨⊤ ← ⊤⟩} correctPassword = "tiger";
String{⟨⊤ → ⊤⟩; ⟨⊤ ← ⊤⟩} inputPassword = readPassword();

// {PC} = {⟨⊥ → ⊥⟩; ⟨⊤ ← ⊤⟩}
// The PC label starts with the least policy possible.

if (inputPassword.equals(correctPassword)) {
  // {PC} = {PC} ⊔ {inputPassword} ⊔ {correctPassword} = {⟨⊤ → ⊤⟩; ⟨⊤ ← ⊤⟩}
  granted = true;
  // {granted} ← {PC} ⊔ {true}
  // Literals and constants have no restrictions on their own,
  // but end up having the same label as their PC.
  // {⟨⊤ → ⊥⟩; ⟨⊤ ← ⊥⟩} ⋢ {⟨⊤ → ⊤⟩; ⟨⊤ ← ⊤⟩} ⊔ {⟨⊥ → ⊥⟩; ⟨⊤ ← ⊤⟩}
  // Assignment is not allowed because {granted} is too weak.
}
```

2.6.8. Termination Labels

When expressions and statements are evaluated, information can be learnt by observing if the evaluation terminates normally, or throws an exception. The `if`-statement in Listing 2.4 demonstrates possible terminations. If the variable `amount` is positive, a `NullPointerException` may be thrown, depending on the value of `account`. If `amount` is zero or negative, the alternative branch is taken, and it is used in division. This may lead to an `ArithmeticException` if the value is exactly 0. It may also be the case, that no exception is thrown and the `if`-statement terminates normally.

A label is assigned to each termination path, reflecting the security concerns of all involved operands. If the `if`-statement terminates with a `NullPointerException` the label assigned to the thrown exception object is $\mathbf{pc} \sqcup \mathbf{amount} \sqcup \mathbf{account}$. The label assigned to a thrown `ArithmeticException` object is $\mathbf{pc} \sqcup \mathbf{amount}$. A non-exceptional termination of the statement is assigned the label $\mathbf{pc} \sqcup \mathbf{amount} \sqcup \mathbf{account}$. The respective labels assigned to the termination paths are called *normal termination label* and *exceptional termination label*, the label assigned to the value an expression evaluates to is called *normal value*. [Cho+09].

Listing 2.4: Termination labels.

```
if (amount > 0) {
  account.balance -= amount;
} else {
  a = 500/amount;
}
```

2.6.9. Handling Exceptions

Throwing an exception from a context that is considered secure, may leak information to the context that catches the exception. Java distinguishes between two types of exceptions. *Checked exceptions* that must be either caught inside of the method they arise or must be explicitly declared to be re-thrown to the caller of the method, and *runtime exceptions* that are unchecked and (if not caught) handled by the Java runtime. To avoid covert channels, JIF requires all exceptions to be checked.

Since information flows from a possibly more secure context inside the `try` block to the receiver of the exception, label checking must be employed. The exceptional termination label of the statement that throws an exception must therefore be less restrictive than the label of the exception declared in the catch clause, $\mathbf{L1} \sqsubseteq \mathbf{pc} \sqcup \mathbf{S}$ or $\mathbf{L2} \sqsubseteq \mathbf{pc} \sqcup \mathbf{S}$.

Control flow mechanics of a `try-catch-finally` are identical to Java. Execution is started in the `try` block which either terminates successfully or throws an exception. If it throws an exception and a suitable `catch` clause exists, control is transferred to the `catch`. After the `catch`, or a successfully terminated `try`, control is transferred to the `finally` block. If no suitable `catch` clause exists for a thrown exception, it must have been declared to be re-throwable in the method signature (`throws` clause); execution is then transferred from the failing `try` to the `finally` block; afterwards, the exception is re-thrown to the caller of the method.

```
try {
  S;
  ...
} catch (Exception1{L1} e1) {
  ...
} catch {Exception2{L2} e2) {
  ...
} finally {
  ...
}
```

2.6.10. Authority

Code executed in JIF runs on behalf of a set of principals, this is referred to as *authority*. For a set of statements, principals in the authority delegate authoritative power to JIF. Special operations like downgrading of security policies (see below) require the accordance of the principal that owns the policy, or a principal that is allowed to act for the owner.

Authority can be defined per class, enabling its methods to declare that they need the authority of one of the principals defined for the class:

```
class C authority(alice, bob) {
  void m() where authority(alice) {
    ...
  }
}
```

Another variant of defining the authority is to declare that a method may only be called from a call site that has the authority of a principal. The class does not have to declare any authority:

```
class C {
  void m() where caller(carl) {
    ...
  }
}
```

2.6.11. Methods

Declaration of methods in JIF is similar to Java. Additional policies can be declared to make sure a method does not leak information.

To restrict calling of methods that may leak information, a *begin-label* can be specified in JIF. This restricts the call to the method, such that the program counter label of the call site is less restrictive than the begin-label of the method, **pc** $\sqsubseteq$ **begin-label**. The begin-label defaults to $\langle \top \rightarrow \top \rangle$.

```
void m{⟨alice → bob⟩; ⟨carl ← don⟩}() {
  ...
}
```

The (non-exceptional) termination of a method may leak information to an attacker. To bound what can be learnt by an observer, JIF supports *end-labels*, that state that the program counter label at the call site must be stricter than the end-label of the called method, **end-label** $\sqsubseteq$ **pc**. The end-label defaults to to least upper bound of all exceptions thrown or to $\langle \bot \leftarrow \bot \rangle$ if no exceptions are thrown.

```
void m() : {⟨alice → bob⟩; ⟨carl ← don⟩} {
  ...
}
```

All method parameters can express a bound on the information that may be learnt, by using a labelled type for method parameters. If no label is stated, the default label $\langle \top \rightarrow \top \rangle$ is assumed.

```
void m(int a, string{⟨alice → bob⟩; } b) {
  ...
}
```

Similarly, the information transferred by a method return value can be bounded by using a labelled type. If no return label is specified, it defaults to the least upper bound of all method parameters and its end-label.

```
int{⟨⊤ → carl⟩; ⟨bob ← emps⟩} m() {
  ...
}
```

Exceptions declared to be thrown by a method may also leak information and can therefore be bounded by using labelled types for the exceptions.

```
void m() throws E{⟨alice → bob⟩; ⟨carl ← don⟩} {
  ...
}
```

Additionally, methods can declare constraints regarding the authority, see above.

2.6.12. Downgrading

Since security lattices, as described above, force labels of a variable to always become stricter, a mechanism for disclosure of information is needed. In JIF this is called *downgrading*. An example for safe disclosure of information that depends on confidential information is the disclosure of a ciphertext (given encryption was done with a reasonable cipher). The ciphertext naturally depends on a secret key, that is usually labelled with a strict policy.

JIF knows two mechanisms to downgrade labels, both of which require the authority of the current method to contain the owner of the policy that is to be weakened:

Downgrading of Confidentiality

To weaken the confidentiality policy of an expression, JIF offers a special *declassify expression* that allows for relabelling of a confidentiality policy. The label of `expression` in the example is downgraded from **L** to **NL**. This means that the label of `expression` is temporarily (for the time of assignment) replaced by the new label **NL**. Default label checking rules, as described above, are used to check if the assignment is valid. After the assignment, the label of `expression` is changed back to **L**.

```
new_expresion = declassify(expression, L to NL);
```

Additionally, a *declassify statement* exists. This is used to downgrade confidentiality policies of the program counter label (**pc**) for a sequence of statements.

```
declassify (L to NL) {
  S1;
  S2;
  ...
}
```

Both, the declassify expression and the declassify statement require the new label to not alter the integrity policies.

Downgrading of Integrity

To weaken the integrity policies of an expression, the *endorse expression* is used. The relabelling works similar to relabelling of confidentiality policies, described above.

```
new_expresion = endorse(expression, L to NL);
```

Additionally there is an *endorse statement* that allows for downgrading of the program counter label (**pc**).

```
endorse (L to NL) {
  S1;
  S2;
  ...
}
```

As with declassify, endorse must only change the integrity policies of a label.

Typing Judgements

Downgrading can be expressed more formally in terms of typing judgements used during label checking. Since the downgrade expression (or statement) requires the current authority to contain the owner of the policy to be downgraded, a downgrade can be stated as:

$$\frac{\mathtt{A} \vdash \mathbf{L_A} = (\bigsqcup_{\mathtt{p}\in\mathtt{A}} \langle \mathtt{p} \leftrightarrow \mathtt{p} \rangle)}{\mathtt{A} \vdash \text{downgrade } \mathbf{L} \text{ to } \mathbf{NL}} \quad \mathbf{L} \sqsubseteq_{\mathsf{X}} \mathbf{NL} \sqcap \mathbf{L_A} \wedge \mathbf{L} \sim_{\bar{\mathsf{X}}} \mathbf{NL}$$

Where $\mathtt{A}$ is the current authority and $\langle \cdot \leftrightarrow \cdot \rangle$ depicts either a confidentiality policy $\langle \cdot \rightarrow \cdot \rangle$ when declassifying, or an integrity policy $\langle \cdot \leftarrow \cdot \rangle$ when endorsing. $\sqsubseteq_{\mathsf{X}}$ denotes the corresponding binary relation $\sqsubseteq_{\mathsf{C}}$ for confidentiality or $\sqsubseteq_{\mathsf{I}}$ for integrity. $\sim_{\bar{\mathsf{X}}}$ denotes the equivalence relation for the policy type opposite to X.

2.6.13. Working with Arrays

Arrays can be regarded as a set of single elements that are contained in a structure. Due to this, an array in JIF has two labels attached. One label for the base type of the array (**LB**), and one for the structure itself (**LA**).

```
T{LB}[]{LA} array;
```

In JIF, assignment of (complete) arrays is only permitted if the policy of the array base types is *equivalent*. The following example shows, why this restriction must be imposed:

Listing 2.5: Assignment of arrays.

```
int{⟨b → b⟩}[] x;
int{⟨a → a⟩}[] y;
int{⟨⊤ → ⊤⟩}[] z;

x = new int[3];
y = x;  // prohibited by JIF
z = y;  // prohibited by JIF

z[0] = 42;
// Store a confidential value to z.
// Since y and x are aliases of z, an attacker could read the
// confidential value '42' via the array x that defines
// less restrictive policies than z.
System.out.println(x[0]); // would print '42'
```

2.6.14. Polymorphic Labels

Building reusable data structures raises the requirement to use generic labels in a class. The generic labels may then be substituted by a specific label, depending on the usage of the class. JIF supports *parametrised classes* that take a label or principal parameter and allows for a generic label definition.

A class can declare label and principal parameters that must be stated when an instance of the class is created.

```
public class C[label L, label M, principal P] {
  private int{L; M} x;
  private int{⟨P → ⊥⟩; M} y;
}

C[⟨alice → bob⟩, ⟨carl ← don⟩, ed] c = new C();
```

2.6.15. Java Interoperability

For interoperation with existing Java classes, the concept of *signatures* was introduced in JIF. A signature allows for definition of security policies for already existing library methods. A special JIF file must be provided that defines the information flow behaviour for a method, by declaring policies for input parameters, method begin and end labels, the return value, and exceptions. No method body is provided.

During label checking, JIF uses flow behaviour of the signature file to calculate if a program is leaking information. At run time, JIF references the original implementation of the library method.

2.6.16. Writing JIF Programs

When writing JIF programs in an IDE or a text editor, labels have the following notion:

```
int{alice->bob; carl<-don;*->_} x;
```

Type-labels are enclosed in curly braces, the arrows are expressed as digraph, ⊤ is written as `*` and ⊥ is written as `_`.

2.7. The Polyglot Compiler Framework

The Polyglot compiler framework [NCM03] is an extensible compiler framework that acts as a compiler front end to the Java language. It was designed to be highly customisable and allows for rapid prototyping of new programming languages.

Polyglot provides a complete infrastructure to cover all phases of a compiler, from lexical analysis until code generation. Polyglot compiles programs to Java code which are then passed to a standard Java compiler.

The JIF language described above is based on Polyglot.

2.8. Compiler Construction 101

This section gives an overview of some important compiler construction terms used in later chapters.

2.8.1. Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a very important data structure used in compiler construction. It is a tree that describes the structure of *expressions*. Every (nested) operator in an expression can be expressed as a node in a tree. The operands to the operator are represented as child nodes of the operator node. Using appropriate operators and operands, *"any programming construct can be handled"* [Aho+07, §2.5.1].

Analysing or transforming a program usually includes operations on the AST. The AST must be traversed in some specific order. A common traversal strategy is *depth-first*; it first evaluates the children of a node, before the node itself is evaluated [Aho+07].

AST transformations yield new ASTs. At the end of the optimising phase, an AST can be translated into intermediate code.

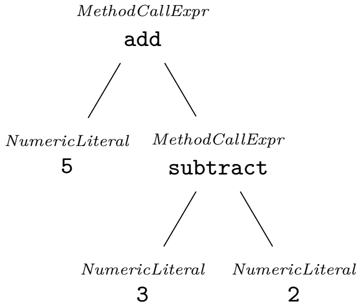


Figure 2.8.: An example of an AST generated from the expression in Listing 2.6.

Example

Parsing the following method call statement may produce for example an AST as shown in Figure 2.8. The AST structure depends on operator precedence.

Listing 2.6: Parsing a method call expression.

```
add(5, subtract(3, 2))
```

2.8.2. Compiler Passes

Compiler construction distinguishes between *multi-pass* and *single-pass* compilers. A single-pass compiler traverses the AST only once. A multi-pass compiler traverses the AST several times, where each pass takes as input the output of the previous pass. This allows for incremental improvement (or transformation) of a program. Multi-pass compilers are more flexible and allow for better optimisations but usually use up more resources during compilation. [Bor90]

2.9. Trusted Execution Environments

A *Trusted Execution Environment* is an execution environment that provides special guarantees in terms of integrity and confidentiality. Code and data present in this environment is protected from manipulation (integrity) and is protected from malicious access from unprivileged readers (confidentiality). A trusted execution environment can thus be used used to isolate the execution of critical code. [Vas+12]

Furthermore, trusted execution environments offer special mechanisms to provide trusted communication with peripherals, e.g. a trusted IO to read characters from a keyboard. Such communication mechanisms are commonly referred to as *trusted path* [Yee02].

A prominent example for a hardware implementation of a trusted execution environment is the ARM TrustZone [ARM].

CHAPTER 3

RELATED WORK

3.1. Security Enhanced Languages

The basic works on security lattices date back to the 1970s and build mostly on the work of Denning [Den76] and Bell/LaPadula [BL73]. Those works focus solely on confidentiality polices. A similar approach for integrity policies was done by Biba [Bib77]. Simultaneous enforcement of both confidentiality and integrity was discussed in literature several times [San93].

JIF is an extension to the Java [AGH96] language and builds on the work of Myers [Mye99b; Mye99a]. It is an implementation of the *Decentralized Label Model* by Myers and Liskov [ML97; ML98; ML00]. JIF implements information flow control as defined by Denning and offers certified compilation. JIF extends the Java type system and attaches confidentiality and integrity labels to a variable [VIS96] and applies the *Decentralized Label Model* [ML97; ML98; ML00] to infer security labels and determine secure information flow as defined by Denning [DD77; Den76]. It makes sure that a program only compiles if the property of *non-interference* [GM82] is met. This means that no information is leaked with respect to the defined policies.

JIF was successfully used to implement large real-world software as for example the *JPmail* email client [HAM06] or *Civitas* [CCM08], a secure remote voting system based on distributed cryptography.

Bebop uses JIF as input language. Denning-style information flow control and JIF itself are described in detail in chapter 2.

Flow Caml [SR03] is an extension to the OCaml language. It supports labelled types similar to JIF, but takes the idea of a secure type system to λ-calculus [PS03]. Flow Caml lacks run-time checks provided by JIF, but employs better type inference so it needs fewer annotations than JIF. Flow Caml could have been an alternative to JIF as *Bebop*'s secure input language.

3.2. Focused on Partitioning for Security

The *Swift* project [Cho+07] is built upon JIF and was designed to automatically perform program partition with respect to secure information flow. Swift focuses on the domain of secure web applications and largely builds upon a dual system comprised of a *server* and *client* principal. Additionally it its possible to define custom principals and design program functionality around these by using the *actsfor*-relation [ML00]. Partitioned code is executed as Java code on the server and translated to client-side JavaScript code with the help of Google Web Toolkit. Partitioning used in Swift was previously explored by the *Jif/split* project [Zda02; Zda+02] which separates Java programs to run on different hosts with different levels of trust. Swift is similar to *Bebop*: both use JIF as input language and partition programs for two target environments. As described above, Swift focuses on web applications and heavily relies on the Google Web Toolkit. *Bebop* was designed to target lightweight environments as for example the ARM TrustZone and *Bebop* does not rely on large third-party libraries. Swift directly emits Java code. One of the objectives of *Bebop* was, to emit JIF code with attached policies when emitting runtime-related code.

Using Swift to achieve the goals of this thesis would have have been an option, but would conflict with some of the objectives defined in chapter 1. Swift does not emit JIF code, it emits Java code. Furthermore, the Google Web Toolkit is tightly integrated into the system and would need to be replaced by a non-Web mechanism. Based on these restrictions we decided to start *Bebop* from scratch.

Fabric [Liu+09] is an extension to JIF and allows security enforced distributed computation on many worker nodes. The security model is very similar to Swift, but Fabric focuses on transactions and remote procedure calls between many nodes with different trust. The partition process, i.e. distributing methods to different hosts, has to be done manually. Fabric's goals are different to the goals of *Bebop*: Fabric aims to provide a way to establish secure grid-computing, where *Bebop* aims to partition programs automatically.

Another area of interest is privilege separation—the automatic partitioning of programs in (large) parts that run under the user's rights, and (small) parts that need elevation. Due to the principle of least privilege [SS75a], program parts that run with special rights should be as small as possible, to prevent possible damage. *Privtrans* [BS04] allows for annotating C programs to enforce security policies. In a C-to-C compilation process programs are automatically partitioned into a monitor, that runs with elevated rights, and a slave that runs with user rights. The approach is successfully applied to OpenSSH, which previously was also separated manually [PFH03] to prevent privilege escalation. Building on the idea of privilege separation, libraries like *Privman* [Kil03] seek to ease the development of Unix applications written in C. Although, decision what has to be executed with which rights has to be completely defined manually. The solutions described above focus on partitioning programs that run on a single host but under different processes with different access rights.

The *ProgramCutter* project [Wu+13] is using a dynamic data dependency analysis that tracks data flow between functions. A graph based approach is used to automatically separate program parts that use privileged system calls from program parts that can run in unprivileged mode. Instead of static analysis, ProgramCutter uses execution traces to determine the data flow in a program. The tool partitions a program into multiple processes that are executed on a single host. *Bebop* calculates a program partitioning automatically, depending on the annotated security policies. *Bebop* is generic and allows to run split program on different hosts that communicate via an encrypted connection.

3.3. Focused on Partitioning for Convenience

Many projects focus on easing the development and hardening of web applications.

Hop [SGL06] is a higher-order language focusing on rich interactive web user interfaces. The LISP-like language allows for writing code in a main *stratum* and a GUI stratum and automatically partitions the program in server code and JavaScript client code. The partitioning decision is more or less based on the strata and not derived from information flow. Hop employs its strata-based program partitioning to enforce separation of user interface from business logic. This differs from the goals of *Bebop*, which bases the partitioning on secure information flow and to partition a program with respect to different levels of trust.

The *Links* language [Coo+07] is a functional language and has similar goals as Hop. It allows the programmer to write web application code that is automatically partitioned into business logic parts that run on a server, and presentation parts that run on a client. It recently got extended by the *Fable/SELinks* [SCH08] project. Fable is a labelled type system for defining security concerns, similar to JIF. It allows to define security policies to control information flow in a program. SELinks is an implementation of the Fable system into the Links language. It allows to write Links programs annotated with security policies and it ensures that the policies are enforced. SELinks—just as the Links language—aims to partition web applications. *Bebop* aims to target applications that run in isolated, trusted execution environments, e.g. the ARM TrustZone.

Dynamic partitioning [CM10] seeks to decide at runtime where to execute code. A cloud-based server and a (mobile) device share the same code. The device reasons about the partitioning on the basis of its capabilities and processing power. It is possible to pin methods to a location due to security reasons or the need for sensors on devices. Separation is done on a per-method basis. Dynamic partitioning focusses on optimally using the resources of a device, whereas *Bebop* focusses on security. Also, *Bebop* is able to partition applications on a per-statement basis.

3.4. Trusted Execution Environments

The ARM TrustZone [ARM] offers hardware support for separating a CPU into a trusted and normal part. Its use with high-level languages is not yet well integrated into recent partitioning tools. Although there exist projects to incorporate the .NET runtime into the TrustZone [San+11; San+14] to provide a more convenient and safer (because managed) way to develop trusted applications, program partitioning must be done by the programmer.

ANDIX [Fit14] is a trusted operation system. It provides a trusted execution environment by using features of the ARM TrustZone. ANDIX would be a perfect trusted execution environment for the use with *Bebop*. Unfortunately, Java was not yet ported to ANDIX.

CHAPTER 4

THE *BEBOP* COMPILER

4.1. Motivation

Two-step authentication is a commonly used mechanism to improve the security of web logins. The algorithm derives a one-time token from a secret that is shared between the web application and the user.

The secret must be stored in a secure way. Typically, the creation of a TOTP token is executed on a smartphone; Google Authenticator[1] is one example for such an application. If the smartphone is equipped with an ARM TrustZone CPU, one possible solution would be, to store the shared secret in trusted storage.

Writing programs that use trusted execution environments is intricate. To execute the TOTP algorithm the developer of the application must now write two programs. One program that runs in the trusted environment and derives the token from the shared secret, and one program that runs in the normal world and displays the derived token to the user.

Manually partitioning an application is cumbersome and may be error prone. We aim to partition applications in an automatic way, based on security policies that are defined by the developer.

Listing 4.1: A schematic version of a TOTP component.

```
function TOTP
  (trusted) user = authenticateUser()
  (?) secret = readSharedSecret(user)
  (?) token = deriveToken(secret)
  (normal) printToken(declassify(token))
end
```

[1] https://play.google.com/store/apps/details?id=com.google.android.apps.authenticator2

Listing 4.1 shows a schematic program that implements user authentication and derives a TOTP token from a shared secret that is stored in trusted I/O. We indicate attached security policies by writing `(trusted)` or `(normal)`. Policies of type `(?)` are not defined by the developer but are inferred later. Information flow analysis determines that the policy in lines 3 and 4 must be `(trusted)` since the secret is read from trusted storage and the token is derived from confidential data. Declassification in line 5 is necessary since `token` was inferred to be of high confidentiality and the `printToken` statement is forced by the developer to be on the normal environment.

Code and data that are labelled with `(trusted)` should be executed in a trusted environment. The remaining code and data should be placed in the normal environment. *Bebop* automatically partitions the application based on secure information flow and allows the developer to write the algorithm in a straightforward manner. *Bebop* partitions on a per-statement level and creates two output programs. Listing 4.2 and Listing 4.3 show a possible partitioning for the TOTP component of Listing 4.1. We evaluate an extended version of this example in detail in chapter 6.

Listing 4.2: Trusted part of the TOTP component.

```
function TOTP_trusted
  user = authenticateUser()
  secret = readSharedSecret(user)
  token = deriveToken(token)

end
```

Listing 4.3: Normal part of the TOTP component.

