A FRAMEWORK FOR COMPOSING SECURITY-TYPED LANGUAGES

APPROVED BY SUPERVISING COMMITTEE:

Jeffery von Ronne, Ph.D., Chair

Ram Krishnan, Ph.D.

Jianwei Niu, Ph.D.

Ravi Sandhu, Ph.D.

Xiaoyin Wang, Ph.D.

Gregory B. White, Ph.D.

Accepted:

Dean, Graduate School

DEDICATION

It's been a long road,
Getting from there to here.
It's been a long time,
But my time is finally near.

Diane Warren, *Faith of the Heart*

This dissertation is dedicated to all the friends and family who accompanied me along the long and winding road.

A FRAMEWORK FOR COMPOSING SECURITY-TYPED LANGUAGES

by

ANDREAS ROBERT GAMPE, B.Sc.

DISSERTATION
Presented to the Graduate Faculty of
The University of Texas at San Antonio
In Partial Fulfillment
Of the Requirements
For the Degree of

DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT SAN ANTONIO
College of Sciences
Department of Computer Science
December 2013

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

ACKNOWLEDGEMENTS

Many people over the years have shared my road with me, influenced me in my direction, and shaped my future. To all I am deeply grateful, and I could never hope to have a complete list here.

I am deeply indebted to my academic advisors Wolfram Amme and Jeffery von Ronne. Wolfram was the one who originally woke