```
function TOTP_normal



  printToken(token)
end
```

In this chapter we introduce the *Bebop* Compiler that automatically partitions programs based on information flow policies.

4.2. Goals

Converting a textual program to an executable program raises the need for a compiler. To be able to automatically partition programs with respect to their security policies, we must develop an appropriate compiler. We defined a set of objectives for this newly developed compiler:

- *(Objective A)* **Automatic** The compiler shall take a JIF program as input and automatically produce two partial programs as output. One partial program is destined to be executed in a trusted environment, the other partial program is to be executed in the normal environment. The partial programs must—when executed in a cooperative fashion—provide the same functionality as the original program. To enable seamless interaction with JIF, the compiler will be written as an extension to the Polyglot framework.

- *(Objective B)* **Secure** The compiler shall emit JIF code for the partial programs. JIF can then be used to verify security policies for the partial programs and security policies for the emitted runtime interaction.

- *(Objective C)* **Minimal Trusted Computing Base** When determining a program partitioning, a non-trivial partitioning should be generated. This means, that—if possible—not the whole program should be moved to the trusted environment. Only critical statements should run in the trusted environment. The trusted environment partial program should be as large as necessary, but as small as possible.

- *(Objective D.1)* **Confidentiality** The end user should be able to define *confidentiality* policies for the information flow in the program. The compiler shall respected the policies and determine a program partitioning that does not violate these policies.

- *(Objective D.2)* **Integrity** The end user should be able to define *integrity* policies for the information flow in the program. The compiler shall respected the policies and determine a program partitioning that does not violate these policies.

We developed the *Bebop* Compiler that reads input programs written in the *Bebop* Language (cf. subsection 4.3.3) and produces a split program that will execute in two different execution environments *(Objective A)*. The partial programs generated are partitioned in a way that respects defined security policies, such that critical statements are only executed in a trusted environment. The remaining statements are executed in a normal environment. The compilation strategy strives for an optimal partitioning with respect to the amount of code placed to the trusted environment. We avoid that the program as a whole is executed only in a trusted environment, because this environment may have limited resources. The *Bebop* Compiler therefore meets *(Objective C)*.

The *Bebop* Compiler uses the JIF compiler to determine secure information flow of a program and to make sure that policies are satisfied. The annotated policies are used to determine a program partitioning during compilation. The *Bebop* Compiler focusses on confidentiality policies *(Objective D.1)* only, integrity policies are left for future work; hence we failed to meet *(Objective D.2)*. The *Bebop* Compiler, like the JIF compiler, is written as an extension to the Polyglot compiler framework.

We make sure that generated partial programs are still secure in terms of secure information flow. We again use the JIF compiler to check the output of the *Bebop* Compiler for policy violations and information leaks. Assuming that JIF is correct, this gives a guarantee that the program is still secure *(Objective B)*.

To support the *Bebop* Compiler, we developed an accompanying runtime environment and runtime library that are described in detail in chapter 5.

In this chapter we describe how the *Bebop* Compiler works. We give an overview of the underlying architecture and state what transformations are made to the input program. We show how the *Bebop* Compiler integrates into the Polyglot compiler framework and how it interacts with the JIF compiler. For each transformation of the input program we describe in detail why it must be done and how it is carried out.

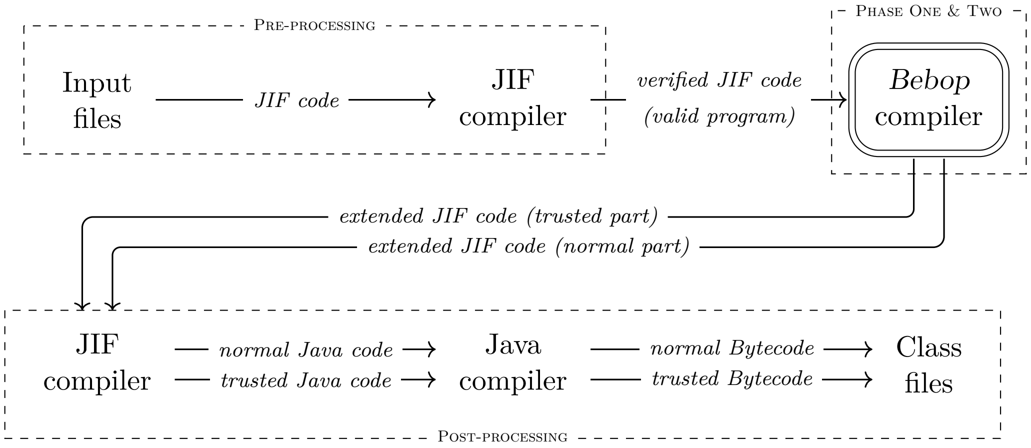


Figure 4.1.: A complete compile cycle for the *Bebop* Compiler.

4.3. Architecture Overview

The *Bebop* Compiler is written as an extension to the Polyglot compiler framework that is described in short in section 2.7. This chapter gives an overview of the general architecture of the *Bebop* Compiler.

4.3.1. The *Bebop* Compilation Pipeline

Compiling a file includes several steps that must be executed. Figure 4.1 gives an overview of the high-level compiler parts that are involved during compilation of an input file.

As a first step, we pass input files to the JIF compiler to make sure that the input satisfies the specified policies. The JIF compiler parses the file, generates an AST and type checks the AST to make sure the program is valid and adheres to the specified policies. The *valid* JIF code is then passed to the *Bebop* Compiler. The *Bebop* Compiler applies its partitioning strategies and generates two outputs, one trusted environment partial program and one normal environment partial program.

These partial programs are again passed to the JIF compiler to make sure that applied transformations and added code still lead to a program that does not violate policies. This is an important step since we defined in our goals that code emitted by the *Bebop* Compiler must be secure. By emitting JIF code we can use the JIF compiler to check if security policies are satisfied.

If the JIF compilation succeeds, its outputs are passed to a Java compiler which creates bytecode files as an output. For each input class one trusted and one normal `.class` file is created.

The *Bebop* Compiler consists of various small steps—called passes—grouped into two phases of compilation. Figure 4.2 gives a more detailed overview of the passes that are executed in the *Bebop* Compiler. Each of these passes is described in detail in the following sections.

4.3.2. Integration in the Compiler Framework

The *Bebop* Compiler depends on the JIF compiler to determine secure information flow. The JIF language allows to specify security policies for a program. The JIF compiler makes sure that these policies are not violated and that information flow is secure. The security model of JIF revolves around a security enhanced type system that is described in section 2.6.

We wrote the *Bebop* Compiler as an extension to the Polyglot compiler framework. Since JIF is already a Polyglot extension, it makes sense to also base our compiler on the Polyglot framework. Figure 4.3 gives an overview.

We use the JIF compiler in the pre- and post-processing steps of the *Bebop* compilation pipeline, described in subsection 4.3.1. We use it to make sure that the input program satisfies the specified security policies and to make sure that code generated by the *Bebop* Compiler does not leak information. We also use the JIF language as base for the *Bebop* Language. The *Bebop* Language is a small extension to the JIF language.

4.3.3. The Input Language

We require input programs to the *Bebop* Compiler to be written in a special language we call the *Bebop* Language. The *Bebop* Language is a small extension to the JIF language described in section 2.6.

The only difference to JIF is, that the *Bebop* Language knows a default principal named `trusted`. JIF only knows the default principals $\top$ and $\bot$; other principals must be created during runtime. The newly added principal allows us to express policies for our trusted environment and it is used to define a *threshold label*, as described below. Using only the principals $\top$ and $\bot$ would not be sufficient. Especially downgrading is problematic since JIF does not allow $\top$ to be in the authority of a method. To support downgrading for two principals, the newly introduced principal `trusted` is necessary. The Swift project used a similar approach [Cho+07].

Scanner and parser for the *Bebop* Language are generated from the extended JIF grammar files and are announced to the Polyglot infrastructure when the *Bebop* Compiler extension is loaded.

4.3.4. The *Bebop* Compilation Strategy

The main responsibility of the *Bebop* Compiler is to produce split programs. Since in Java (and therefore in JIF) programs consist of a set of classes, the *Bebop* Compiler focusses on partitioning a class in two parts. One part is destined to run in a trusted environment, the other part will run in the normal environment.

Given an AST of an input class `C` consisting of a set of fields $\mathtt{F}_1, \ldots, \mathtt{F}_n$ and a set of methods $\mathtt{M}_1, \ldots, \mathtt{M}_m$, we derive a placement for each statement in each method by taking into account the policy information stored in the type system. Attaching a placement

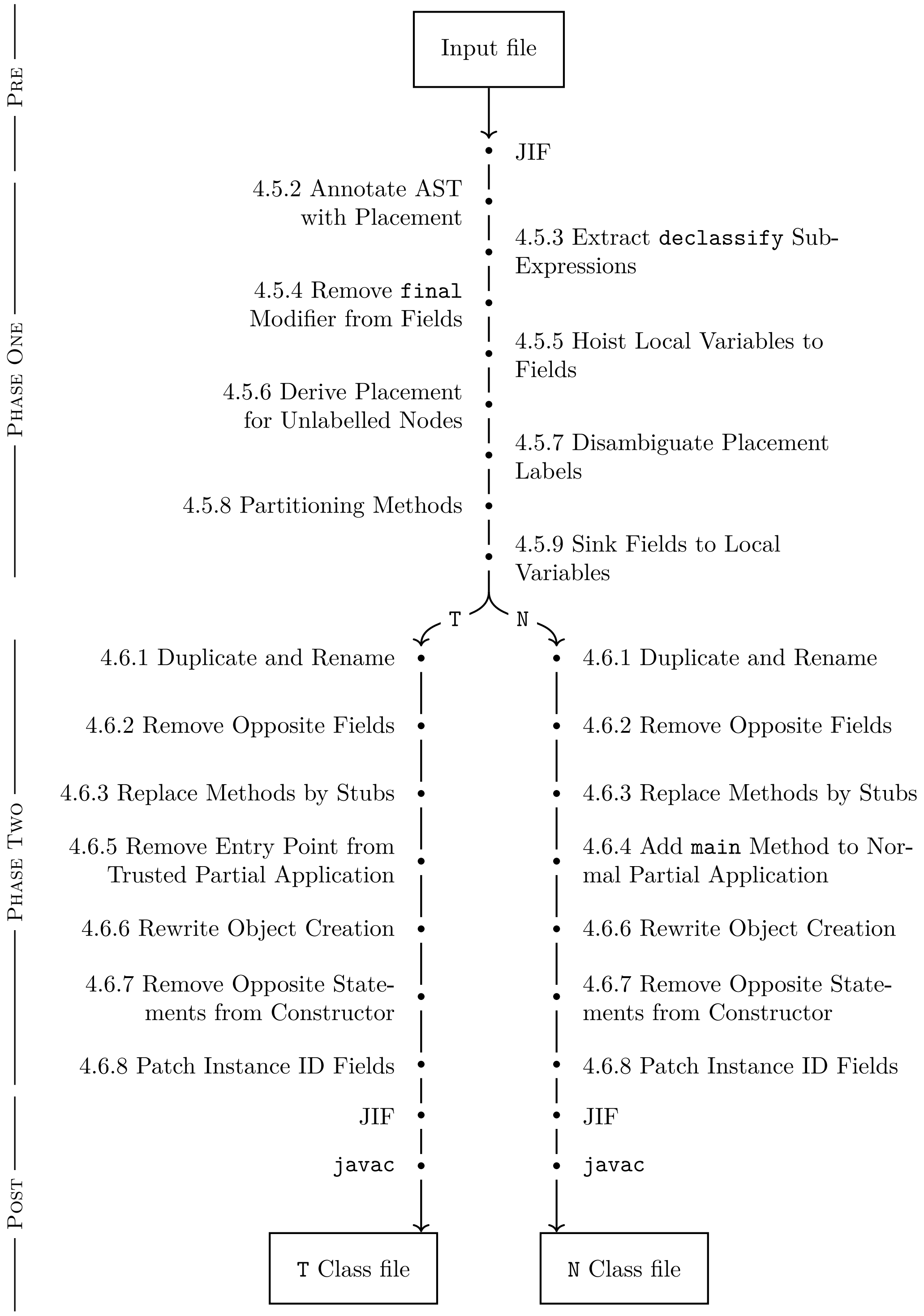

Pre
Phase One
Phase Two
Post
Input file
JIF
4.5.2 Annotate AST with Placement
4.5.3 Extract declassify Sub-Expressions
4.5.4 Remove final Modifier from Fields
4.5.5 Hoist Local Variables to Fields
4.5.6 Derive Placement for Unlabelled Nodes
4.5.7 Disambiguate Placement Labels
4.5.8 Partitioning Methods
4.5.9 Sink Fields to Local Variables
T
N
4.6.1 Duplicate and Rename
4.6.2 Remove Opposite Fields
4.6.3 Replace Methods by Stubs
4.6.5 Remove Entry Point from Trusted Partial Application
4.6.4 Add main Method to Normal Partial Application
4.6.6 Rewrite Object Creation
4.6.7 Remove Opposite Statements from Constructor
4.6.8 Patch Instance ID Fields
JIF
javac
T Class file
N Class file

Figure 4.2.: The sequence of passes executed during compilation of a *Bebop* program. Phase One and Phase Two are the core of the *Bebop* Compiler.

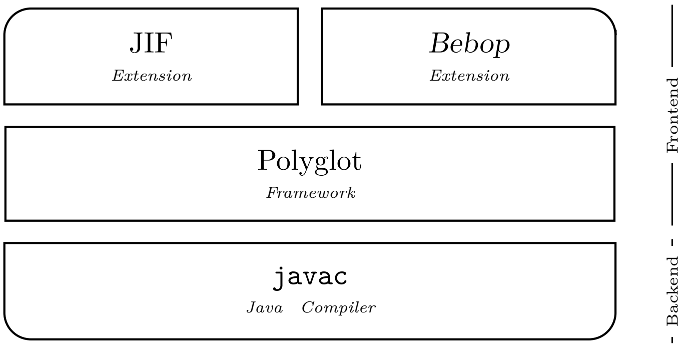
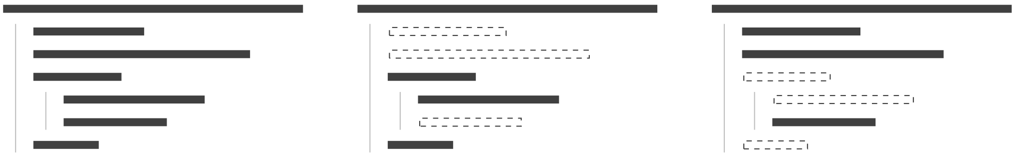


Figure 4.3.: The *Bebop* Compiler is—like JIF—an extension to the Polyglot compiler framework.



(a) Original. (b) Trusted environment. (c) Normal environment.

Figure 4.4.: Policies affect the placement of statements. Consecutive statements with same placement can be grouped and executed in different environments.

label of either „trusted environment“ or „normal environment“ to each statement requires several passes over the AST, as described in section 4.5. Placement labels are stored to AST nodes and are not to be confused with policy labels that extend the type system, as described in section 2.6.

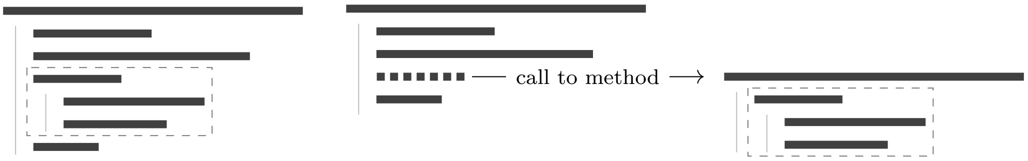
Having placement information for each statement, we can then inspect the statements inside of methods and group consecutive statements with similar placement into *statement groups*. A method can be seen as an interleaving of statement groups executing in different environments, as Figure 4.4 depicts.

We decompose a method into several new methods, each consisting of a statement group. The original method contains calls to „orchestrate“ the calling of method groups. Figure 4.5 gives a visual representation of the process.

Creating small new methods, where all statements have the same placement, allows us to assign the same placement to the method itself. This enables us to assign methods to one specific execution environment.

The *Bebop* Compiler produces two classes as output. The class $\mathtt{C}_\mathtt{T}$ is destined to be executed in the trusted environment, the class $\mathtt{C}_\mathtt{N}$ is destined to be executed in the normal environment. Both classes still contain all methods $\mathtt{M}_1, \ldots, \mathtt{M}_\mathtt{m}$, and they contain additional (synthetic) methods that were created when moving out statement groups.

In partial classes for the normal environment the methods labelled „trusted environment“ are replaced by *method stubs* that contain remote method invocation calls to the trusted environment, as described in subsection 4.6.3. The same process is applied to



(a) Original method.

(b) Statements with same placement moved out to a new method.

Figure 4.5.: Statement groups are moved to new methods and the original statement group is replaced by a call to the new method. This allows us to assign a definite placement to methods. Methods with definite placement can be distributed between the two execution environments.

methods labelled „normal environment“ of partial classes for the trusted environment.

Fields are distributed between the trusted and normal environment. The fields contained in $\mathtt{C_N}$ are a subset of the original set of fields available, $\mathcal{F}_N \subseteq \{\mathtt{F_1}, \ldots, \mathtt{F_n}\}$. The same holds for the trusted-class $\mathtt{C_T}$, $\mathcal{F}_T \subseteq \{\mathtt{F_1}, \ldots, \mathtt{F_n}\}$.

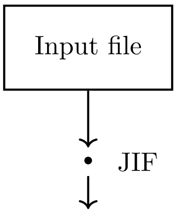
Data flow from and to fields is restricted by the annotated security policies. Secure information flow imposes restrictions on the program. This allows us to distribute fields to separate classes, since data dependencies influence secure information flow and vice versa. The only point where secure information flow is „circumvented“ is when declassification occurs. Such cases are treated in a special way. The partitioning process is described in detail in the following sections.

4.3.5. Interacting with the *Bebop* Runtime Environment

Partitioning a program and running it on different execution environments raises the need for additional mechanisms that execute partial programs and enable communication between partial programs. Code generated by the *Bebop* Compiler is designed to interact with the *Bebop* Runtime Environment that provides such mechanisms. The *Bebop* Runtime Environment is described in detail in chapter 5. Interaction with the *Bebop* Runtime Environment is achieved by using JIF method signatures as described in subsection 2.6.15.

4.4. Pre-Processing Phase

Figure 4.6: Pre-processing steps include scanning, parsing and using JIF to make sure the unsplit program is correctly typed and secure.



For every input file of a program, we first must execute the scanning and parsing steps. We use the previously described *Bebop* grammar to generate a scanner and a parser with the help of *JFlex* and *CUP*, as intended by the Polyglot infrastructure. After successful parsing, the initial AST is created.

In the next step we must check that the program is correctly typed and that language rules are followed. Since the *Bebop* Language is only a small extension to the JIF language, we use JIF to execute this step. We use all JIF passes, but exclude the code generation step.

If all JIF passes execute successfully, we have made sure that the input program satisfies the language rules and that information flow adheres the specified policies. The AST in this state could be used for code generation of a JIF program. Since we have made sure that we are dealing with valid input, control is now handed back to the *Bebop* Compiler.

4.5. Phase One

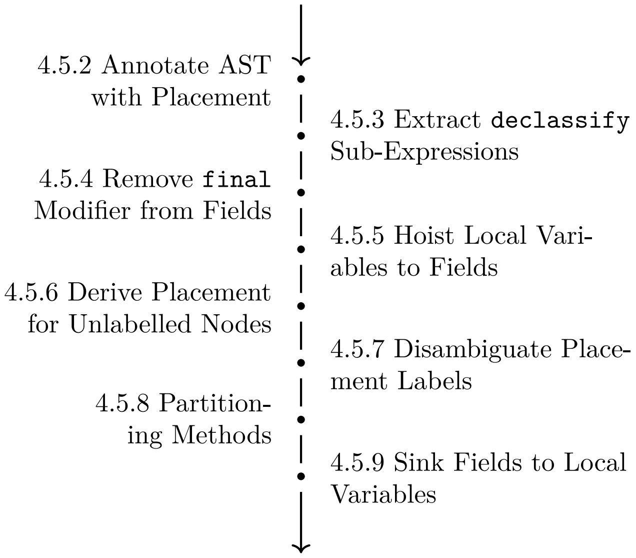

4.5.2 Annotate AST with Placement
4.5.3 Extract `declassify` Sub-Expressions
4.5.4 Remove `final` Modifier from Fields
4.5.5 Hoist Local Variables to Fields
4.5.6 Derive Placement for Unlabelled Nodes
4.5.7 Disambiguate Placement Labels
4.5.8 Partitioning Methods
4.5.9 Sink Fields to Local Variables

Figure 4.7: All passes that happen in Phase One of the compilation pipeline.

We call the first set of *Bebop*-specific passes that are executed *Phase One.* In Phase One we transform the input AST in preparation for the partitioning into two classes that happens in Phase Two. The passes 4.5.2, 4.5.6 and 4.5.7 determine a placement for each AST node. The passes 4.5.3, 4.5.4 and 4.5.5 transform the AST to make it more „splittable“. This includes for example introduction of temporary variables for sub-expressions or hoisting (i.e. increasing the scope) of local variables to fields. The pass 4.5.8 partitions methods into smaller methods that can be assigned to a specific execution environment. The pass 4.5.9 applies an optimisation. Figure 4.7 shows the sequence of passes executed. The following subsections describe the passes in detail.

4.5.1. Finding the Placement of a Statement

We must determine a placement for every statement in each method. We use a multipass approach to infer a statement from policy information stored in the AST. The sections 4.5.2, 4.5.6 and 4.5.7 describe the passes that are involved when determining a placement for a statement.

4.5.2. Annotate AST with Placement

In this pass, we assign a placement annotation to every node of the AST.

To determine a definite placement for every statement of a method in a later pass, we first must determine a placement for every sub node of each statement node in the AST. The placement of subnodes of a statement node determines the placement of a statement node. Placements of statements determine the placement of a method.

Five possible placement qualifiers exist:

- Trusted Environment,
- Normal Environment,
- Both,
- Either,
- Undecided

Placement for sub nodes of method nodes and for field nodes is determined by taking policy information into account. If a policy label exists for an AST node we compare the label against a *threshold label.* A threshold label acts as a barrier between the trusted environment and the normal environment. If a label $\mathbf{L_1}$ is stricter than or equal to the threshold label $\mathbf{T}$, that is $\mathbf{T} \sqsubseteq_C \mathbf{L_1}$, then we assign Trusted Environment as a placement. Likewise, if a label $\mathbf{L_2}$ is weaker than the threshold label $\mathbf{T}$, that is $\mathbf{T} \not\sqsubseteq_C \mathbf{L_2}$, we assign the placement Normal Environment. Thresholding is possible because the policy lattice effectively degrades to a chain due to the limitation of possible principals in the *Bebop* Language, cf. Figure 4.8.

The current implementation of the *Bebop* Compiler focuses solely on confidentiality concerns. Thus we only respect the confidentiality part of a policy label. The threshold label $\mathbf{T}$ is defined as $\mathbf{T} := \langle \texttt{trusted} \rightarrow \texttt{trusted} \rangle$. Figure 4.8 gives an overview what placement is implied by a specific policy.

We assign the placement Both if the node is either:

- a constructor
- a call to a constructor,

The placement Either is assigned to nodes that adapt their placement in a later pass. The placement for such nodes depends on the placement of surrounding nodes. Either is

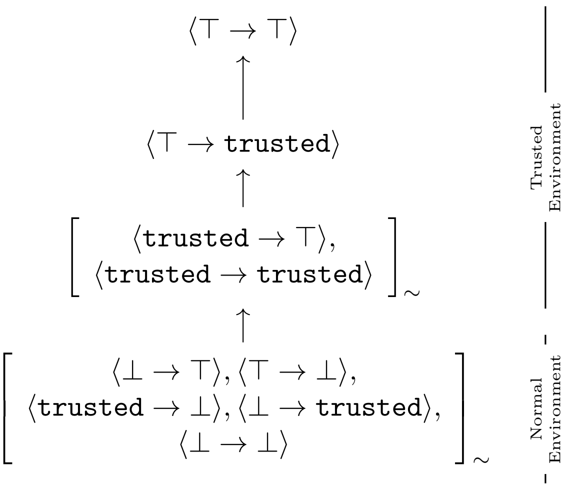

$\langle \top \to \top \rangle$
$\langle \top \to \texttt{trusted} \rangle$
$\left[\begin{matrix} \langle \texttt{trusted} \to \top \rangle, \\ \langle \texttt{trusted} \to \texttt{trusted} \rangle \end{matrix} \right]_\sim$
$\left[\begin{matrix} \langle \bot \to \top \rangle, \langle \top \to \bot \rangle, \\ \langle \texttt{trusted} \to \bot \rangle, \langle \bot \to \texttt{trusted} \rangle, \\ \langle \bot \to \bot \rangle \end{matrix} \right]_\sim$
Trusted Environment
Normal Environment

Figure 4.8: The lattice shows the effective confidentiality policies and the derived placement.

for example assigned to nodes of type *TypeNode*, *ID*, . . . In a later pass we clear these ambiguities and relabel such nodes accordingly, see subsection 4.5.7.

If no policy can be found for a given AST node and it is not of a special type, we assign the placement Undecided. In a later pass we then try to derive a placement for all nodes that were labelled Undecided, see subsection 4.5.6.

4.5.3. Extract `declassify` Sub-Expressions

In this pass we extract all `declassify` sub-expressions and assign their value to a new temporary local variable.

The security lattice described in section 2.6 is a one way lattice, allowing values to only become stricter. To make a value less strict, a *downgrading mechanism* must be used. For expressions, the `declassify` expression allows to re-label the confidentiality policy of an expression.

We must treat `declassify` expressions in a special way. The outer statement—e.g. a method call—may be placed on the normal environment. The inner `declassify` expression produces as a result a value that will most likely be placed on the normal environment, depending on the label. The parameters of the `declassify` expression are most likely bound to the trusted environment, depending on the label. Since partitioning of methods works on a per-statement base, we must move the `declassify` sub-expression out of the original statement, before executing the partitioning. Otherwise, the original statement will not be splittable. We assign the `declassify` sub-expression to a new temporary variable and use this temporary variable as new sub-expression in the original statement. The following example illustrates this:

```
public void m{trusted->_}() where caller(trusted) {
  String{trusted->} v = "Secret";
  n(declassify(v, {trusted->_}));
}
```

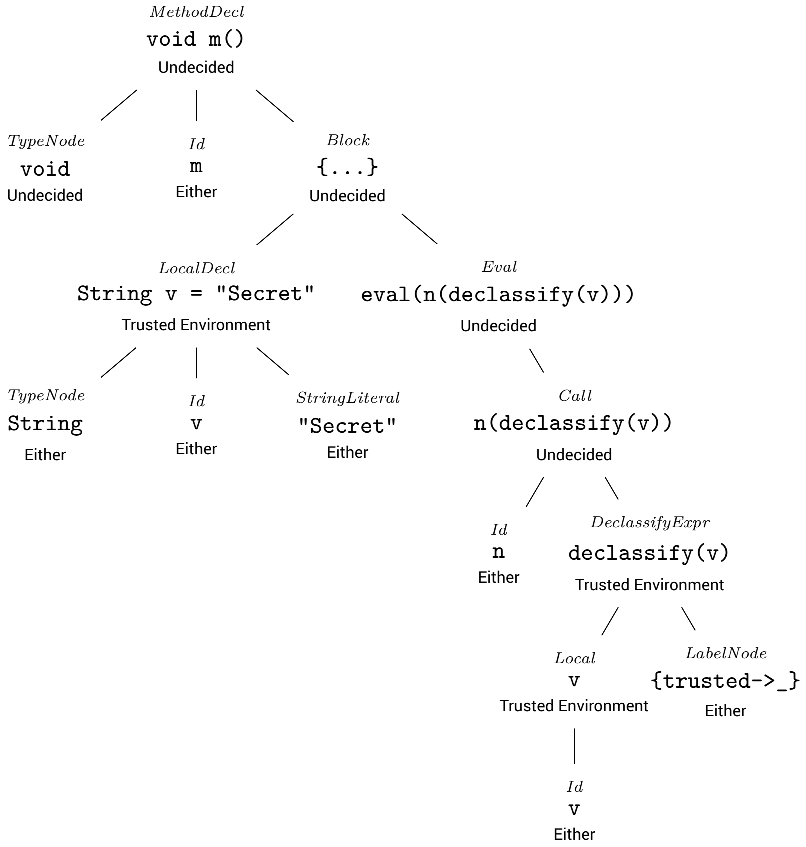


Figure 4.9.: A simplified AST of method m from the example in subsection 4.5.3

```
public void n(String{trusted->_} v) {
  // do something
}
```

Before execution of this pass, the (simplified) AST for method m looks like Figure 4.9. The label derived for the *DeclassifyExpr* node by the previous pass is still marked as Trusted Environment since the special semantics for the `declassify` expression are introduced in this pass.

After extraction of the `declassify` sub-expression, the AST looks like Figure 4.10. The newly introduced temporary variable now has the correct placement Normal Environment.

4.5.4. Remove `final` Modifier from Fields

In this pass we remove the `final` modifier from all fields in a class. This is necessary because partitioning constructors may move assignment to the `final` field out of the constructor.

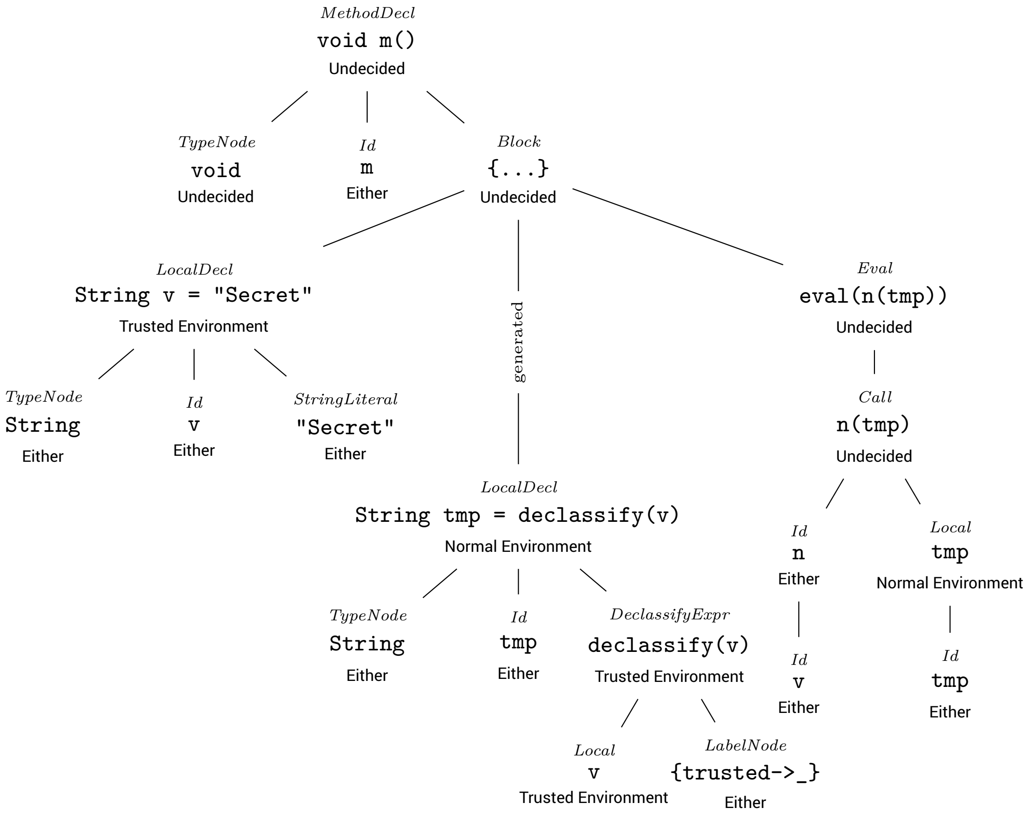


Figure 4.10.: The pass introduced a new *LocalDecl* that holds the *DeclassifyExpr* subtree. The original statement now uses the new temporary variable as sub-expression.

Moving assignment of final fields out of the constructor has two effects: first, it tricks the Java compiler into thinking that the field is not definitely assigned; second, it produces assignments to `final` fields outside of the constructor, which are illegal. This violates the Java Language Specification §8.3.1.2 [Gos+14] that requires `final` fields to be definitely assigned in the constructor, resulting in a compile-time error. Since we are only moving statements to another method that is guaranteed to execute before the constructor ends, we remove the `final` modifier. The fields become *effectively final*, cf. §4.12.4 of the Java Language Specification.

Since fields are effectively final, the `final` modifier could be re-introduced in a later pass, after the partitioning has been executed. This would require inlining of synthetic methods that are called from the constructor. This is an optimisation and may be targeted by future work.

4.5.5. Hoist Local Variables to Fields

In this pass we replace all local variables of all methods by fields.

Partitioning a method into a set of smaller methods and calling those methods may lead to data dependency issues. Consider the following example:

```
public void m() {
  int some_N_value;
  int some_T_value;
  int another_N_value;

  some_N_value = 5;
  some_T_value = 10;
  another_N_value = some_N_value;
}
```

Assume for now, that policies are chosen in a way that the variables `some_N_value` and `another_N_value` are bound to the normal environment, and that the variable `some_T_value` is bound to the trusted environment. A possible normal environment partial program would be:

```
public void m() {
  m_1();
  m_2();
  m_3();
}

// Runs on N
public void m_1() {
  int some_N_value;  // Declared local
  some_N_value = 5;
}

// Runs on T
public void m_2() {
  ...
  // Remote call
  ...
}

// Runs on N
public void m_3() {
  int another_N_value;
  another_N_value = some_N_value;  // Error: 'some_N_value' not found
}
```

We now face the problem that we must somehow transfer all data that `m_3` depends on from `m_1` to `m_3` after `m_2` returned from the trusted environment. One possibility would be to pass data as method parameters, but this leads to cumbersome data synchronisation code generation. Another possibility is to turn all local variables to fields, using random suffixes to avoid name-clashes. Since JIF explicitly forbids concurrency features of the Java language, we cannot run into interleaving problems. This easy form of data synchronisation has one major disadvantage: Due to hoisting a method does not have stack variables any more and hence recursive methods will not work.

We find that the simplicity of data synchronisation outweighs the lack of recursion for the first version of the *Bebop* Compiler. Future work may consider changing the data synchronisation mechanism such that recursive methods are supported.

4.5.6. Derive Placement for Unlabelled Nodes

In this pass, we try to replace all Undecided placement labels by a more specific placement.

The previous placement labelling pass does not always derive a useful placement for AST nodes. Specifically, nodes labelled Undecided must be relabelled. In this pass we visit all nodes below a method node and derive a better placement from information stored in surrounding nodes.

At the end of this pass, all sub nodes of a method node have a placement of either Trusted Environment, Normal Environment, Either or Both. We refer to these three placements as *useful* during this section.

General Inference Rules

When relabelling nodes, we traverse the AST *in-order*. Hence, the bottom-most nodes in the tree are evaluated first. On the way to the leaf-nodes—when we *enter* a node—we store placement information of all parent nodes that were entered on the way to the leaf. On the way up—when we *leave* a node—we relabel if necessary. This implies that child nodes are mutated before their parents.

Traversal of the tree starts at method nodes. Nodes that lie outside of this sub-tree do not need exact labelling—such as class nodes, since classes must be present on both execution environments—or do already have a definite placement, such as fields.

On all nodes in the relevant sub-tree, the following relabel rules are applied.

- Nodes that already have a placement of Trusted Environment, Normal Environment, Either or Both are not changed.

- Parent nodes are labelled Normal Environment, only if all non-Either children are labelled Normal Environment. We exclude nodes labelled Either here, because they adapt their label to the surrounding nodes in a later pass.

- If at least one child node is labelled Trusted Environment, the parent node is labelled Trusted Environment too. This reflects the idea that an expression that contains at least one high confidential (or high integrity) factor forces all derived expressions to also be high confidential (of high integrity).

- Parent nodes are labelled Both, only if all children are labelled Both.

- Parent nodes are labelled Either, only if all children are labelled Either. This is a fall-back rule. If none of the previous rules matched, we carry the Either placement up in the AST.

Handling *Block* and *ProcedureDecl* Nodes

Block and *ProcedureDecl* nodes need special treatment. A *Block* node is the AST's representation of a lexical scope, i.e. it is the parent node of all statements inside of curly braces (`{...}`). We label a *Block* or *ProcedureDecl* Trusted Environment, only if all non-Both children are labelled Trusted Environment. Otherwise the *Block* or *ProcedureDecl* will be labelled Normal Environment.

This relabelling reflects the idea that the parent of a set of statements is only moved to the trusted environment if all its children are to be placed to the trusted environment. This semantic keeps the code for the trusted environment smaller, resulting in a better partitioning with respect to the code size in the trusted environment.

Handling `try-catch` Statements

`try-catch-finally` statements need special treatment. If either the `try`-*Block*, one of the `catch`-*Block*s or the `finally`-*Block* are labelled Normal Environment, we relabel the parenting *Try* node to Normal Environment. This lifts the `try-catch-finally` control structure to the normal environment but allows us to place the bodies of the `try`, `catch` or `finally` clauses to be placed to a different environment. `try` and `finally` *Block*s take the placement of their children, as described above. Otherwise, the whole *Try* node is moved to the trusted environment.

4.5.7. Disambiguate Placement Labels

In this pass we try to find a *definite placement* of either Trusted Environment, Normal Environment or Both. We call this process *disambiguation.*

For disambiguation, we visit all sub nodes of each method node and replace placement labels Either by the placement label of the parent node. If the parent node does not have a definite placement, we recursively search for a parent node with definite placement.

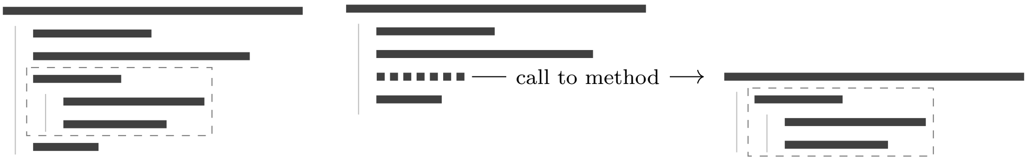
4.5.8. Partitioning Methods

In this pass, we actually partition methods into smaller parts that can be assigned to a specific execution environment.

For each method in a class, we first determine all *statement groups* that make up the method. A statement group is a set of consecutive statements that have the same placement. A statement group may span over basic block boundaries. A method is a sequence of statement groups with alternating placement.

Statement groups can be moved to a new method each, until every method has a definite placement, i.e. all statements in a method will completely run in one specific execution environment. Calls to these new methods replace the statement group in the original method, see Figure 4.11.

If a method consists of only one statement group we assign the placement of the statement group to the method and we are done. If more statement groups exist, we apply the steps described in the following sections.



(a) Original method.

(b) Statements with same placement moved out to a new method.

Figure 4.11.: Replacing statement groups by remote calls to synthetic methods allows assigning a definite placement to each method. Methods can then be distributed between the two environments and called with a remote calling mechanism.

Building New Methods for Moved Statement Groups

For each statement group that is moved out of a method, we must introduce a new method that will contain the statement group.

In the first step we create a new method declaration for a randomly named method with one parameter of type `CallingContext` and with a return type of `void`. The `CallingContext` is used to transfer data dependencies over different execution environments. Furthermore we assign a begin label to the new method. If the parenting statement of the statement group that is processed is a `declassify` statement, we use the label $\mathbf{D}$ that corresponds to the inner environment of the `declassify` statement and the begin label of the original method $\mathbf{M}$ and assign the calculated label $\mathbf{D} \sqcup \mathbf{M}$. Otherwise, we copy the begin label of the parenting method. The end label of the new method is determined by JIF. We scan all statements of the current statement group and declare a `throws` clause for every exception that is thrown. Additional JIF method constraints are copied from the original method.

```
private void partialMethod{beginLabel}(bebop.runtime.CallingContext{__ctx}
    ↪ __ctx) : {endLabel} [throws Exception1, ...] [where caller (trusted)]
    ↪ {
  ...
}
```

All statements of the method group are added as method body. We check each statement if control flow changes must be synchronised as described in section 4.5.8.

Since we move out complete statements from one method to another, data dependencies may no longer be fulfilled in the new method, i.e. we try to use variables that are no longer present because they are declared and/or assigned to in another method now. To compensate for this, we must transfer all necessary data from the original method to the new method. In section 4.5.8 we describe the `CallingContext` that stores all this data. In our new method we now must extract the data and declare the appropriate local variables. We add this code *at the top* of the new method.

```
// Declare local
T fieldName;

// For each field/local that must be synchronised, emit:
fieldName = (T)
    ↪ declassify(__ctx.load(bebop.runtime.BoxHelper.box(fieldName)), ..);
```

Depending on the generated synchronisation code, we may additionally declare a `NullPointerException`, `ArrayIndexOutOfBoundsException`, and/or `ClassCastException` to be thrown by the new method.

After execution of the statements from the statement group, data may have been produced that must be synchronised back to the call site. For each variable that is used after the call to the new method, we must emit synchronisation code *at the bottom* of the new method.

```
// For each field/local that must be synchronised, emit:
__ctx.store("fieldName", bebop.runtime.BoxHelper.box(fieldName));
```

Finally, the placement label for the new method is set to the placement of the statement group that was moved out of the original method.

Invoking Methods for Moved-out Statements

In the original method we replace the moved out statement group with a call to the newly introduced method.

We create a `CallingContext` that wraps all necessary data the newly introduced method needs. The `CallingContext` is a dictionary-like structure that stores values for all variables and fields used in the new method. In return, the new method stores control flow changes and return values to the `CallingContext` that are evaluated at the call site, after the call finished.

```
final bebop.runtime.CallingContext{__ctx} __ctx =
    ↪ bebop.runtime.CallingContext.create();
```

We store data dependencies that have to be transferred over execution environment boundaries in the `CallingContext`. All *opposite* (i.e. with placement different to the newly introduced method) fields, local variables or parameters of the original method that are read in the new method must be transferred to the opposite execution environment. Similarly, all opposite fields, local variables or method parameters written in the new method must be transferred back.

We must emit code that stores all fields and local variables with opposite placement and parameters of the original method that are required in the newly created method to the `CallingContext` before emitting a call to the newly introduced method. First we scan the statements that were moved to a new method and track all the fields, local variables and parameters that are used. For all fields, local variables and used method parameters of the original method in the moved out statement group, we select those with opposite placement. Those must be stored to the `CallingContext`. Required boxing of primitive types is done via a helper method of the `BoxHelper`.

```
// For each field/local that must be synchronised, emit:
__ctx.store("fieldName", bebop.runtime.BoxHelper.box(fieldName));
```

In the next step, we call the newly created method and pass our `CallingContext`.

```
partialMethod(__ctx);
```

After the call—similar to the synchronisation before the call—we must ensure that fields with opposite placement or parameters of the original method that are written to, are updated accordingly. We first scan all statements of the new method if fields, method parameters or local variables are written to. We select those that have opposite placement and are used after the moved out statements have executed. This includes traversal of the control flow graph to make sure we do not miss usages that occur *in a loop body before* the statement group.

Since the `CallingContext` may have a stricter label than the target field or local, we optionally must emit a `declassify` expression or statement to downgrade to the target field.

```
// (Optionally) Declare local if necessary
T fieldName;

// For each field/local that must be synchronised, emit:
fieldName = (T)
    ↪ declassify(__ctx.load(bebop.runtime.BoxHelper.box(fieldName)), ..);
```

Finally, we must wrap the call to the new method including the synchronisation parts in a `try-catch`. Since JIF treats all exceptions as checked exceptions, we may encounter one or more of the following exceptions:

- a `NullPointerException`, when the newly created method accesses fields of object or invokes a call on an object,
- a `ClassCastException`, any time we load from the `CallingContext`,
- an `ArrayIndexOutOfBoundsException`, if we emit synchronisation code for arrays, as described in section 4.5.8.

We inspect the moved out statements to determine which exceptions can be thrown to avoid emitting unnecessary `catch` clauses. Furthermore, if the original method declares one of the exceptions described above to be thrown, we do not emit a `catch` clause either. If possible, we avoid the generation of the `try-catch` as a whole.

If declarations for local variables were emitted, we must move those declarations (without the initialisation part) before the `try` statement. This makes sure that the variables are in scope for statements *after* the generated call to the new method.

If we move out statements of a method that has been moved out itself, we must pass all changes to fields and locals, or changes to the control flow to the outer calling context. We emit code that transfers the contents of one `CallingContext` to another:

```
  __outer_ctx.assimilate(__ctx);
```

Synchronising Data Dependencies

For the sake of simplicity, the previous sections understated the synchronisation mechanics that must be executed. This section details what is actually happening.

We can identify two different mechanisms that are executed. First is *storing to* the `CallingContext`, second is *loading from* the `CallingContext`. The actions are executed pair-wise, once at the call site, once in the new method.

Note: *Bebop* is currently only able to synchronise variables of primitive types, see section 4.8.

Listing 4.4: The `CallingContext` signature file.

```
package bebop.runtime;

public class CallingContext implements java.io.Serializable {
  public static native CallingContext{*->_} create();
  public native void store(String{*->} id, Object{*->} o);
  public native Object{*->_} load{*->}(String{*->} id) : {*->_};
  ...
}
```

Storing Saving data to the `CallingContext` is straightforward. As Listing 4.4 depicts, the `store` method takes the name of a variable and an object, both annotated with the strictest confidentiality policy. This allows us to call the method with arguments of any confidentiality policy, as $\langle \top \rightarrow \top \rangle$ is always at least as strict as the policy of the argument and thus the method parameter is „assignable“. Since JIF has no auto-boxing capabilities, we wrap the object that is to be stored in a method call to the `BoxHelper` of our *Bebop* Runtime Library. The `box` method either boxes a primitive type or returns the reference if a non-primitive type is passed. Listing 4.5 gives an overview of the boxing mechanics.

Listing 4.5: The `BoxHelper` allows for convenient boxing.

```
public class BoxHelper {
  public static Object{*->_} box(Object{*->} o) { return o; }
  public static Byte{*->_} box(byte{*->} b) { return new Byte(b); }
  public static Integer{*->_} box(int{*->} i) { return new Integer(i); }
  ...

  public static native byte{*->_}[]{*->_} unboxByteArray(Object o);
  public static native int{*->_}[]{*->_} unboxIntegerArray(Object o);
  ...
}
```

The final *store* statement—emitted for each field that needs to be synchronised—has the following form:

```
__ctx.store("fieldName", bebop.runtime.BoxHelper.box(fieldName));
```

Loading Loading data from the `CallingContext` requires more attention.

In the first step we create a new label $\mathbf{L_1}$. If the type of the field we want to load is labelled, we set $\mathbf{L_1}$ = labelof(typeof($field$)). If the type of the field is not labelled, we set $\mathbf{L_1}$ = **pc**, where **pc** is the current program counter label. If the type was not labelled, we create a label variable to let JIF derive a label for the type.

If the type of the field is not an array, we can load it from the `CallingContext`, cast it to the appropriate type and issue a downgrading. The outer `declassify` statement downgrades the program counter. This statement is only generated if $\mathbf{L_1}$ was *not* set to **pc**. The `declassify` expression on the right hand side of the assignment is generated because the label of `__ctx` may be stricter than the label of `T`.

```
declassify({__ctx} to {l1}) { // optional
  T fieldName = declassify((T) __ctx.load("fieldName"), {__ctx} to {l1});
}
```

If the type of the field that needs to be synchronised is an array, more complex code is generated. JIF permits assigning of arrays only if the policies are equivalent (due to aliasing, see subsection 2.6.13). To transfer the content of one array to another, we must issue a copy loop.

We first emit code to load the array from the `CallingContext` to a temporary variable. The outer `declassify` statement is optional, as described above. Since JIF does not support *casting* to arrays, we generate code to work around this limitation. A call to `BoxHelper.unbox`*Type*`Array` is issued, where Type is replaced by the respective type of the array. The `unbox`*Type*`Array` methods internally cast the Object to an array of Type and return a copy of this array.

```
declassify({__ctx} to {l1}) { // optional
  T[] tmpArray = declassify(bebop.runtime.BoxHelper.unboxIntArray(
      ↪ __ctx.load("fieldName")), {__ctx} to {l1});
}
```

We then emit a declaration for a new array of the same type and length as `tmpArray`.

```
T[] fieldName = new T[tmpArray.length];
```

Afterwards, we emit a copy loop that copies single array elements. The access to the target array may need declassification.

```
for (int i=0; i < tmpArray.length; ++i) {
  declassify(fieldName[declassify(i, {__ctx} to {fieldName})], {__ctx} to
      ↪ {fieldName}) = declassify(tmpArray[declassify(i, {__ctx} to
      ↪ {fieldName}], {__ctx} to {fieldName});
}
```

Synchronising Control Flow Changes

Moving out statements of a method may affect control flow structures in the program. If the body of a loop contains control flow changing commands like `break`, `continue` or `return`, the control flow of the program is changed.

If we move parts of a loop body that contains control flow changing statements to a new method, we must ensure to synchronise control flow changes back to the call site. This includes replacing the `break` and `continue` commands by appropriate runtime functions—because branching statements without a parenting loop are illegal—and emitting synchronisation code at the call site.

When visiting the AST for a method declaration, we track if we visited a *Loop* node. If this is the case we store that an outer loop is present. We scan all statements of the statement group that is to be moved out if it contains a `break` or `continue` statement. If this is the case and an outer loop is present, we store the control flow change in the context and replace the statement by a `return`.

```
// Before
private void new_synth_method(bebop.runtime.CallingContext __ctx) {
  ...
  break;
  ...
}

// After
private void new_synth_method(bebop.runtime.CallingContext __ctx) {
  ...
  __ctx.setDidBreak();   // or __ctx.setDidContinue()
  return;
  ...
}
```

If we encounter a `return` statement in the newly created method, we must replace it by appropriate runtime calls to store the return value to the `CallingContext`, because the newly created method was declared to return `void`. Since JIF derives from (approximately) Java 1.4, we use the `box()` method of the `BoxHelper` that provides various overloads to box primitive types if necessary, but also accepts objects that are „routed through“.

```
// Before
private void new_synth_method(bebop.runtime.CallingContext __ctx) {
  ...
  return someValue;
  ...
}

// After
private void new_synth_method(bebop.runtime.CallingContext __ctx) {
  ...
  __ctx.setReturnValue(bebop.runtime.BoxHelper.box(someValue));
  return;
  ...
}
```

At the call site, we must insert appropriate statements to unpack and return the return value stored in the `CallingContext`. We add such statements *only if* the moved out statements included a **`return`** statement. Values retrieved from the `CallingContext` must be cast to the appropriate return type of the original method.

```
return (SomeType) ret.getReturnValue();
```

4.5.9. Sink Fields to Local Variables

In this pass we *sink* fields of a class back into local variables of a method. Sinking is the process of moving variables from an outer scope to an inner scope, if the variable is only used in the inner scope. This is also possible for fields.

Since not all local variables need data synchronisation between method calls, it is possible to „undo“ the hoisting pass described in subsection 4.5.5 for *some* of the fields, after we partition methods into smaller parts. This reduces the amount of fields in a class and sinks appropriate fields back to a more narrow scope, turning them into stack variables which are in general „cheaper“ than fields. To achieve this goal, we first analyse the usage of each field. If it is used in *only one method*, we are allowed to sink it back to a local variable.

4.6. Phase Two

The *Phase Two* of the *Bebop* compilation cycle finalises the program partitioning.

We take as input in Phase Two an AST of an input class. The AST was already transformed in a way that aids the partitioning process. The main transformation we applied has partitioned methods into smaller parts that can be assigned a definite placement. Phase Two now creates classes for each execution environment and applies changes that wire the program to the *Bebop* Runtime Environment. Additionally, code that does not

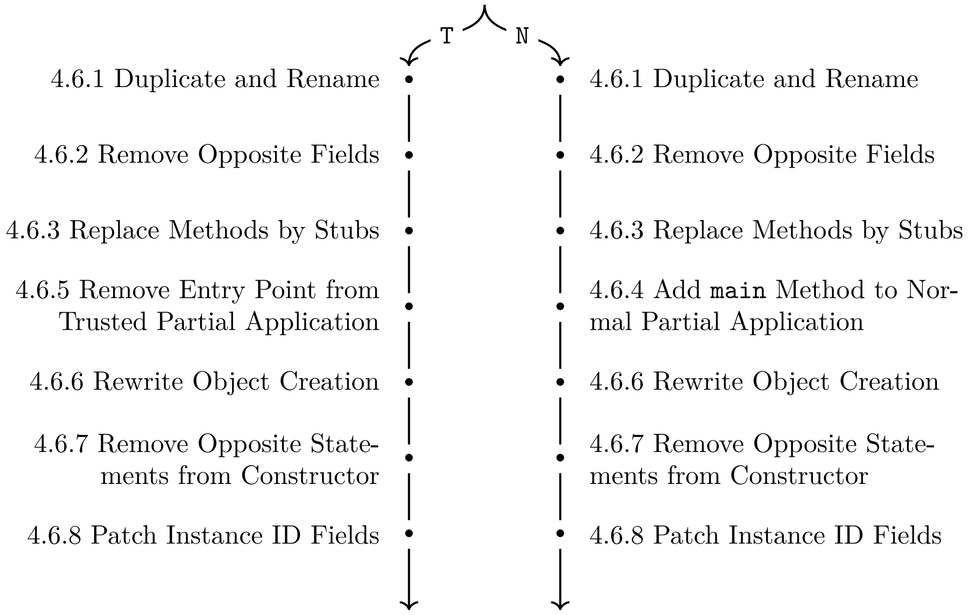

T
N
4.6.1 Duplicate and Rename
4.6.2 Remove Opposite Fields
4.6.3 Replace Methods by Stubs
4.6.5 Remove Entry Point from Trusted Partial Application
4.6.6 Rewrite Object Creation
4.6.7 Remove Opposite Statements from Constructor
4.6.8 Patch Instance ID Fields
4.6.1 Duplicate and Rename
4.6.2 Remove Opposite Fields
4.6.3 Replace Methods by Stubs
4.6.4 Add main Method to Normal Partial Application
4.6.6 Rewrite Object Creation
4.6.7 Remove Opposite Statements from Constructor
4.6.8 Patch Instance ID Fields

Figure 4.12.: All AST manipulations that happen in Phase Two of the compilation pipeline.

belong to an environment is removed. Figure 4.12 gives an overview of the passes we apply. This section describes each pass in detail.

4.6.1. Duplicate and Rename

In this pass we duplicate the AST of the original input class to create two distinct classes for the two execution environments.

The compiler produces as output two partial programs, one for the trusted environment and one for the normal environment. Since the two partial programs need slightly different treatment during compilation, we duplicate the AST for each original class for further processing. This includes the appropriate duplication of the type objects in the Polyglot type system. In order to make a clear distinction from the original class, we rename it with a suffix of either `_N` for classes destined to reside on the normal environment or `_T` for classes destined to reside on the trusted environment. Another reason for the rename is that the underlying polyglot compiler framework would have needed rework to support two different types and ASTs sharing the same name.

4.6.2. Remove Opposite Fields

In this pass we remove all fields that have a placement opposite to the placement of the partial class that is currently processed.

Since code that resides on the normal environment does not depend on fields destined to be on the trusted environment, we can remove fields labelled Trusted Environment from the normal environment class, and vice versa. The only exception to this data flow is the use of a `declassify` statement or expression. This case is handled separately and described in subsection 4.5.8.

4.6.3. Replace Methods by Stubs

In this pass we replace methods determined to run on the opposite environment by *method stubs* that execute a remote call. We use the term *method stub* as it is used in Java RMI nomenclature [Ora14, §3.1], also the basic calling principle is similar to RMI.

Methods with opposite placement must eventually be removed from the environment we are currently looking at. If we just delete the method, all call sites that issue a call to this method must be updated, i.e. a remote call must be generated at every call site. This is an intricate task and not very effective. A better approach is, to keep the original method declaration, but replace its entire body by a remote call. This change is transparent to all call sites and less invasive than the previous approach.

Remote method invocation effectively consists of two parts: generated code to issue a call and runtime support to execute the call. A detailed look at the runtime part is given in section 5.6.

At first, we must generate a `MethodCall` object that encapsulates the call information and the parameters passed to the method stub. The `instanceId` is either the objects instance ID (cf. subsection 4.6.8) or a special ID if the called method is `static`.

```
final Class{*->}[]{*->} types = new Class[] { ArgType1.class, ArgType2, ...
    ↪ };
final Object{*->}[]{*->} args = new Object[] { arg1, arg2, ... };

final bebop.runtime.rmi.MethodCall{*->} mc =
    ↪ bebop.runtime.rmi.MethodCallFactory.createCall("remote.TypeName",
    ↪ "methodName", instanceId, types, args);
```

We pass the generated MethodCall object to a *Bebop* Runtime Library method that passes control to the *Bebop* Runtime Environment while the local site is blocked in its execution.

```
final bebop.runtime.rmi.ReturnValue{*->_} ret =
    ↪ bebop.runtime.Runtime.callRemote(mc);
```

After execution of the method in the remote environment has terminated, we pass control back to the local environment. A method may terminate in a normal way, yielding a return value, or in an exceptional way, yielding an exception. We must issue code to detect if the remote call site threw. Since Java uses checked exceptions, we cannot just throw a `Throwable` without declaring it in the method head. We do not want to alter the method head. Therefore we need some kind of dispatching mechanism to only throw exceptions that are declared. Since we know at compile time which exceptions might be thrown, we can check the received `Throwable` if it is of a certain exception type and issue a cast.

```
if (bebop.runtime.ThrowHelper.shouldThrow(ret)) {
  Throwable{*->_} t = bebop.runtime.ThrowHelper.extractException(ret);

  // For each declared exception we generate:
  if (t instanceof TEx) { throw (TEx) t; }
  ...
}
```

For all arguments passed to the method stub, we must ensure that changes to these values made during execution of the remote method are synchronised back to the local environment. This is important to reflect changes made to mutable objects. If a method parameter is of a *primitive* type, we need no synchronisation.

Primitive types in Java (cf. §4.2 of the Java Language Specification [Gos+14]) are passed *by-value*; their contents are „copied“ during method invocation. Changes made to primitive values are effectively lost when the scope of a method ends, enabling us to omit synchronisation.

Parameters typed with a class in the current compilation unit also do not need synchronisation. Since changes in a class can only be made to its fields, we exploit the property that Normal Environment code does not depend on Trusted Environment fields and vice versa.

Arrays of primitive types must be synchronised. We load the mutated parameter from the `RemoteCallingContext` created during remote execution, see section 5.6. Since JIF does not allow assignment of arrays, see section 2.6, we generate a copy loop to push elements of the remote array to its local counterpart. For primitive types we additionally must issue a call to `BoxHelper` to work around the problem that JIF does not support auto-boxing, e.g. `bebop.runtime.BoxHelper.unboxByteArray(..)`. This mechanism works similar to what is described in section 4.5.8.

```
final T{*->_}[]{*->_} tmp = ret.getCallingContext().load(argIndex);
for (int i=0; i < tmp.length; ++i) {
  arg[i] = tmp[i];
}
```

A parameter of type `CallingContext`, introduced during method partitioning described in subsection 4.5.8, needs special treatment. If we encounter such a parameter, we *assimilate* its state to the current `CallingContext`. This allows to reflect changes in control flow—i.e. `continue` and `break`—back to the local site. `__ctx` is the `CallingContext` of the current (local) method.

```
__ctx.assimilate((bebop.runtime.CallingContext)
    ↪ ret.getCallingContext().load(argIndex));
```

If a parameter is of a reference type not in the current compilation unit or it is an array of non-primitive types, we are currently unable to provide a synchronisation. This is a known limitation of the *Bebop* Compiler and may be addressed in future work.

If the method declaration defines a non-void return type, we must emit a `return` statement that extracts the return value from the retrieved `ReturnValue`:

```
return (TReturn) ret.getReturnValue();
```

4.6.4. Add `main` Method to Normal Partial Application

In this pass we add a `main` method to the normal environment entry point class.

Java needs a `main` method to start an application. Additionally, we must initialise the normal environment container in the `main` method, before we execute the application's `EntryPoint.run` method.

The entry point of a *Bebop* application is imposed by the *Bebop* Runtime Library interface `EntryPoint` and its `run` method. The main method is added to the class that implements the `EntryPoint` interface. The responsibility of the main method is to bootstrap the container and inform the container that the entry point class must be instantiated and execution of the normal environment partial program must begin. Bootstrapping of the container is described in detail in section 5.7. The code added to the entry point class is:

```
public static void main(String[] args) {
  bebop.runtime.NwContainer.bootstrap(args, EntryPointNodeClass.class);
}
```

4.6.5. Remove Entry Point from Trusted Partial Application

In this pass we remove the `EntryPoint` interface from the trusted environment.

Since the trusted environment does not define an entry point to start execution, we can safely remove the interface `EntryPoint` from the respective entry point class on the trusted environment.

4.6.6. Rewrite Object Creation

In this pass we change the way objects are instantiated.

Every time an object for a class in the current compilation unit—these are all classes specified for compilation when the *Bebop* Compiler is invoked—is instantiated, we must intercept the instantiation. As described in section 5.5, we need to make sure that activation of an object in one environment leads to activation of a corresponding object in the other environment.

We search for all occurrences of object instantiation with the `new` keyword of classes in the current compilation unit. We change the `new` statement to a *Bebop* Runtime Library call.

```
// Original instantiation:
T t = new T(ctorArg1, ctorArg2, ...);

// Changed instantiation:
Class{*->}[]{*->} types = new Class[] { ArgType1.class, ArgType2.class, ...
    ↪ };
Object{*->}[]{*->} args = new Object[] { ctorArg1, ctorArg2, ... };
T t = (T) bebop.runtime.Runtime.activateLocalObject(class, types, args);
```

JIF treats `NullPointerException`s as checked exceptions, therefore the *Bebop* Runtime Library call cannot be guaranteed to be non-null by JIF. Since we know that this call always returns a non-null reference, we changed the behaviour of JIF to treat the *Bebop* Runtime Library call in a special way.

4.6.7. Remove Opposite Statements from Constructor

In this pass we remove statements with opposite placement from the constructors.

Constructors are treated differently than ordinary methods. In *Bebop*, constructors are not allowed to contain calls to methods, or downgrading expressions or downgrading statements. We impose this restriction to be able to always initialise a class in each execution environment. This allows us to simply delete statements that have opposite placement, if no „forbidden“ statement was found earlier.

Therefore, in a constructor, we either see:

- A *field assignment* that can safely be removed (if the placement is opposite), since fields are already distributed to the partial programs.

- A call to another (superclass) constructor. These calls are already marked to be kept in both partial programs.

4.6.8. Patch Instance ID Fields

In this pass we add an additional instance ID field to each class.

To uniquely identify an object during runtime of the split program, we need an additional object ID. We must be able to uniquely identify objects for remote method invocation, since a call must be invoked on a specific object. A description of the remote method invocation mechanism is given in section 5.6. For this, we introduced a private instance field that holds a globally unique UUID wrapped in the *Bebop*-specific `Id` class. The code inserted is:

```
private bebop.runtime.ID{*->} __bebop_instance_id = bebop.runtime.Id.newId();
```

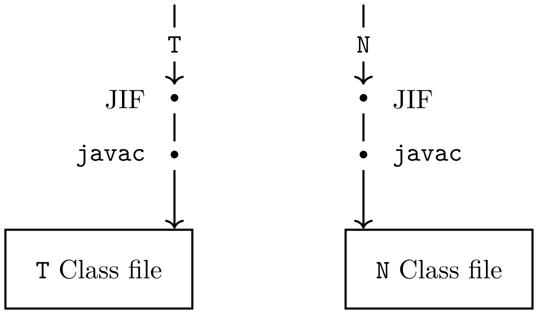


Figure 4.13: Post-processing of the ASTs emitted after Phase Two.

4.7. Post-Processing Phase

After executing all *Bebop* passes, some post-processing steps must be executed. The AST emitted by Phase Two is passed (again) to the JIF Label Checker to make sure information flow is consistent and no policies are violated. If this pass succeeds, we trigger the JIF code generation pass and emit Java code for each AST. The Java code is passed to a standard `javac` compiler by the Polyglot infrastructure.

As a result, we have generated a `.java` and `.class` file for each AST. Since ASTs exist in a Normal Environment and Trusted Environment version for every original input file, the code generation pass emits two `.java` and two `.class` files. The files are spread to different output directories, depending on the environment they are assigned to.

4.8. Limitations and Future Work

We were not able to finish all the work necessary to build a full-fledged partitioning-compiler, because this would exceed the time frame for this thesis. Therefore some restrictions and limitations apply:

- *Integrity properties are neglected.* Despite being defined as a goal, we skipped support for integrity policies. In particular, threshold label calculation does not consider integrity policies, support for the `endorse` statement and expression are missing, and runtime policies must be adapted to reflect integrity concerns.

- *The Bebop Language does not support method calls, and `declassify` statements and `declassify` expressions in constructors.* We made this decision to ease partitioning of constructors as described in subsection 4.6.7.

- *JIF's polymorphic label mechanism is not supported.* JIF supports mechanisms to define *generic* label parameters for classes as described in subsection 2.6.14. We are currently unable to derive placement information from label parameters. This is saved for future work.

- *Method parameters used may be only of primitive types.* Partitioning of standard Java classes requires support for polymorphic labels. The *Bebop* Compiler restricts method parameters to be of primitive type as described in subsection 4.5.8.

- It may be the case that final code for a partial class contains method stubs that are not used at all. Future versions of the compiler shall execut a inter-environment usage analysis and remove methdos that are orphaned.

- The control flow of a program is currently not secured against manipulation. Future work shall add cryptographic methods to ensure that methods cannot be called in arbitrary order by an attacker.

- Recursive methods are currently not supported. This is due to hoisting of fields for reentrancy synchronisation as described in subsection 4.5.5. Support for recursive methods can be established by replacing the synchronisation mechanism by something more sophisticated than hoisting.

4.9. Chapter Summary

We created the *Bebop* Compiler that is able to partition programs into two cooperative parts that run in different execution environments. The major goal defined in chapter 1—building a working compiler—is met, although the *Bebop* Compiler currently imposes restrictions on the input program as described in section 4.8.

When we generate code for the partitioning and code to interact with the *Bebop* Runtime Environment, we emit JIF code that is annotated with security policies. Before the program is compiled to Java code, JIF is used to check the information flow properties. This covers our second major goal: emitted code should not violate security policies.

Partitioning of programs works on a per-statement mechanism, allowing for a fine grained distribution of statements across the two execution environments. This enables us to produce better partitionings in contrast to the trivial solution of putting everything to the trusted environment. This partitioning criterion was defined in our original goals. Although the granularity of the partitioning can surely be improved by focussing on sub-expressions instead of whole statements, the goal can be seen as accomplished.

The original goals defined that we consider both confidentiality and integrity when it comes to partitioning programs. Due to time constraints we neglected the integrity property and focussed only on confidentiality as a first step. Support for integrity properties is left for future work.

CHAPTER 5

THE *BEBOP* RUNTIME

5.1. Introduction

The *Bebop* Compiler described in the previous chapter produces as output two programs that cooperatively rebuild the functionality of the original input program. The program that will run in a trusted environment is called *trusted environment partial program* and the program that will run in a normal environment is called *normal environment partial program.*

The trusted environment partial program cannot be started on its own, since the program entry point is fixed to be in the normal environment. We therefore need a *runtime* that is able to load the trusted environment partial program and handle calls to (and from) the trusted environment. Similarly, the normal environment partial program cannot run on its own because it must interact with the trusted environment partial program. The runtime establishes communication between the two execution environments and is used by both partial programs when a switch to the opposite execution environment is needed. Calls to the runtime are woven directly into the partial programs.

We define the following goals for this chapter:

- *(Objective E)* **Remote Method Invocation** Provide a *runtime* that enables communication between two partial programs. The runtime hosts partial programs, connects the two execution environments and is responsible for dispatching calls to the other execution environment.

- *(Objective F)* **Flexible** The runtime should be lightweight, scalable, extensible, and should easily be portable to different communication interfaces (Sockets, TrustZone-TEE, . . .). The runtime will therefore also be suitable for devices with limited resources.

The *Bebop* Runtime Environment's main responsibility is to provide ways to start partial programs, and to establish communication between the two environments. Additionally, the *Bebop* Runtime Library provides utility methods used by the *Bebop* Compiler during code generation. This targets *(Objective E)*.

In this chapter we describe the architecture of the *Bebop* Runtime Environment and have a detailed look at its components. The concepts of remote object activation and remote method invocation are explained in detail. We show how the *Bebop* Runtime Environment hosts partial programs, and how the execution of a split program works.

5.2. Architecture Overview

Original goals considered when we designed the *Bebop* Runtime Environment were that it should be lightweight, scalable, extensible, and should easily be portable to different communication interfaces. It is preferable to have a lightweight implementation, if the *Bebop* Runtime Environment is executed on environments with limited resources. Also considering the partitioned nature of classes produced by the *Bebop* Compiler—it raises the need of special mechanisms for object identification and call dispatch—we decided to implement a custom runtime instead of building on existing technologies as for example Java RMI. Good portability is important if the *Bebop* Runtime Environment is used in various combinations of environments with different levels of trust. Examples are classical client/server scenarios or the use of the ARM TrustZone. We addressed this issue by having an interchangeable transport layer used for communication, as section 5.4 describes. Finally, extensibility and scalability must be considered when building sustainable software. We based the communication framework of the *Bebop* Runtime Environment on the idea of *message passing* as used in the *actor model*, since we are dealing with a reactive system. As argued by [Lie81], message passing enables good extensibility and modularity and therefore enables us to reach our defined goals. Another reason for message passing is, that original Object Oriented paradigms revolve tightly around the idea of objects sending messages to each other, e.g. for calling procedures, as described in [Hew77]. This strengthens our decision to use the message passing pattern, as one of the main responsibilities of the *Bebop* Runtime Environment is remote method invocation as described in section 5.6. Using the actor model should largely cover what we defined in *(Objective F)*.

During this chapter, we neglect the notion of „trusted environment“ and „normal environment“ and focus on the idea of a *local environment* and a *remote environment. Local* refers to the environment we are currently executing in and *remote* in this context refers to the opposite environment. This is more convenient when describing the *Bebop* Runtime Environment, since it usually does not matter to the runtime if the execution environment is trusted or normal (except where stated explicitly), but it matters if resources are local or remote.

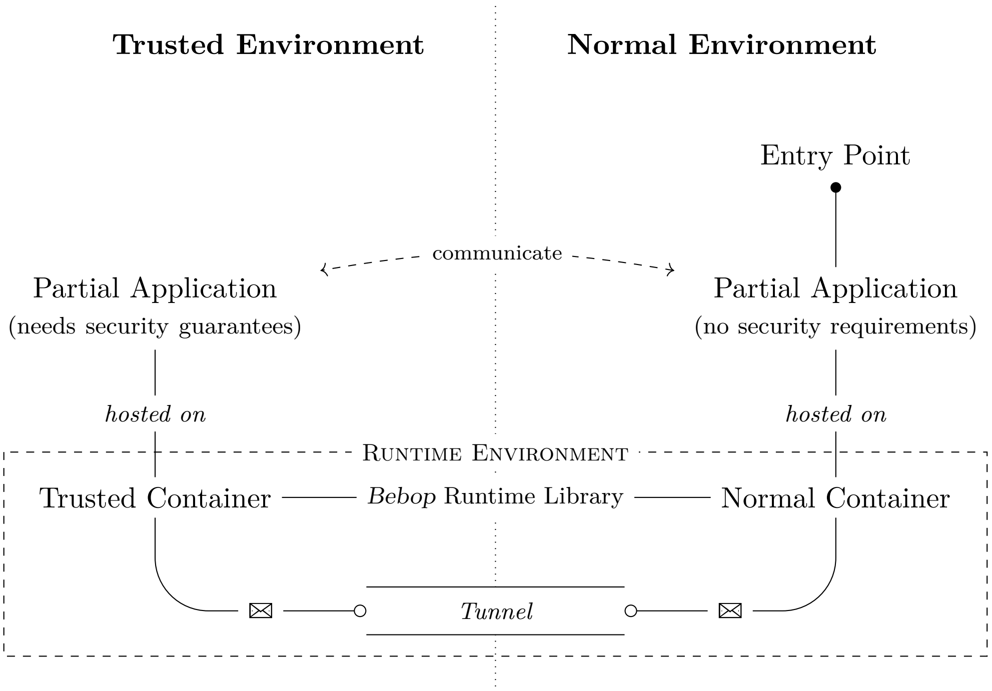


Figure 5.1.: *Bebop* Runtime Environment overview.

5.2.1. The Runtime Environment

Having two partial programs that run simultaneously on two different environments and cooperatively rebuild the functionality of the original program, we must provide an ecosystem that offers the required functionality. The *Bebop Runtime Environment* is the combination of all components necessary to run a split program and is therefore a meta component.

As Figure 5.1 shows, a trusted environment and a normal environment are connected by a (secure) tunnel. All communication between the two environments is bound to the provided tunnel. As described above, partial programs cannot run on their own and are therefore hosted in special *containers* (cf. section 5.3) provided by the *Bebop* Runtime Environment. Partial programs communicate via message passing; the container thus offers a lightweight message passing implementation. Functionality of the *Bebop* Runtime Environment is used by partial programs and the necessary API is provided to the partial programs by the *Bebop* Runtime Library. Single components of the *Bebop* Runtime Environment are described in detail in the following sections.

5.2.2. The Runtime Library

The *Bebop* Runtime Library is a set of methods that expose functionality of the *Bebop* Runtime Environment to a partial program. Calls to the *Bebop* Runtime Library are woven into partial programs during compile time by the *Bebop* Compiler.

The *Bebop* Runtime Library includes the API used during remote object activation and remote method invocation, see section 5.5 and section 5.6 respectively. Additionally, it offers a set of tools used during execution of a partial program:

BoxHelper

The `BoxHelper` is used when extracting array objects transferred in a remote calling context, as described later in this chapter. This is necessary as JIF does not support casting of arrays, see section 2.6.

ThrowHelper

The `ThrowHelper` offers methods to handle exceptions received from the remote environment.

SerializationHelper

This class is responsible for serialisation and deserialisation of method arguments that are to be transported to a remote environment. Arguments are serialised to a `ByteBuffer` using standard Java serialisation, that is, the object must implement the `Serializable` interface—this is true for the primitive types for method calls, that we currently support. A `ByteBuffer` together with information about the type of the object are encapsulated by a `TransportObject` that is used in remote communication.

ClassLoadingHelper

The `ClassLoadingHelper` is the component responsible for loading a Java `Class` object from a given class name. We use Java's `SystemClassLoader` in combination with `Class.forName(..)` to retrieve an appropriate `Class` object.

5.3. Hosting Applications

Loading and executing partial programs is the responsibility of a *container*. The container defines the interaction of several components of the *Bebop* Runtime Environment as shown in Figure 5.2.

The container establishes a connection to a remote container and instantiates an *endpoint* that receives incoming messages, see section 5.4 for details about the endpoint. The container registers *message handlers* at the endpoint that handle incoming messages. Every message type has its own message handler. The partial programis a set of Java class files, present on the file system. Functionality of the partial program is invoked by message handlers, depending on the desired action. The main actions are *remote object activation* as described in section 5.5 and *remote method invocation* as described in section 5.6. The partial program reacts to the messages by executing parts of the split program, eventually producing results that must be passed back to the remote

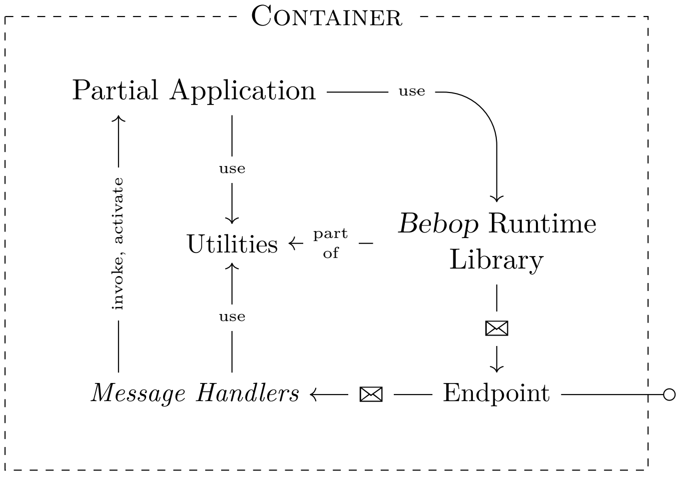


Figure 5.2: A *container* hosts a partial application and is responsible for establishing a connection to a remote container. Additionally, it initialises the *Bebop* Runtime Library and registers message handlers.

environment. For this purpose, it uses methods provided by the *Bebop* Runtime Library and its utility classes; necessary code is woven into the partial program during compile time.

The normal environment container offers additional mechanics that provide an entry point for the split program. The container receives the entry point class of the split program via command line arguments. It loads the class and instantiates it. Afterwards, the control is passed to the entry point in the normal environment partial program, effectively starting the split program.

A detailed look at execution of a split program is given in section 5.7.

5.4. Connecting the Two Worlds

One of the design goals of the *Bebop* Runtime Environment was the interchangeability of the *transport layer* to make the *Bebop* Runtime Environment portable to different systems. To achieve this goal, the *Bebop* Runtime Environment is designed to only rely on an `InputStream` and `OutputStream` for communication with a remote environment. The specific type of the stream does not matter, it may be a network stream, a pipe or a completely different type of communication.

As Figure 5.3 shows, all communication is tunnelled through the provided streams.

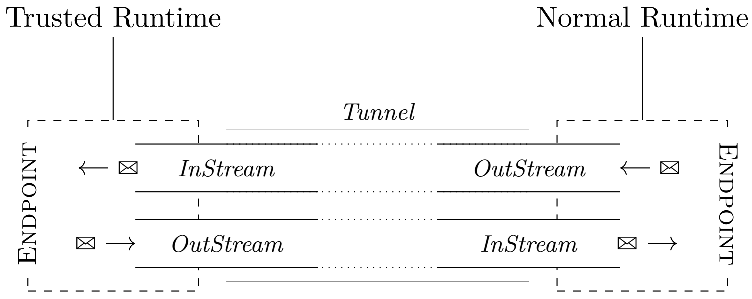


Figure 5.3: The transport layer consists of two stream pairs that connect the endpoints of the two environments.

The sender and receiver on each side of the tunnel are what we call *endpoints*. An endpoint's responsibility is to send messages to a remote environment or to receive messages from a remote environment. We built message processing in the endpoint in a completely asynchronous way. Synchronisation is done via message queues that are processed by tasks. A *task* is a worker method that is executed in a separate thread; tasks are respawned automatically once they have finished their work.

Sending Messages

As Figure 5.4 shows, the endpoint exposes three methods that handle sending of messages. The `send` method takes a message and puts it into the OutQueue. A *Send Task* is waiting for new messages in the queue and relays incoming messages to the *OutStream*.

The `sendAndWaitForReply` method puts a given message into the OutQueue and blocks the calling thread until a response to the given message is received. It polls the ReplyQueue until a reply to the sent message arrives.

The third method, `sendSelf`, takes a message and puts it into the InQueue, effectively mimicking a message received from the remote environment. This is useful if we want to trigger asynchronous actions on the local endpoint.

Receiving messages

Messages coming from the remote environment are coming from an *InStream* and are processed by the *Receive Task* that polls the stream. Upon arrival of a message, the task puts the message into the InQueue. The *Dispatch Task* is polling the InQueue and relays incoming messages to the appropriate *Message Handler*. If the received message is a reply to a previously sent message, the incoming message is put into the ReplyQueue. Message handlers execute the actual action that is linked to a message. Message handlers are registered during initialisation of the endpoint. Depending on the type of the message, execution of the message handler can be either synchronous or asynchronous.

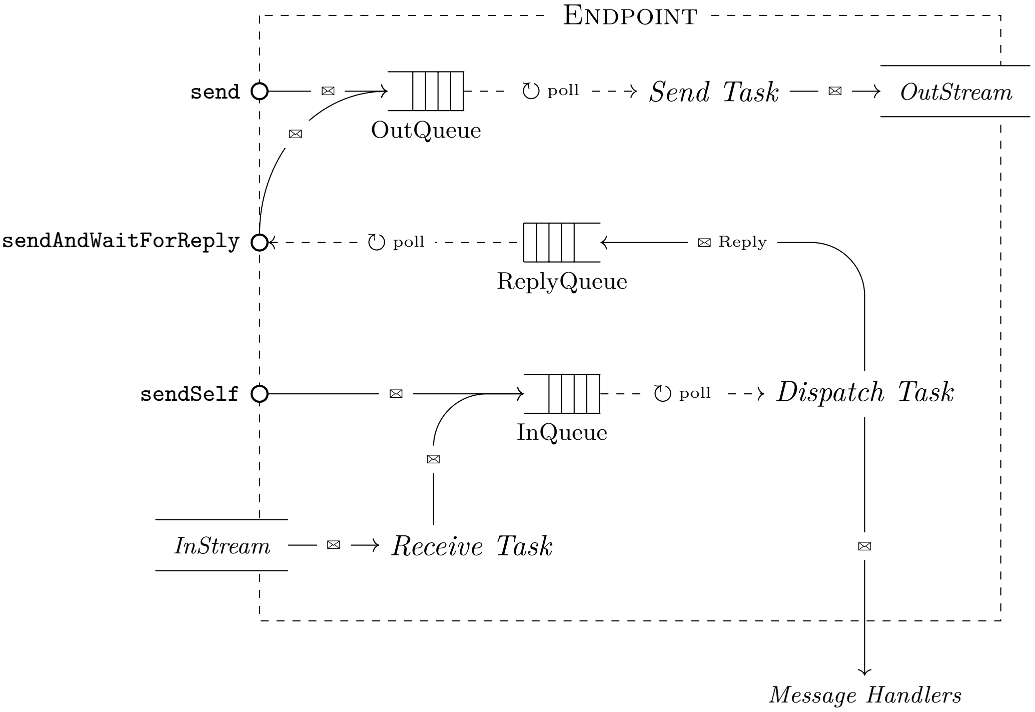


Figure 5.4.: Internals of an *endpoint.* It is responsible for sending messages to the opposite environment and for dispatching actions when messages are received.

5.5. Remote Object Activation

Every time an object is instantiated in the normal environment, we must create a complementing object in the trusted environment, and vice versa. This ensures that an object is present in the remote environment if needed for later calls. We call such objects *opposite objects.*

In the first step we load a Java class from the given class name, with the help of the *Bebop* Runtime Library's `ClassLoadingHelper` utility class, see subsection 5.2.2. We create an instance of the class and invoke the appropriate trusted environment and normal environment constructors on the newly created instances. We call the instantiation and initialisation of an object its *activation.*

As described in subsection 4.6.6, we changed the way objects are created in a program. Every occurrence of „`new X(...)`“ is replaced by a call to a special method of the *Bebop* Runtime Library that is responsible for object creation. This allows us to intercept object creation and to execute our remote activation procedures. Figure 5.5 gives a timeline view of object activation.

Both the trusted and normal object instance must be marked as being related. We introduced the synthetic field `__bebop_instance_id` in both objects during compilation, see subsection 4.6.8. This field holds a UUID to identify an object instance. The ID

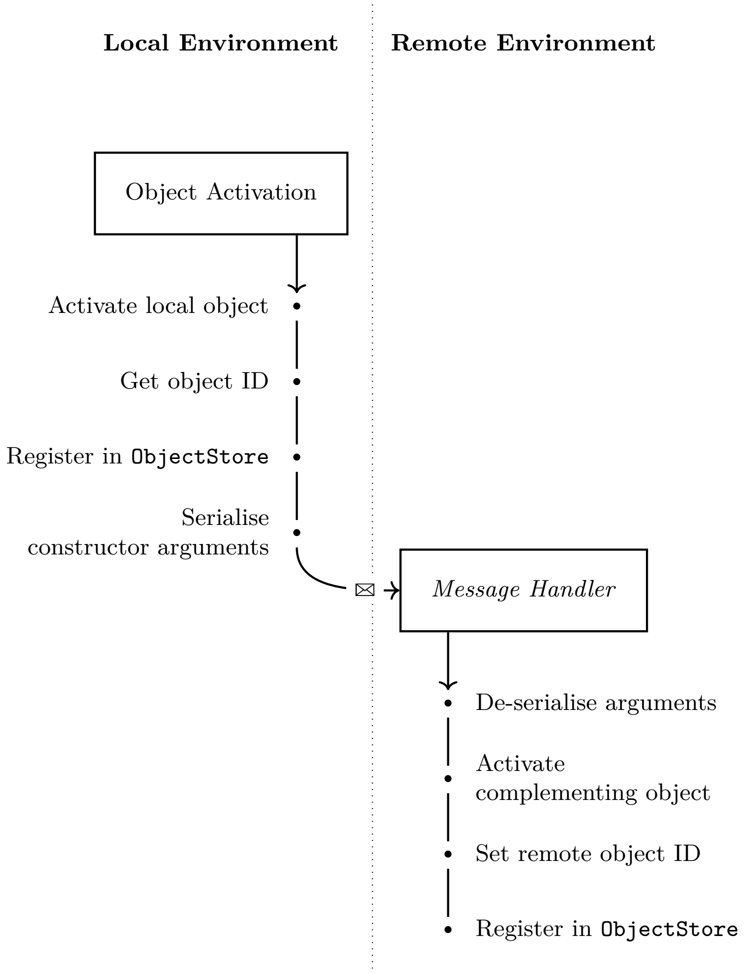


Figure 5.5: Activation and storage of an object in the local environment leads to activation of a complementing object in the remote environment.

is created when the object is activated in one environment and is sent along with the activation message to the other environment. The *Bebop* Runtime Environment makes sure that the ID field is correctly set when an object is instantiated.

Activated objects are stored to an `ObjectStore`. This is a dictionary that allows us to store (and look up) objects by their IDs. This is needed to implement remote method invocation, because a method must be invoked on the correct object—identified by the ID that is sent along the remote invocation message.

Object activation on the remote environment works similar, with two subtle differences. First, all constructor arguments must be de-serialised after being sent in a message. Second, the ID of the object is dictated by the ID defined in the message. The *Bebop* Runtime Environment must explicitly set the ID sent along the message on the new object.

As described in section 5.8, the `ObjectStore` lacks a mechanism to remove unused objects and may thus grow very large. This is a limitation that shall be targeted by future work.

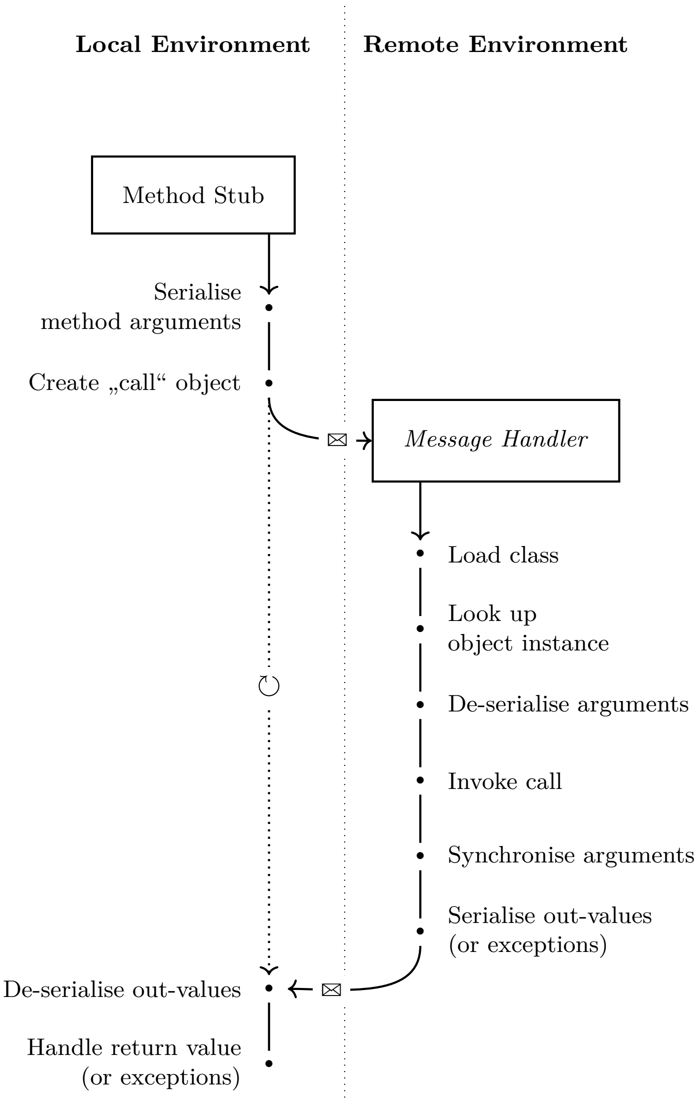


Figure 5.6: Calling a remote method from a local method stub blocks the local execution while the remote environent executes the call. Afterwards, control is passed back to the original call site.

5.6. Remote Method Invocation

Since partitioning a program moves parts of its functionality to a non-local site, we cannot employ ordinary means of method invocation. We must support some form of *remote method invocation* to transfer control to a remote site. As described in subsection 4.6.3, methods with opposite placement are replaced by *stubs* that initiate a remote invocation. Figure 5.6 gives an overview of a remote method call.

Invoking a Remote Method

In a remote method stub, the local call site creates a `MethodCallMessage` that contains all information necessary to dispatch a call on the remote site. In particular this is:

- the name of the opposite class,
- the name of the method to be called,
- the instance ID of the remote object (or a special ID if the call is static),
- a list of type names to identify the correct method if overloads exist, and
- a list of serialised arguments to the method.

Since all arguments must be transferred „over the wire“, all method parameters must be `Serializable`. A helper class of the *Bebop* Runtime Library is used to encapsulate arguments for transport, see subsection 5.2.2.

Transferring Control

After creation of a `MethodCallMessage`, the *Bebop* Runtime Environment on the local environment sends the message to the remote environment. Since method calls must be executed synchronously, the *Bebop* Runtime Environment blocks the program on the local call site until a response message is received. This is accomplished with the help of the Endpoint's method `sendAndWaitForReply`, described in section 5.4.

Dispatching Calls on the Remote Site

When the remote environment receives a `MethodCallMessage`, its *MethodCallHandler* is responsible for processing the message. The handler first loads the target class the call must be invoked on. We use the helper class `ClassLoadingHelper` provided by the *Bebop* Runtime Library, see subsection 5.2.2.

In the next step, we use the instance ID of the message and the `ObjectStore` to load the object instance, we must dispatch the call on. Instances are stored in the `ObjectStore` upon creation, as described in section 5.5. Static method invocations must be distinguished. We use a special „empty“ object ID in the message to mark a call as being static.

We make use of Java Reflection to find the method specified in the message. We invoke the method on the previously retrieved object (or invoke a static call), and catch exceptions that may be thrown by the target method.

After a successful invocation, we store all mutable reference type arguments to a `RemoteCallingContext`. This ensures that reference types that may have been updated during the execution of the method are synchronised back to the original call site. A `MethodReturnMessage` transfers control back to the original call site. It additionally contains the return value of the call and all updated arguments.

On exceptional termination of the method invocation, we must transfer the exceptions back to the original call site. A special `MethodReturnMessage` is sent to the original call site, containing the exception. Reference type arguments are not synchronised in this case, as we make no guarantees if an exception is thrown. This behaviour may cause inconsistencies in the program state. This is clearly not desirable and is part of the known limitations to be solved by future work, see section 5.8.

Return to Local Call Site

The local call site blocks execution and waits until the remote site terminates the call. A message handler on the local site is waiting for the corresponding `MethodReturnMessage` to continue execution. All out-values—that is the return value of the method (if any) or exceptions thrown—are de-serialised and control is passed back to the method stub. The message stub then processes the out-values as described in subsection 4.6.3.

5.7. Executing a Program

The execution of a split program is depicted in Figure 5.7. Each container first initialises the transport layer, where the trusted container is passively waiting for a connection and the normal container is actively connecting. In the reference implementation, the transport layer is using TCP network streams, but this can easily be changed as described in section 5.4.

When a connection has been established, both containers initialise their endpoints; this includes initialisation of the `ObjectStore` as described in section 5.5, and registration of the message handlers that process incoming messages. The most important message handlers that are registered are processing remote method invocation messages, object activation messages and a shutdown message handler that terminates a container if the program exits in the opposite environment. The endpoint runs in a separate thread, after initialisation is done.

Having a running endpoint, we initialise the rest of the *Bebop* Runtime Environment by passing it the endpoint reference. Initialisation of the trusted container is complete after these steps.

The container for the normal environment needs more initialisation. Since the entry point of a program always lies in the normal environment, the normal container is responsible for starting the execution of the split program. We load the provided entry point class and activate it—that is we create an instance and invoke the JIF constructor. The *Bebop* Runtime Environment requires entry point classes to only contain one parameterless constructor. Additionally, the entry point class must implement the `EntryPoint` interface. As described in previous sections, we must store the activated entry point object to the local `ObjectStore` and inform the remote environment that the complementing object must be activated.

Having an activated entry point object, we trigger execution of the program by calling the `run()` method, declared in the `EntryPoint` interface. After execution of the program, both containers are shut down.

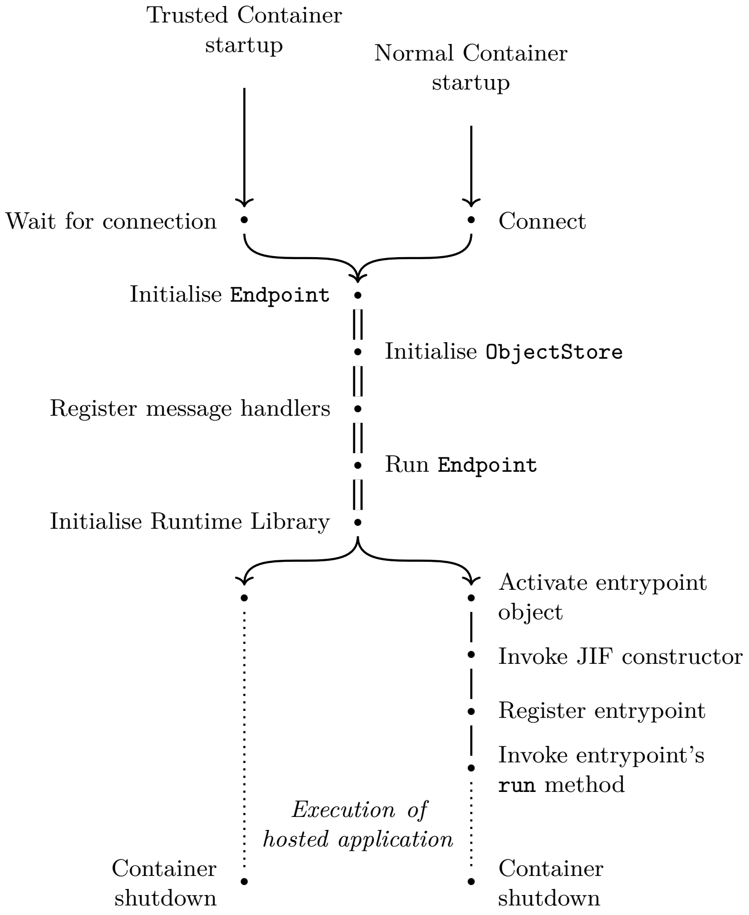


Figure 5.7: Initialisation of the containers that host the partial applications. The input program's entry point is invoked by the normal container.

5.8. Limitations and Future Work

The current implementation of the *Bebop* Runtime Environment contains limitations that were not solved in the time frame of this thesis.

- The `RemoteCallingContext` described in section 5.6 is currently not synchronised correctly if an exception is encountered during execution of a remote method. Data synchronisation code must be added to the *Bebop* Runtime Environment. This will be targeted in future work.

- The `RemoteCallingContext` may grow very large, if many parameters must be synchronised. Future projects should strive to reduce the list of parameters to those that are really necessary.

- The `ObjectStore` currently lacks a mechanism to clear out destroyed (or orphaned) objects. Since the `ObjectStore` always holds a reference to an object even if it is not used any more, the garbage collector will never free the memory. This may lead to unnecessary memory consumption. Mechanisms to remove unused or destroyed objects from the `ObjectStore` could be investigated in future projects.

- There is currently no native TEE support due to lack of a Java port. The current reference implementation uses a TCP stream and two Java instances to simulate the two execution environments. We already prepared the *Bebop* Runtime Environment that a TEE „stream“ can be integrated quickly.

5.9. Chapter Summary

As described in the previous sections, we have built an ecosystem that allows us to run a split program across two different execution environments. The decision to use the actor model (message passing) turned out to be correct: Extending the functionality of the *Bebop* Runtime Environment during its development was straight forward and caused little to no interference with other components; since the *Bebop* Runtime Environment is a reactive system, the natural choice is to use a reactive pattern

The *Bebop* Runtime Environment provides ways to load and execute partial programs; this includes ways that allow communication between partial programs. Required functionality to run a split program—such as remote object activation or remote method invocation—have been implemented and are working.

One weakness of the *Bebop* Runtime Environment is, that it is not very efficient with respect to high-speed performance. Since execution performance was not defined as a goal, we consider this a minor drawback. Due to its modular nature, optimising the performance of single components of the runtime should be easily possible in future projects.

CHAPTER 6

EVALUATION

In this chapter we evaluate our work. We evaluate the developed *Bebop* Compiler and *Bebop* Runtime Environment in the first two sections. Afterwards, we present two sample programs that can be successfully partition with the *Bebop* Compiler.

6.1. Compiler Evaluation

In chapter 4 we defined a set of objectives our compiler should fulfil. The *(Objective A)* described the overall goal of writing a compiler that is able to partition programs, with respect to security policies defined in the input program.

We developed a compiler that is able to produce split programs. We used an extension of the JIF language [Mye99a] as input language and we emit two programs that cooperatively rebuild the functionality defined in the original program. We based our compiler on the Polyglot framework [NCM03]. Since JIF is based on Polyglot and JIF's source code is available, using Polyglot as base for our compiler was the obvious way to go.

In *(Objective B)* we defined that our compiler should not directly emit Java code but should emit the partial programs in JIF code. This enables us to attach security policies to synthetic code that is produced by the compiler. We can thus express security concerns for the code that is generated in order to communicate with the *Bebop* Runtime Environment. This distinguishes our compiler from the similar Swift project [Cho+07], which directly emits Java code. JIF ensures us that the partitioned user code and the generated runtime code adhere to the defined policies.

Our compiler strives for an optimal partitioning with respect to the amount of code placed in one execution environment. In fact, we are able to partition a program on a per-statement level. The placement of a statement depends on the security policies of each sub-expression and is tightly coupled to the information flow policies defined by JIF.

JIF supports two basic types of policies: confidentiality and integrity policies. Although we initially planned to base the program partitioning on both policy types, we focussed only on confidentiality *(Objective D.1)*. Incorporating integrity policies as defined in *(Objective D.2)* is possible, but the objective was dropped due to time constraints.

We present two very general examples of programs that are partitioned by the *Bebop* Compiler in section 6.3 and section 6.4. The examples show, that the goals described above were met. Although the *Bebop* Compiler has limitations (cf. section 4.8), it can be used to generate split programs for real-world scenarios.

6.1.1. On Reusing Swift

When we started this thesis, we investigated if reusing Swift would be an option for us. Swift targets web applications and emits Java code that is destined to run on a web server (trusted environment) and JavaScript code that is destined to run in the browser (normal environment). Much of the data synchronisation is done with the help of Google Web Toolkit (GWT). Since our project does not target the browser, using GWT is not an option for us. Work regarding data synchronisation would have had to be done in any case. Additionally, *Bebop* emits JIF code for the split programs. We therefore can define security policies for the runtime interoperation. Swift directly emits Java code. We would thus have had to change the code generation and probably reimplement parts of Swift's runtime.

We decided to start writing *Bebop* „from scratch“, accepting that the resulting prototype may lack features. This is mainly due to the timely limitations of a master's thesis.

6.2. Runtime Evaluation

The *Bebop* Runtime Environment and the *Bebop* Runtime Library were designed to be lightweight, scalable, extensible and easily portable to different communication interfaces.

6.2.1. Extensibility

As already argued in chapter 5, we achieved extensibility by using a message passing approach. New functionality can easily be added by defining appropriate messages and message handlers. During development, adding (or changing) functionality was painless and caused less to no side effects. This is largely due to the message passing approach, that encourages loose coupling.

6.2.2. Lightweightness

The *Bebop* Runtime Environment and the *Bebop* Runtime Library are relatively small: they consist of only 1,500 lines of Java code (counted with *Cloc* [1]). For comparison, the OpenJDK 7 Update 10 version of RMI consists of roughly 26,000 lines of Java code (counted with Cloc)—excluding most of the CORBA parts. Additionally, only very minimal requirements are imposed on the execution environment, as described in the following paragraphs. We thus consider the ecosystem for *Bebop* programs to be lightweight.

6.2.3. Portability

Portability of the *Bebop* Runtime Environment is provided by the Java language itself, as the Java Runtime Environment supports various computer architectures. One of the objectives for the *Bebop* Runtime Environment was, to be portable to different communication infrastructures. One of the scenarios is the ARM TrustZone case, where communication between the trusted environment and normal environment is one on-chip with the help of special memory areas and special hardware support. Another case would be a trusted server and a normal client that communicate via a network socket. We designed the communication interface in a way that imposes minimum requirements. In fact, the *Bebop* Runtime Environment supports all kinds of communication that can be expressed by Java's `InputStream` and `OutputStream`. The *Bebop* Runtime Environment exposes a `StreamProvider` interface that can easily be used to implement support for various communication interfaces.

The *Bebop* Runtime Environment can easily be adapted to various communication interfaces and implementation scenarios. In our reference implementation used in this thesis, we use a TCP connection and two Java Runtime Environment instances to emulate a trusted environment and normal environment. If we switch to Java's `SSLSockets` for the network streams, the connection is encrypted and *Bebop* can also be used in a setup with two distinct PCs. This would be an implementation of the client-server scenario described above. Support for the ARM TrustZone can be implemented with the help of JNI. The communication with the TrustZone is done by accessing shared memory with a special API. A C program could be used to abstract the memory access. A JNI-wrapper and appropriate Java classes could then expose a pair of streams that can be used with the *Bebop* Runtime Environment. While having the TrustZone scenario in mind during the designing of *Bebop*, we consider support for the ARM TrustZone as future work.

6.2.4. Scalability and Performance

To give statements about the scalability of the *Bebop* Runtime Environment, we analysed the performance of the *Bebop* Runtime Environment and how it reacts to increasing

[1] `http://cloc.sourceforge.net/` visited on Nov 20 2014

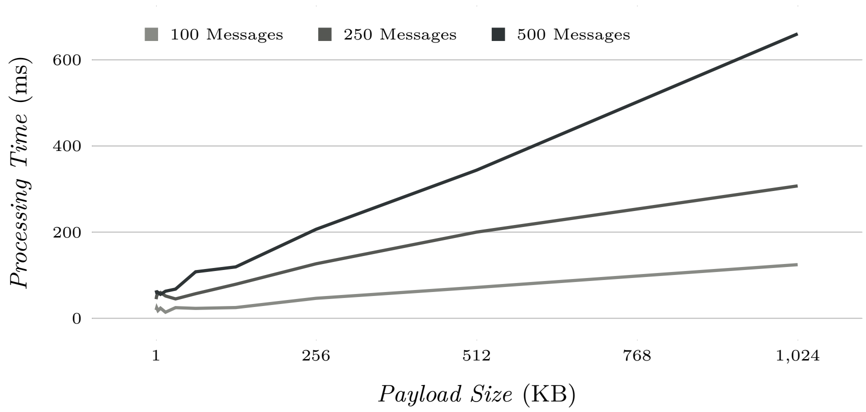


Figure 6.1.: Up to a certain size, the *Bebop* Runtime Environment handles doubling of the payload size in constant time. If message payloads are greater than 32 KB, processing time increases and shows linear behaviour.

amount of messages. All performance measurement were executed on a Intel Core i5 3317 CPU using Oracle's Java 1.8.0 Update 25.

For the first analysis we created messages that carry a random payload with different size. We send messages from one JRE instance to another, mimicking the normal environment and trusted environment. We tested the system with different amounts of messages that must be processed. We used 100, 250 and 500 simultaneous messages. The payload was successively increased, ranging from 1 to 1024 KB. We conducted a total of six measurement runs for each message amount and payload size pair. We dropped the first run to exclude Java type load and JIT-ing times. We then calculated the average of the remaining five runs, to get a realistic processing time. Figure 6.1 shows the results.

For messages with a payload of up to 32 KB, the processing time is almost constant, regardless of the payload. For larger messages, the processing time increases and shows linear behaviour. Doubling the payload effectively doubles the processing time for payload sizes of 128 KB to 1024 KB.

We assume that in real-world use, messages with small payload will be predominant. A typical world-change consists of two messages: the remote invocation—that must transport the method arguments to the remote environment—and the result that must be returned. We analysed how the amount of simultaneous messages with a fixed 4 KB payload influences the processing time. Figure 6.2 shows the results. We chose the 4 KB payload because small messages up to 32 KB can be processed in quasi-constant time as shown in Figure 6.1. We also assume that method arguments or return values that must be transported do usually not exceed this limit, because we are currently limited to primitive types only.

The *Bebop* Runtime Environment handles the increasing amount very well. The chart shows that increasing the amount of messages by a factor of 100 increases the processing times only by a factor of roughly 20.

Our analysis shows, that the *Bebop* Runtime Environment performs well and shows

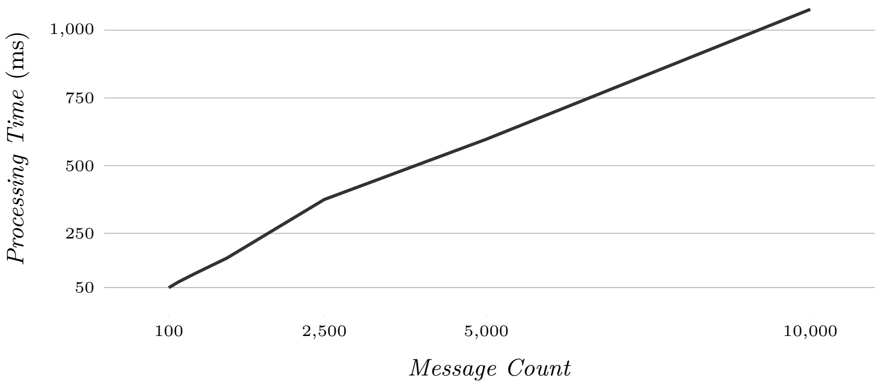


Figure 6.2.: The processing performance of messages with a random 4 KB payload. Increasing the amount of messages that must be processed has a linear impact on the processing time.

a linear behaviour in processing time versus message load. We used payload sizes and messages that we think are realistic for larger programs. For comparison, in the example presented in section 6.3, the trusted environment sends 3 messages to the normal environment. In return, it receives 10 messages from the normal environment. This includes debug messages.

6.3. Example: Time-Based One-Time Password Derivation

6.3.1. Motivation

Deriving one-time passwords or one-time keys from a secret is a common scenario in cryptography [Che08]. In this example we use the popular TOTP algorithm [MRa+11] to derive a one-time password. At the time of writing, Wikipedia lists[2] about sixty major websites/corporations supporting this algorithm, featuring big players like Google, Facebook, Microsoft, etc.

6.3.2. The Program

We designed a program to derive a one-time password with the TOTP algorithm, printed in Listing 6.1. The *Bebop* TEE Toolbox offers a cryptographic primitive that encapsulates the algorithm. The algorithm is only part of the trusted-version of the Toolbox and produces a one-time password with the policy $\langle$`trusted` → `trusted`$\rangle$. We need to pass it the correct parameters for key derivation—we chose to select parameters that are compatible with Google Authenticator [Van14]. One-time passwords derived with our implementation are thus compatible with the login mechanisms of e.g. Gmail [Goo14] or Github [Git13].

[2] `http://en.wikipedia.org/w/index.php?title=Google_Authenticator&oldid=634678467`

Our program first asks the user to enter their name and PIN in the trusted environment to authenticate to the system. This is achieved by acquiring a `TrustedToolbox` and calling its `read` method. Having ARM TrustZone devices in mind, these are able to handle keyboard input in a special way that is directly passed to the trusted environment. Asking the user to enter their credentials via this secure channel makes sure that the credentials are not intercepted. If the PIN was correct, the shared secret is read from trusted storage in the trusted environment. The program then calculates a one-time password and prints it to the user on the normal environment. One-time passwords are typically part of a two-factor authentication and only valid for a certain period of time. We do not consider printing of the one-time password to the normal environment a security issue here. Nevertheless, information flow policies force us to issue a declassification of the generated one-time password, before it can be printed to the normal environment. This is due to information flow from the shared secret—that has a strict policy—to the one-time token.

Listing 6.1: A TOTP example written for *Bebop*.

```
package example;
import bebop.tee.TeeToolboxFactory;
public class TOTP implements bebop.runtime.EntryPoint {
  public void run{trusted->_}() : {trusted->_} where caller (trusted) {
    // TOTP settings compatible to Google Authenticator
    long interval = 30;
    long startTime = 0;
    int{trusted->_} digits = 6;
    String{trusted->_} hash = "HmacSHA1";
    // Compute TOTP time slice
    long now = TeeToolboxFactory.getNormalToolbox().getUnixTimestamp();
    long time = now - startTime;
    try {
      if (time >= 0) {
        time = time / interval;
      } else {
        time = (time - (interval - 1)) / interval;
      }
    } catch (ArithmeticException{trusted->} ae) {}
    TeeToolboxFactory.getTrustedToolbox().println("Please enter your user name.");
    String user = TeeToolboxFactory.getTrustedToolbox().read();
    TeeToolboxFactory.getTrustedToolbox().println("Please enter your PIN code.");
    String pin = TeeToolboxFactory.getTrustedToolbox().read();
    String referencePin = TeeToolboxFactory.getTrustedToolbox().readPin(user);
    try {
      if (!pin.equals(referencePin)) {return;}
    } catch (NullPointerException{trusted->} npe) {return;}
    byte{trusted->}[] key = TeeToolboxFactory.getTrustedToolbox().readSharedSecret(user);
    // Calculate TOTP token
    String otp = TeeToolboxFactory.getTrustedToolbox().generateTotp(key, time, digits, hash);
    declassify ({trusted->} to {trusted->_}) {
      String declOtp = declassify(otp, {trusted->} to {trusted->_});
      TeeToolboxFactory.getNormalToolbox().println("Your one-time password is: " + declOtp);
    }
  }
}
```

The TOTP algorithm uses the current time with a precision of 30 seconds as an input. We are reading the system time from the normal environment. This is no limitation for the use of the TOTP algorithm. Since derived tokens are used for two-step verification during a log-in process, manipulating the time just increases the chance that a wrong token, which is not accepted by the log-in server, is generated.

6.3.3. Partitioning

The TOTP algorithm depends on a shared secret. This secret *must* be kept secure. By reading it from trusted storage, the *Bebop* Compiler will place the read on the trusted environment. Authentication of the user to the system is placed on the trusted environment since credentials are read from trusted I/O. The one-time token is derived from a variable with a strict policy and thus will also be placed on the trusted environment. The public parameters for the algorithm do not need special policies and are placed on the normal environment. The result of the declassification carries a weak policy and will thus be placed on the normal environment.

The *Bebop* Compiler is invoked for the input file:

```
java polyglot.main.Main
  -extclass bebop.BebopExtensionInfo
  -D bebop-1.0.0/bebop_out
  -d bebop-1.0.0/bebop_out
  -sigcp jif-3.4.1/sig-classes:bebop-1.0.0/sig-classes
  -sourcepath jif-3.4.1/lib-src
  -classpath jif-3.4.1/rt-classes:bebop-runtime-1.0.0/classes
  -noserial -simpleoutput -fqcn -globalsolve -e
    applications/TOTP/TOTP.jif
```

The *Bebop* Compiler produces two executable Java `.class` files along with the accompanying Java source code files. The sources for the trusted part are printed in Listing A.1, the sources for the normal part are printed in Listing A.2.

Execution of the split program always begins in the normal environment. As described in chapter 5, execution starts in the added `main` method that bootstraps the *Bebop* Runtime Environment. Figure 6.3 gives a graphical overview of the control flow.

After bootstrapping, the *Bebop* Runtime Environment passes control to the `run` method of the class. The original `run` method has been partitioned into three parts. The first part contains the calculation of the TOTP time slice. The user authentication and derivation of the TOTP token was moved out to a separate method named `__m_run_rigyjj`. The method `__m_run_rigyjj` in the normal environment does not contain the code to derive the TOTP token. Instead, this method is the local *method stub* that initiates control transfer to the trusted environment.

After the token was derived, we print the received TOTP token to the standard out of the normal environment. This happens in the method `__m_run_ov0was` that is executed in the normal environment.

The trusted environment partial application is effectively entered via the method `__m_run_rigyjj`, when the normal environment passes control in its `__m_run_rigyjj` method stub. This method contains the user authentication and derivation of the TOTP token. Parameters for the TOTP algorithm are passed to the trusted environment in a `CallingContext` because their labels in the original program were $\langle$`trusted` $\rightarrow \perp\rangle$ and the *Bebop* Compiler placed the variables in the normal environment.

After derivation of the one-time password, the result is declassified and stored to a `CallingContext` for transport to the normal environment.

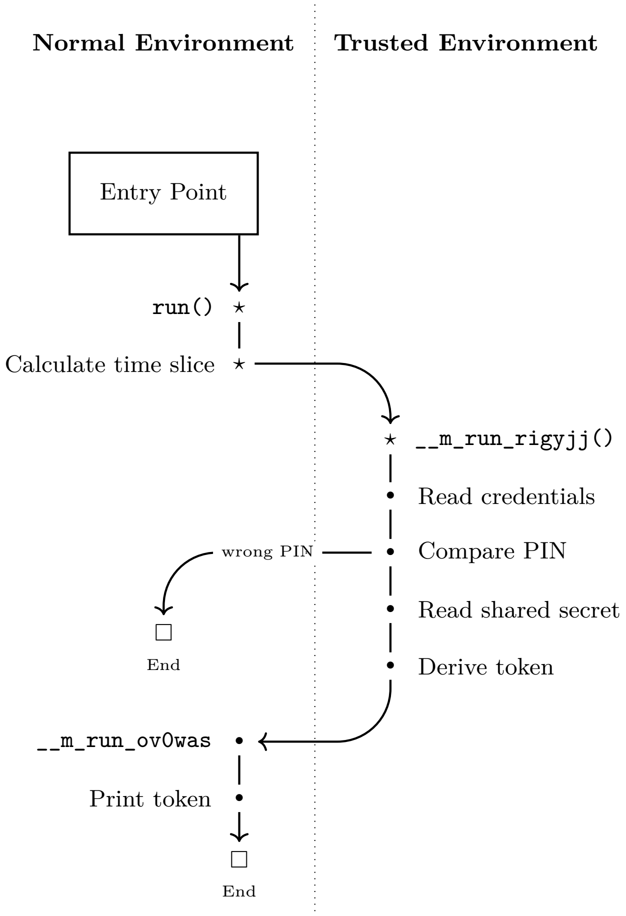


Figure 6.3: The *Bebop* Compiler moved derivation of the one-time token completely to the trusted environment. This adheres to the confidentiality policies stated in Listing 6.1.

The run method of the trusted environment partial program is the opposite method stub to the run method of the normal environment. It is actually not used, because it is not called from the normal environment and execution of a split program does not start in the trusted environment. This method stub has no use and may safely be removed; this is planned for future releases of the *Bebop* Compiler.

6.4. Example: A Password Guessing Game

6.4.1. Motivation

In this second example we implemented a simple log-in mechanism. Comparing a password entered by a user to a stored password is a very common task. Ideally, we want to be able to make sure, that this process cannot be tampered with. By partitioning the log-in program into two parts with the critical statements running in a trusted environment like the ARM TrustZone, manipulation of the log-in process can be mitigated. Usually, a password is not stored in plaintext but the hash of a password is stored. For this example we omitted hashing for the sake of simplicity and compare passwords directly.

6.4.2. The Program

Listing 6.2 shows the log-in program. We defined the number of tries and the reference password with high-confidentiality policies. *Note*: for the sake of simplicity, we stored the password in plain text. A real-world implementation should *never* store a password in plain text. One possibility is, to use a key derivation function like PBKDF2 [Kal00] to derive a password hash. Future work may extend the *Bebop* Runtime Environment to expose more cryptographic primitives.

We read the user password via a trusted input channel and execute a password comparison. Data flow policies ensure that statements responsible for the comparison „inherit“ the security policy of the user password read from the trusted input channel.

The final result of the comparison is printed to the user on the normal environment. A declassification is necessary, to express we want to „leak“ the information if the password was correct on purpose.

Listing 6.2: A password compare example written for *Bebop*.

```
package iaik;

import bebop.tee.TeeToolboxFactory;

public class PwdCmp implements bebop.runtime.EntryPoint {
  private String{trusted->} password = "my_secret";
  private int{trusted->} triesLeft = 3;

  public void run{trusted->}() : {trusted->_} where caller (trusted) {

    while (triesLeft > 0) {
      String{trusted->} p = TeeToolboxFactory.getTrustedToolbox().read();
      triesLeft--;

      if (compare(p)) {
        declassify ({trusted->} to {trusted->_}) {
          TeeToolboxFactory.getNormalToolbox().println("Your guess was correct.");
        }
        return;
      }

      declassify ({trusted->} to {trusted->_}) {
        TeeToolboxFactory.getNormalToolbox().println("Wrong password. Try again.");
      }
    }

    declassify ({trusted->} to {trusted->_}) {
      TeeToolboxFactory.getNormalToolbox().println("Out of tries. Good bye.");
    }
  }

  private boolean{trusted->_} compare{trusted->}(String{trusted->} p) : {trusted->_} where caller
       ↪ (trusted) {
    try {
      if (p.equals(password)) {
        declassify ({trusted->_}) {
          return true;
        }
      }
    } catch (NullPointerException{trusted->} npe) {
      declassify ({trusted->_}) {
        return false;
      }
    }

    declassify ({trusted->_}) {
      return false;
    }
  }
}
```

6.4.3. Partitioning

The *Bebop* Compiler is invoked for the input file:

```
java polyglot.main.Main
  -extclass bebop.BebopExtensionInfo
  -D bebop-1.0.0/bebop_out
  -d bebop-1.0.0/bebop_out
  -sigcp jif-3.4.1/sig-classes:bebop-1.0.0/sig-classes
  -sourcepath jif-3.4.1/lib-src
  -classpath jif-3.4.1/rt-classes:bebop-runtime-1.0.0/classes
  -noserial -simpleoutput -fqcn -globalsolve -e
    applications/PwdCmp/PwdCmp.jif
```

The *Bebop* Compiler produces two executable Java `.class` files along with the accompanying Java source code files. The sources for the trusted part are printed in Listing A.3, the sources for the normal part are printed in Listing A.4.

Execution of the program starts in the `run` method of the normal environment. Control is immediately passed to the trusted environment where the password is read from a trusted input channel. The try-counter—that only exists in the trusted environment—is decremented and the password is compared. Control is passed to either `__m_run_igt121()` or `__m_run_ti0zoc()` on the normal environment and leads to an output informing the user if the password was correct. If the password was correct, the program terminates. If the password was wrong, the program on the trusted environment loops, if tries are left. After all tries are used up, control is passed to the normal environment and the user is informed that no tries are left. Afterwards the program terminates.

The *Bebop* Compiler moved all statements affected by high-confidentiality policies to the trusted environment. Only output that is explicitly declassified has weaker confidentiality policies and is placed on the normal environment.

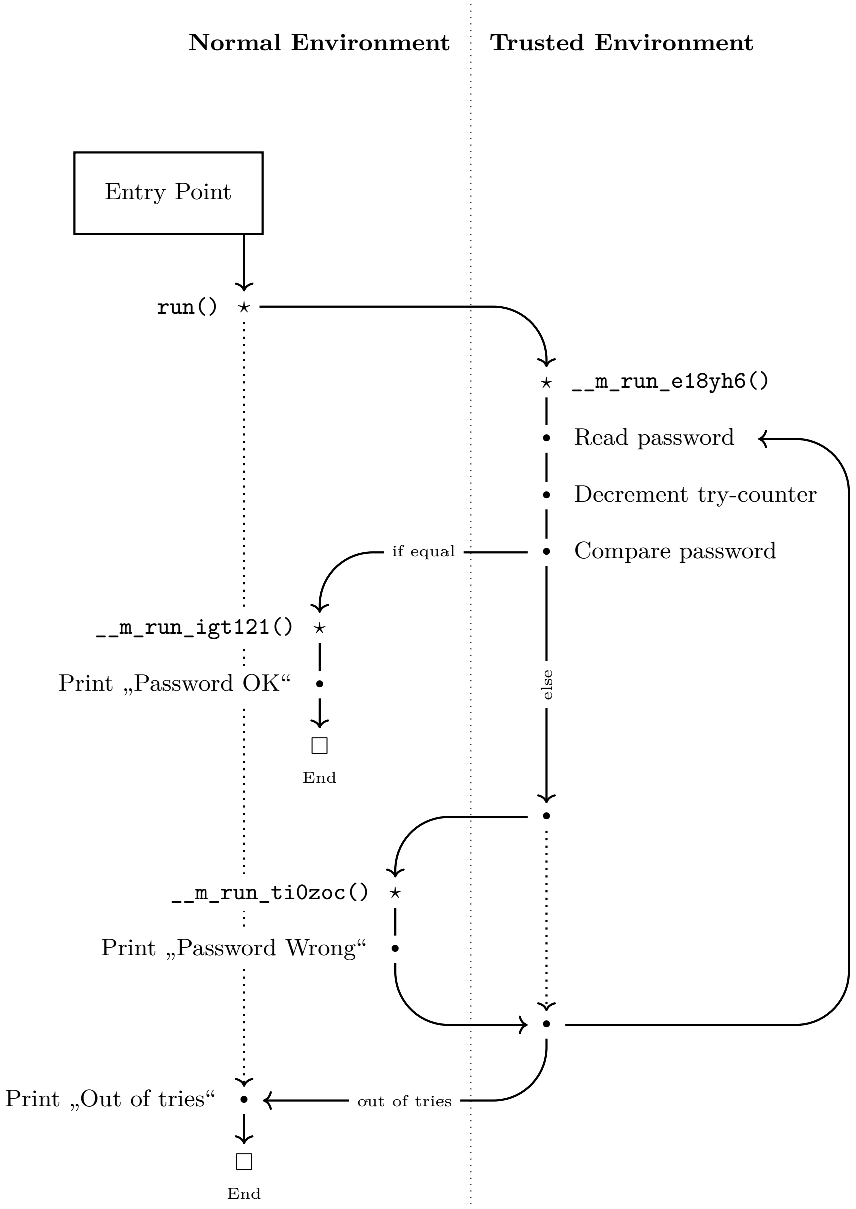


Figure 6.4.: The *Bebop* Compiler moved password comparison and handling of the try-counter to the trusted environment. The results are declassified and lead to textual output on the normal environment, that informs the user if the password was correct.

6.5. Future Evaluations

We evaluated two of the most prominent usage scenarios of program partitioning in the examples above. Clearly, many other examples exist that could be evaluated.

Since we have a working partitioning for the two examples presented above, many other applications qualify as candidates for partitioning. We propose the following scenarios for evaluation in future work. We expect the examples to be partitionable with *Bebop*, because the basic mechanism is similar to the evaluated examples: sending data to the trusted world, processing it and (optionally) returning a derived result.

6.5.1. Trusted Maintenance Log

This scenario reflects the idea that devices sold by a manufacturer should be able to write maintenance logs in a secure way. The log data should be stored in the trusted environment. In normal operation mode, the application on the device generates log files and statistics necessary for device maintenance. These logs are sent to the trusted environment and stored in a secure way. The device is not able to read the logs in normal operation mode.

In maintenance mode, a service technician should be able to read out the accumulated log files. The technician must therefore authenticate against the trusted environment.

6.5.2. Trusted Model Branding

Many hardware devices undergo „branding“ during the first-time operation. The branding decides the feature set of a device. The feature set configuration set by the manufacturer must be protected against malicious changes by device operators.

This scenario involves secure storage of the device configuration in the trusted environment. The normal environment should be able to read the configuration, but should not be able to alter it.

A special upgrade scenario should be considered: a service technician should be able to change the branding of a device, if a customer paid for it. This involves secure authentication of the technician.

6.5.3. Electronic Signature

This scenario is an equivalent to signing documents with a digital signature. The private key should be stored on the trusted environment only. Data is „uploaded“ to the trusted environment and signed with the private key. Before data is signed, the user should be required to enter a PIN code through a trusted channel.

6.5.4. Password Safe

A password safe stores (and generates) secure passwords. The user opens the safe with a master key that should be read from a trusted channel. The user can then generate or retrieve passwords for a specific service. The password safe should print the password to a trusted display device on the trusted environment.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1. Conclusion

We explored partitioning of programs with respect to security policies. We assume that we have a system with two execution environments where one environment has special security guarantees and is regarded as *trusted*. An example for a trusted environment is the ANDIX OS [Fit14]. Partitioning a program into two parts and moving critical code to a trusted environment enables us to run critical code with higher security guarantees. *Bebop* is similar to projects like Swift [Cho+07], but Swift has a focus on web development.

We developed the *Bebop* Compiler to automatically generate split program that can be executed in two distinct execution environments. *Bebop* respects security policies that are annotated in the input source code. We generate two output programs where one program targets a trusted environment and one program targets a normal environment. We used the developed compiler to successfully partition programs, as shown in chapter 6.

To execute split program, we developed the accompanying *Bebop* Runtime Environment. The *Bebop* Runtime Environment is used to bootstrap and execute each split program. Additionally, the *Bebop* Runtime Environment provides the mechanisms used to invoke remote methods, and to synchronise objects and data across two execution environments. Calls to the *Bebop* Runtime Library are woven into the split programs during the compilation process.

7.1.1. Retrospective

If we were to start the development of *Bebop* again from the beginning, some of our decisions would be different. A system that employs a two world partitioning essentially needs a mechanism to classify data (and its flow) as *critical* or *non-critical.* Critical in this sense can either mean *of high confidentiality* or *needs high integrity guarantees.* Using a type system and typing rules—as proposed by Volpano [VIS96] and used in JIF [Mye99a]—is a very natural way to express the security concerns. However, JIF itself may be too „powerful“ if the task is to partition a program. The principal hierarchies and complex label lattices that follow could be replaced by a much simpler high-low model, as used by Denning [Den76] in her original work on information flow. Using a high-low model in combination with type inference for security enhanced types may allow for writing programs in a more straightforward manner than it is with JIF.

Regarding memory aliasing, we think that functional languages may be a better choice for the input language, since such languages usually only allow assigning to a variable once. Especially *purely functional* languages like Haskell offer ways of writing side effect free code. We think that implicit information flow for such a language may in many cases be explicitly visible in code due to single assign, if-expressions and similar features. We think that studying these properties in combination with a security enhanced type system may lead to interesting results.

Future projects might also consider changing the granularity of the program partitioning. The *Bebop* Compiler determines a program partitioning on a per-statement basis. It may be easier to partition on a per-method basis. This would move parts of the partitioning process back to the developer who would be required to write methods that can be moved as a whole.

In general we think there are two main directions for future work on program partitioning:

- Formally correct partitioning by adhering to security policies and by strictly tracking information flow in a program. We claim that this leads to unintuitive programs that are hard to write, but can be proven to be secure, and can be partitioned in a secure way. Especially *label creep* makes secure programs hard to write.

- Convenient partitioning that allows for input programs that are easy to write but can not be proven to be secure. This could be achieved by not tracking information flow but looking at data flow in a program. Also, the compiler could issue warnings instead of failing compilation. This lets the developer write code in a way they are used to, but raises awareness for potential information leaks. The developer could then tweak only important parts of the code. Such a system could for example employ a per-method partitioning.

7.2. Future Work

Clearly, developing a compiler is a tall order. For our thesis we were not able to implement all features we would like to have implemented. Some of the implemented features show room for improvement.

7.2.1. Extending the *Bebop* Compiler

The *Bebop* Compiler is working, but limitations apply. The most prominent limitation currently is the lack of support for integrity policies. When determining a program partitioning, we currently only consider confidentiality policies. Integrity policies are the dual [Bib77] security properties that should be considered when partitioning a program. Support of this feature includes calculation of a threshold label for integrity policies, considering the label when calculating a partitioning, support for downgrading mechanisms like the `endorse`-statement and `endorse`-expression, and redefinition of the runtime policies to also include integrity. Although we initially aimed to also support integrity policies, we moved this feature to future work.

Currently, we can only derive the placement of a statement if no dynamic or polymorphic labels are used. JIF supports label parameters for classes that allow for a generic definition of the labels used in a class. The *Bebop* Compiler currently does not support parametrised classes. Future projects may investigate mechanisms to derive static placements from label parameters. This will enable the *Bebop* Compiler to support more object oriented features of the input language and thus removes restrictions that apply for method parameters.

A detailed list of limitations of the *Bebop* Compiler is already given in section 4.8. This includes restrictions regarding the type of method parameters, restrictions for statements used in constructors, and optimisations in code generation.

7.2.2. Extending the *Bebop* Runtime Environment

The *Bebop* Runtime Environment works well and there are no urgent features that must be implemented. Although, one drawback of the *Bebop* Runtime Environment might be investigated by future projects: Every time objects are created, the remote environment must also create a corresponding object. All created objects are stored in a dictionary, together with an ID to support lookup of objects when it comes to dispatching a remote call. Usually, objects that go out of scope or are no longer reachable are collected by the garbage collector. Since the object dictionary always holds a reference to the object, it can no longer be collected. Additionally, the corresponding remote object must be destroyed as well.

Currently, there is no mechanism to remove objects from the dictionary, and there is no mechanism to destroy the remote object. This may lead to massive memory consumption. Future projects should investigate mechanisms to solve this drawback.

Additional minor drawbacks and topics of future interest exist for the *Bebop* Runtime Environment. A detailed list of limitations that shall be addressed in future work is already given in section 5.8. This includes optimisations for remote data synchronisation and handling of exceptions during execution of remote methods.

APPENDIX A

SOURCE CODE LISTINGS

A.1. Example: Time-Based One-Time Password Derivation

The original input source code for this example is shown in Listing 6.1 in section 6.3. This section shows the Java code that is emitted after the post-processing phase of the *Bebop* Compiler.

Listing A.1: Trusted environment partial program (Java code) for the TOTP example of Listing 6.1.

```
package example;
public class TOTP_T {
  public void run() {
    { final java.lang.Class[] types = new java.lang.Class[] {  };
      final java.lang.Object[] args = new java.lang.Object[] {  };
      final bebop.runtime.rmi.MethodCall mc = bebop.runtime.rmi.MethodCallFactory.createCall(
          "example.TOTP_N", "run", this.__bebop_instance_id, types, args);
      final bebop.runtime.rmi.ReturnValue ret = bebop.runtime.Runtime.callRemote(mc); }
  }
  public example.TOTP_T example$TOTP_T$() {
    this.jif$init();
    {}
    return this;
  }
  private void __m_run_ov0was(bebop.runtime.CallingContext __ctx) throws java.lang.ClassCastException {
    { final java.lang.Class[] types = new java.lang.Class[] { bebop.runtime.CallingContext.class };
      final java.lang.Object[] args = new java.lang.Object[] { bebop.runtime.BoxHelper.box(__ctx) };
      final bebop.runtime.rmi.MethodCall mc = bebop.runtime.rmi.MethodCallFactory.createCall(
          "example.TOTP_N", "__m_run_ov0was", this.__bebop_instance_id, types, args);
      final bebop.runtime.rmi.ReturnValue ret = bebop.runtime.Runtime.callRemote(mc);
      if (bebop.runtime.ThrowHelper.shouldThrow(ret)) {
        java.lang.Throwable t = bebop.runtime.ThrowHelper.extractException(ret);
        if (t instanceof java.lang.ClassCastException) {
          throw (java.lang.ClassCastException) t; }
      }
      __ctx.assimilate((bebop.runtime.CallingContext) ret.getCallingContext().load("0")); }
  }
  private void __m_run_rigyjj(bebop.runtime.CallingContext __ctx)
    throws java.lang.NullPointerException, java.lang.ClassCastException {
    java.lang.String hash_4a32pd;
    { hash_4a32pd = (java.lang.String) __ctx.load("hash_4a32pd"); }
    long time_icqzpb;
    { time_icqzpb = ((java.lang.Long) __ctx.load("time_icqzpb")).longValue(); }
    int digits_h0l4de;
    { digits_h0l4de = ((java.lang.Integer) __ctx.load("digits_h0l4de")).intValue(); }
    bebop.tee.TeeToolboxFactory.getTrustedToolbox().println("Please enter your user name.");
    java.lang.String user_kr0wbb = bebop.tee.TeeToolboxFactory.getTrustedToolbox().read();
    bebop.tee.TeeToolboxFactory.getTrustedToolbox().println("Please enter your PIN code.");
    java.lang.String pin_s6u46d = bebop.tee.TeeToolboxFactory.getTrustedToolbox().read();
    java.lang.String referencePin_bydygh =
        ↪ bebop.tee.TeeToolboxFactory.getTrustedToolbox().readPin(user_kr0wbb);
    try {
      if (!pin_s6u46d.equals(referencePin_bydygh)) { return; }
    } catch (final java.lang.NullPointerException npe) { return; }
    byte[] key_dkhbgo = bebop.tee.TeeToolboxFactory.getTrustedToolbox().readSharedSecret(user_kr0wbb);
    java.lang.String otp_i6i7er = bebop.tee.TeeToolboxFactory.getTrustedToolbox().generateTotp(
        key_dkhbgo, time_icqzpb, digits_h0l4de, hash_4a32pd);
    { java.lang.String declOtp_fqh8fs = otp_i6i7er;
      try {
        final bebop.runtime.CallingContext __ctx_tenuvd = bebop.runtime.CallingContext.create();
        __ctx_tenuvd.store("declOtp_fqh8fs", bebop.runtime.BoxHelper.box(declOtp_fqh8fs));
        this.__m_run_ov0was(__ctx_tenuvd);
      } catch (java.lang.ClassCastException cce_w64sz4) { throw new java.lang.Error(); } }
  }
  private bebop.runtime.Id __bebop_instance_id;
  public TOTP_T() {
    super();
  }
  public void jif$invokeDefConstructor() {
    this.example$TOTP_T$();
  }
  private void jif$init() {
    __bebop_instance_id = bebop.runtime.Id.newId();
  }
}
```

Listing A.2: Normal environment partial program (Java code) for the TOTP example of Listing 6.1.

```
package example;
public class TOTP_N implements bebop.runtime.EntryPoint {
  public void run() {
    long interval_dyuoqu = 30, startTime_dtkysg = 0; int digits_h0l4de = 6;
    java.lang.String hash_4a32pd = "HmacSHA1";
    long now_rtzude = bebop.tee.TeeToolboxFactory.getNormalToolbox().getUnixTimestamp();
    long time_icqzpb = now_rtzude - startTime_dtkysg;
    try {
      if (time_icqzpb >= 0) { time_icqzpb = time_icqzpb / interval_dyuoqu; }
      else { time_icqzpb = (time_icqzpb - (interval_dyuoqu - 1)) / interval_dyuoqu; }
    } catch (final java.lang.ArithmeticException ae) {}
    try {
      final bebop.runtime.CallingContext __ctx_ialvmk = bebop.runtime.CallingContext.create();
      __ctx_ialvmk.store("digits_h0l4de", bebop.runtime.BoxHelper.box(digits_h0l4de));
      __ctx_ialvmk.store("time_icqzpb", bebop.runtime.BoxHelper.box(time_icqzpb));
      __ctx_ialvmk.store("hash_4a32pd", bebop.runtime.BoxHelper.box(hash_4a32pd));
      this.__m_run_rigyjj(__ctx_ialvmk);
      {
        if (__ctx_ialvmk.didReturn()) {return;}
      }
    } catch (java.lang.ClassCastException cce_fcjr11) {throw new java.lang.Error();
    } catch (java.lang.NullPointerException npe_vgl5i9) {throw new java.lang.Error();}
  }
  public example.TOTP_N example$TOTP_N$() {
    this.jif$init();
    {}
    return this;
  }
  private void __m_run_ov0was(bebop.runtime.CallingContext __ctx) throws java.lang.ClassCastException {
    java.lang.String declOtp_fqh8fs;
    {declOtp_fqh8fs = (java.lang.String) __ctx.load("declOtp_fqh8fs");}
    bebop.tee.TeeToolboxFactory.getNormalToolbox().println("Your one-time password is: " +
        ↪ declOtp_fqh8fs);
  }
  private void __m_run_rigyjj(bebop.runtime.CallingContext __ctx)
    throws java.lang.NullPointerException, java.lang.ClassCastException {
    {final java.lang.Class[] types = new java.lang.Class[] { bebop.runtime.CallingContext.class };
      final java.lang.Object[] args = new java.lang.Object[] { bebop.runtime.BoxHelper.box(__ctx) };
      final bebop.runtime.rmi.MethodCall mc = bebop.runtime.rmi.MethodCallFactory.createCall(
          "example.TOTP_T", "__m_run_rigyjj", this.__bebop_instance_id, types, args);
      final bebop.runtime.rmi.ReturnValue ret = bebop.runtime.Runtime.callRemote(mc);
      __ctx.assimilate((bebop.runtime.CallingContext) ret.getCallingContext().load("0")); }
  }
  public static void main(java.lang.String[] args) {
    bebop.runtime.NwContainer.bootstrap(args, example.TOTP_N.class);
  }
  private bebop.runtime.Id __bebop_instance_id;
  public TOTP_N() {
    super();
  }
  public void jif$invokeDefConstructor() {
    this.example$TOTP_N$();
  }
  private void jif$init() {
    __bebop_instance_id = bebop.runtime.Id.newId();
  }
}
```

A.2. Example: A Password Guessing Game

The original input source code for this example is shown in Listing 6.2 in section 6.4. This section shows the Java code that is emitted after the post-processing phase of the *Bebop* Compiler.

Listing A.3: Trusted environment partial program (Java code) for the password compare example of Listing 6.2.

```
package iaik;
public class PwdCmp_T {
  private java.lang.String password;
  private int triesLeft;
  public void run() {
    {
      final java.lang.Class[] types = new java.lang.Class[] {  };
      final java.lang.Object[] args = new java.lang.Object[] {  };
      final bebop.runtime.rmi.MethodCall mc =
           ↪ bebop.runtime.rmi.MethodCallFactory.createCall("iaik.PwdCmp_N", "run",
           ↪ this.__bebop_instance_id, types, args);
      final bebop.runtime.rmi.ReturnValue ret = bebop.runtime.Runtime.callRemote(mc);
    }
  }

  private boolean compare(final java.lang.String p) {
    try {
      if (p.equals(this.password)) {
        return true;
      }
    } catch (final java.lang.NullPointerException npe) {
      return false;
    }
    return false;
  }

  public iaik.PwdCmp_T iaik$PwdCmp_T$() {
    this.jif$init();
    {  }
    return this;
  }

  private void __m_run_igtl2l(bebop.runtime.CallingContext __ctx) throws
       ↪ java.lang.NullPointerException {
    {
      final java.lang.Class[] types = new java.lang.Class[] { bebop.runtime.CallingContext.class };
      final java.lang.Object[] args = new java.lang.Object[] { bebop.runtime.BoxHelper.box(__ctx) };
      final bebop.runtime.rmi.MethodCall mc =
           ↪ bebop.runtime.rmi.MethodCallFactory.createCall("iaik.PwdCmp_N", "__m_run_igtl2l",
           ↪ this.__bebop_instance_id, types, args);
      final bebop.runtime.rmi.ReturnValue ret = bebop.runtime.Runtime.callRemote(mc);
      if (bebop.runtime.ThrowHelper.shouldThrow(ret)) {
        java.lang.Throwable t = bebop.runtime.ThrowHelper.extractException(ret);
        if (t instanceof java.lang.NullPointerException) {
          throw (java.lang.NullPointerException) t;
        }
      }
      __ctx.assimilate((bebop.runtime.CallingContext) ret.getCallingContext().load("0"));
    }
  }

  private void __m_run_ti0zoc(bebop.runtime.CallingContext __ctx) {
    {
      final java.lang.Class[] types = new java.lang.Class[] { bebop.runtime.CallingContext.class };
      final java.lang.Object[] args = new java.lang.Object[] { bebop.runtime.BoxHelper.box(__ctx) };
      final bebop.runtime.rmi.MethodCall mc =
           ↪ bebop.runtime.rmi.MethodCallFactory.createCall("iaik.PwdCmp_N", "__m_run_ti0zoc",
           ↪ this.__bebop_instance_id, types, args);
      final bebop.runtime.rmi.ReturnValue ret = bebop.runtime.Runtime.callRemote(mc);
      __ctx.assimilate((bebop.runtime.CallingContext) ret.getCallingContext().load("0"));
    }
  }

  private void __m_run_e18yh6(bebop.runtime.CallingContext __ctx) {
    while (this.triesLeft > 0) {
      java.lang.String p_r37dqb = bebop.tee.TeeToolboxFactory.getTrustedToolbox().read();
      this.triesLeft--;
      if (this.compare(p_r37dqb)) {
        try {
          final bebop.runtime.CallingContext __ctx_ps3bcb = bebop.runtime.CallingContext.create();
          this.__m_run_igtl2l(__ctx_ps3bcb);
          {
            if (__ctx_ps3bcb.didReturn()) {
              __ctx.setReturn();
              return;
            }
          }
        } catch (java.lang.NullPointerException npe_npik0w) {
          throw new java.lang.Error();
        }
      }
      final bebop.runtime.CallingContext __ctx_1cqzy3 = bebop.runtime.CallingContext.create();
      this.__m_run_ti0zoc(__ctx_1cqzy3);
      {
        if (__ctx_1cqzy3.didBreak()) {
          __ctx_1cqzy3.clearBreak();
          break;
        }
        if (__ctx_1cqzy3.didContinue()) {
          __ctx_1cqzy3.clearContinue();
          continue;
        }
      }
    }
  }

  private bebop.runtime.Id __bebop_instance_id;

  public PwdCmp_T() { super(); }

  public void jif$invokeDefConstructor() { this.iaik$PwdCmp_T$(); }

  private void jif$init() {
   password = "my_secret";
    triesLeft = 3;
    __bebop_instance_id = bebop.runtime.Id.newId();
  }
}
```

Listing A.4: Normal environment partial program (Java code) for the password compare example of Listing 6.2.

```
package iaik;

public class PwdCmp_N implements bebop.runtime.EntryPoint {
  public void run() {
    final bebop.runtime.CallingContext __ctx_vahqk7 = bebop.runtime.CallingContext.create();
    this.__m_run_e18yh6(__ctx_vahqk7);
    {
      if (__ctx_vahqk7.didReturn()) {
        return;
      }
    }
    {
      bebop.tee.TeeToolboxFactory.getNormalToolbox().println("Out of tries. Good bye.");
    }
  }

  private boolean compare(final java.lang.String p) {
    {
      final java.lang.Class[] types = new java.lang.Class[] { java.lang.String.class };
      final java.lang.Object[] args = new java.lang.Object[] { bebop.runtime.BoxHelper.box(p) };
      final bebop.runtime.rmi.MethodCall mc =
              ↪ bebop.runtime.rmi.MethodCallFactory.createCall("iaik.PwdCmp_T", "compare",
              ↪ this.__bebop_instance_id, types, args);
      final bebop.runtime.rmi.ReturnValue ret = bebop.runtime.Runtime.callRemote(mc);
      try {
        return ((java.lang.Boolean) ret.getReturnValue()).booleanValue();
      } catch (java.lang.NullPointerException npe_yhgetb) {
        throw new java.lang.Error();
      }
    }
  }

  public iaik.PwdCmp_N iaik$PwdCmp_N$() {
    this.jif$init();
    {  }
    return this;
  }

  private void __m_run_igtl2l(bebop.runtime.CallingContext __ctx) throws
          ↪ java.lang.NullPointerException {
    {
      bebop.tee.TeeToolboxFactory.getNormalToolbox().println("Your guess was correct.");
    }
    return;
  }

  private void __m_run_ti0zoc(bebop.runtime.CallingContext __ctx) {
    {
      bebop.tee.TeeToolboxFactory.getNormalToolbox().println("Wrong password. Try again.");
    }
  }

  private void __m_run_e18yh6(bebop.runtime.CallingContext __ctx) {
    {
      final java.lang.Class[] types = new java.lang.Class[] { bebop.runtime.CallingContext.class };
      final java.lang.Object[] args = new java.lang.Object[] { bebop.runtime.BoxHelper.box(__ctx)
      };

      final bebop.runtime.rmi.MethodCall mc =
              ↪ bebop.runtime.rmi.MethodCallFactory.createCall("iaik.PwdCmp_T", "__m_run_e18yh6",
              ↪ this.__bebop_instance_id, types, args);
      final bebop.runtime.rmi.ReturnValue ret = bebop.runtime.Runtime.callRemote(mc);
      __ctx.assimilate((bebop.runtime.CallingContext) ret.getCallingContext().load("0"));
    }
  }

  public static void main(java.lang.String[] args) {
    bebop.runtime.NwContainer.bootstrap(args, iaik.PwdCmp_N.class);
  }

  private bebop.runtime.Id __bebop_instance_id;

  public PwdCmp_N() { super(); }

  public void jif$invokeDefConstructor() { this.iaik$PwdCmp_N$(); }

  private void jif$init() { __bebop_instance_id = bebop.runtime.Id.newId(); }
}
```

BIBLIOGRAPHY

[AGH96] Ken Arnold, James Gosling and David Holmes. *The Java programming language*. Vol. 2. Addison-Wesley Reading, 1996.

[Aho+07] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools (2nd Edition)*. Boston, MA, USA: Pearson Education, Inc., 2007. ISBN: 0321491696.

[ARM] ARM Ltd. *TrustZone*. URL: `http://www.arm.com/products/processors/technologies/trustzone/index.php` (visited on 11/09/2014).

[Bib77] Kenneth J Biba. *Integrity considerations for secure computer systems*. Tech. rep. DTIC Document, 1977.

[BL73] D Elliott Bell and Leonard J LaPadula. *Secure computer systems: Mathematical foundations*. Tech. rep. DTIC Document, 1973.

[Bor90] Richard Bornat. *Understanding and Writing Compilers: A do-it-yourself guide*. Macmillan Publishing Co., Inc., 1990. URL: `http://www.eis.mdx.ac.uk/staffpages/r_bornat/books/compiling.pdf`.

[BS04] David Brumley and Dawn Song. „Privtrans: Automatically Partitioning Programs for Privilege Separation“. In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX Association, 2004, pp. 5–5. URL: `http://dl.acm.org/citation.cfm?id=1251375.1251380`.

[Cam+03] Katherine Campbell, Lawrence A Gordon, Martin P Loeb and Lei Zhou. „The economic cost of publicly announced information security breaches: empirical evidence from the stock market“. In: *Journal of Computer Security* 11.3 (2003), pp. 431–448.

[CCM08] Michael R. Clarkson, Stephen Chong and Andrew C. Myers. „Civitas: Toward a Secure Voting System“. In: *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. SP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 354–368. ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.32. URL: http://dx.doi.org/10.1109/SP.2008.32.

[Che08] Lily Chen. „Recommendation for Key Derivation Using Pseudorandom Functions“. In: *NIST Special Publication* SP 800-108 (2008).

[Cho+07] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng and Xin Zheng. „Secure Web Applications via Automatic Partitioning“. In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 31–44. ISSN: 0163-5980. DOI: 10.1145/1323293.1294265. URL: http://doi.acm.org/10.1145/1323293.1294265.

[Cho+09] Stephen Chong, Andrew C. Myers, K. Vikram and Zheng Lantian. *Jif Reference Manual*. Feb. 2009. URL: http://www.cs.cornell.edu/jif/doc/jif-3.3.0/manual.html.

[CM10] Byung-Gon Chun and Petros Maniatis. „Dynamically Partitioning Applications Between Weak Devices and Clouds“. In: *Proceedings of the 1st ACM Workshop on Mobile Cloud Computing & Services: Social Networks and Beyond*. MCS '10. San Francisco, California: ACM, 2010, 7:1–7:5. ISBN: 978-1-4503-0155-8. DOI: 10.1145/1810931.1810938. URL: http://doi.acm.org/10.1145/1810931.1810938.

[CMR04] Huseyin Cavusoglu, Birendra Mishra and Srinivasan Raghunathan. „The effect of internet security breach announcements on market value: Capital market reactions for breached firms and internet security developers“. In: *International Journal of Electronic Commerce* 9.1 (2004), pp. 70–104.

[Coo+07] Ezra Cooper, Sam Lindley, Philip Wadler and Jeremy Yallop. „Links: Web Programming Without Tiers“. In: *Proceedings of the 5th International Conference on Formal Methods for Components and Objects*. FMCO'06. Amsterdam, The Netherlands: Springer-Verlag, 2007, pp. 266–296. ISBN: 3-540-74791-5, 978-3-540-74791-8. URL: http://dl.acm.org/citation.cfm?id=1777707.1777724.

[DD77] Dorothy E. Denning and Peter J. Denning. „Certification of programs for secure information flow“. In: *Commun. ACM* 20.7 (July 1977), pp. 504–513. ISSN: 0001-0782. DOI: 10.1145/359636.359712. URL: http://doi.acm.org/10.1145/359636.359712.

[Den76] Dorothy E. Denning. „A lattice model of secure information flow“. In: *Commun. ACM* 19.5 (May 1976), pp. 236–243. ISSN: 0001-0782. DOI: 10.1145/360051.360056. URL: http://doi.acm.org/10.1145/360051.360056.

[DS00] Premkumar T. Devanbu and Stuart Stubblebine. „Software Engineering for Security: A Roadmap“. In: *Proceedings of the Conference on The Future of Software Engineering.* ICSE ’00. Limerick, Ireland: ACM, 2000, pp. 227–239. ISBN: 1-58113-253-0. DOI: 10.1145/336512.336559. URL: http://doi.acm.org/10.1145/336512.336559.

[FH07] Dinei Florencio and Cormac Herley. „A Large-scale Study of Web Password Habits“. In: *Proceedings of the 16th International Conference on World Wide Web.* WWW ’07. Banff, Alberta, Canada: ACM, 2007, pp. 657–666. ISBN: 978-1-59593-654-7. DOI: 10.1145/1242572.1242661. URL: http://doi.acm.org/10.1145/1242572.1242661.

[Fit14] Andreas Gregor Fitzek. „Development of an ARM TrustZone aware operating system ANDIX OS“. MSc Thesis. 2014.

[Git13] GitHub Inc. *Two-factor Authentication.* Sept. 2013. URL: https://github.com/blog/1614-two-factor-authentication (visited on 25/11/2014).

[GM82] Joseph A Goguen and José Meseguer. „Security Policies and Security Models“. In: *IEEE Symposium on Security and Privacy.* IEEE Computer Society. 1982, pp. 11–20.

[Goo14] Google Inc. *Google 2-Step Verification.* 2014. URL: https://www.google.com/landing/2step/ (visited on 25/11/2014).

[Gos+14] James Gosling, Bill Joy, Guy Steele, Gilda Bracha and Alex Buckley. *The Java® Language Specification.* 2014. URL: http://docs.oracle.com/javase/specs/jls/se8/html/.

[HAM06] Boniface Hicks, Kiyan Ahmadizadeh and Patrick Mcdaniel. „From Languages to Systems: Understanding Practical Application Development in Security-typed Languages“. In: *In Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC 2006).* 2006, pp. 11–15.

[Hew77] Carl Hewitt. „Viewing control structures as patterns of passing messages“. In: *Artificial intelligence* 8.3 (1977), pp. 323–364.

[Hon12] Mat Honan. „How Apple and Amazon security flaws led to my epic hacking“. In: *wired.com, August* 6 (2012). URL: http://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/.

[IWS04] Blake Ives, Kenneth R. Walsh and Helmut Schneider. „The Domino Effect of Password Reuse“. In: *Commun. ACM* 47.4 (Apr. 2004), pp. 75–78. ISSN: 0001-0782. DOI: 10.1145/975817.975820. URL: http://doi.acm.org/10.1145/975817.975820.

[Kal00] Burt Kaliski. „PKCS# 5: Password-Based Cryptography Specification Version 2.0“. In: *Internet Requests for Comments, Internet Engineering Task Force (IETF), RFC* 2898 (2000).

[Kil03] Douglas Kilpatrick. „Privman: A Library for Partitioning Applications“. In: *USENIX Annual Technical Conference, FREENIX Track.* 2003, pp. 273–284.

[Lam73] Butler W. Lampson. „A Note on the Confinement Problem“. In: *Commun. ACM* 16.10 (Oct. 1973), pp. 613–615. ISSN: 0001-0782. DOI: 10.1145/362375.362389. URL: http://doi.acm.org/10.1145/362375.362389.

[Lan+94] Carl E. Landwehr, Alan R. Bull, John P. McDermott and William S. Choi. „A Taxonomy of Computer Program Security Flaws“. In: *ACM Comput. Surv.* 26.3 (Sept. 1994), pp. 211–254. ISSN: 0360-0300. DOI: 10.1145/185403.185412. URL: http://doi.acm.org/10.1145/185403.185412.

[Lie81] Henry Lieberman. *A Preview of Act 1.* Tech. rep. DTIC Document, 1981.

[Liu+09] Jed Liu, Michael D. George, K. Vikram, Xin Qi, Lucas Waye and Andrew C. Myers. „Fabric: A Platform for Secure Distributed Computation and Storage“. In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles.* SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 321–334. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629606. URL: http://doi.acm.org/10.1145/1629575.1629606.

[ML00] Andrew C. Myers and Barbara Liskov. „Protecting Privacy Using the Decentralized Label Model“. In: *ACM Trans. Softw. Eng. Methodol.* 9.4 (Oct. 2000), pp. 410–442. ISSN: 1049-331X. DOI: 10.1145/363516.363526. URL: http://doi.acm.org/10.1145/363516.363526.

[ML97] Andrew C. Myers and Barbara Liskov. „A Decentralized Model for Information Flow Control“. In: *SIGOPS Oper. Syst. Rev.* 31.5 (Oct. 1997), pp. 129–142. ISSN: 0163-5980. DOI: 10.1145/269005.266669. URL: http://doi.acm.org/10.1145/269005.266669.

[ML98] A.C. Myers and B. Liskov. „Complete, safe information flow with decentralized labels“. In: *Security and Privacy, 1998. Proceedings. 1998 IEEE Symposium on.* May 1998, pp. 186–197. DOI: 10.1109/SECPRI.1998.674834.

[MRa+11] D M'Raihi, S Machani, M Pei and J Rydell. „TOTP: Time-Based One-Time Password Algorithm“. In: *Internet Requests for Comments, Internet Engineering Task Force (IETF), RFC* 6238 (2011).

[Mye99a] 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. San Antonio, Texas, USA: ACM, 1999, pp. 228–241. ISBN: 1-58113-095-3. DOI: 10.1145/292540.292561. URL: http://doi.acm.org/10.1145/292540.292561.

[Mye99b] Andrew C Myers. „Mostly-static decentralized information flow control“. PhD thesis. Massachusetts Institute of Technology, 1999.

[NCM03] Nathaniel Nystrom, Michael R. Clarkson and Andrew C. Myers. „Polyglot: An Extensible Compiler Framework for Java“. In: *Proceedings of the 12th International Conference on Compiler Construction*. CC'03. Warsaw, Poland: Springer-Verlag, 2003, pp. 138–152. ISBN: 3-540-00904-3. URL: http://dl.acm.org/citation.cfm?id=1765931.1765947.

[NISa] NIST. *National Vulnerability Database*. URL: https://web.nvd.nist.gov/view/vuln/search-results?adv_search=true&cves=on&pub_date_start_month=4&pub_date_start_year=2014&pub_date_end_month=7&pub_date_end_year=2014 (visited on 11/09/2014).

[NISb] NIST. *National Vulnerability Database*. URL: https://web.nvd.nist.gov/view/vuln/statistics-results?adv_search=true&cves=on&cwe_id=CWE-200&pub_date_start_month=0&pub_date_start_year=2003&pub_date_end_month=11&pub_date_end_year=2013 (visited on 11/09/2014).

[NNH99] F. Nielson, H.R. Nielson and C. Hankin. *Principles of Program Analysis*. Springer, 1999. ISBN: 9783540654100.

[Ok10] Efe A. Ok. *Order Theory and its Applications*. New York University, Dec. 2010. URL: https://files.nyu.edu/eo1/public/Book-PDF/Contents%20(ORDER).pdf.

[Ora14] Oracle Inc. *Java Remote Method Invocation*. 2014. URL: https://docs.oracle.com/javase/7/docs/platform/rmi/spec/rmiTOC.html (visited on 15/11/2014).

[PFH03] Niels Provos, Markus Friedl and Peter Honeyman. „Preventing Privilege Escalation“. In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03. Washington, DC: USENIX Association, 2003, pp. 16–16. URL: http://dl.acm.org/citation.cfm?id=1251353.1251369.

[PS03] François Pottier and Vincent Simonet. „Information Flow Inference for ML“. In: *ACM Trans. Program. Lang. Syst.* 25.1 (Jan. 2003), pp. 117–158. ISSN: 0164-0925. DOI: 10.1145/596980.596983. URL: http://doi.acm.org/10.1145/596980.596983.

[San+11] Nuno Santos, Himanshu Raj, Stefan Saroiu and Alec Wolman. „Trusted Language Runtime (TLR): Enabling Trusted Applications on Smartphones“. In: *Proceedings of the 12th Workshop on Mobile Computing Systems and Applications*. HotMobile '11. Phoenix, Arizona: ACM, 2011, pp. 21–26. ISBN: 978-1-4503-0649-2. DOI: 10.1145/2184489.2184495. URL: http://doi.acm.org/10.1145/2184489.2184495.

[San+14] Nuno Santos, Himanshu Raj, Stefan Saroiu and Alec Wolman. „Using ARM Trustzone to Build a Trusted Language Runtime for Mobile Applications“. In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 67–80. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541949. URL: http://doi.acm.org/10.1145/2541940.2541949.

[San93] Ravi S. Sandhu. „Lattice-based access control models“. In: *Computer* 26.11 (1993), pp. 9–19.

[SCH08] Nikhil Swamy, Brian J. Corcoran and Michael Hicks. „Fable: A Language for Enforcing User-defined Security Policies“. In: *Proceedings of the 2008 IEEE Symposium on Security and Privacy*. SP ’08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 369–383. ISBN: 978-0-7695-3168-7. DOI: 10.1109/SP.2008.29. URL: http://dx.doi.org/10.1109/SP.2008.29.

[SGL06] Manuel Serrano, Erick Gallesio and Florian Loitsch. „Hop: a language for programming the web 2.0“. In: *OOPSLA Companion*. 2006, pp. 975–985.

[SM06] A. Sabelfeld and A. C. Myers. „Language-based Information-Flow Security“. In: *IEEE J.Sel. A. Commun.* 21.1 (Sept. 2006), pp. 5–19. ISSN: 0733-8716. DOI: 10.1109/JSAC.2002.806121. URL: http://dx.doi.org/10.1109/JSAC.2002.806121.

[SR03] Vincent Simonet and Inria Rocquencourt. „Flow Caml in a Nutshell“. In: *Proceedings of the first APPSEM-II workshop*. 2003, pp. 152–165.

[SS75a] Jerome H. Saltzer and Michael D. Schroeder. „The protection of information in computer systems“. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.

[SS75b] Gerald Jay Sussman and Guy L Steele Jr. „Scheme: An Interpreter for Extended Lambda Calculus“. In: *AI Memo* No. 349 (1975).

[SS94] Ravi S Sandhu and Pierangela Samarati. „Access Control: Principles and Practice“. In: *Communications Magazine, IEEE* 32.9 (1994), pp. 40–48.

[Van14] Robbie Vanbrabant. *How Google Authenticator Works*. Sept. 2014. URL: http://garbagecollected.org/2014/09/14/how-google-authenticator-works/ (visited on 10/11/2014).

[Vas+12] Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou, James Newsome and Jonathan M McCune. *Trustworthy Execution on Mobile Devices: What security properties can my mobile platform give me?* Springer, 2012.

[VIS96] Dennis Volpano, Cynthia Irvine and Geoffrey Smith. „A Sound Type System for Secure Flow Analysis“. In: *J. Comput. Secur.* 4.2-3 (Jan. 1996), pp. 167–187. ISSN: 0926-227X. URL: http://dl.acm.org/citation.cfm?id=353629.353648.

[Wu+13] Yongzheng Wu, Jun Sun, Yang Liu and Jin Song Dong. „Automatically Partition Software into Least Privilege Components using Dynamic Data Dependency Analysis“. In: *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE. 2013, pp. 323–333.

[Yee02] Ka-Ping Yee. *User Interaction Design for Secure Systems*. Springer, 2002.

[Zda+02] Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom and Andrew C. Myers. „Secure Program Partitioning“. In: *ACM Trans. Comput. Syst.* 20.3 (Aug. 2002), pp. 283–328. ISSN: 0734-2071. DOI: 10.1145/566340.566343. URL: http://doi.acm.org/10.1145/566340.566343.

[Zda02] Stephan Arthur Zdancewic. „Programming languages for information security“. PhD thesis. Cornell University, 2002. URL: http://www.cis.upenn.edu/~stevez/papers/Zda02.pdf.

[ZM01] Steve Zdancewic and Andrew C. Myers. „Robust Declassification“. In: *Proceedings of the 14th IEEE Workshop on Computer Security Foundations.* CSFW '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 5–. URL: http://dl.acm.org/citation.cfm?id=872752.873524.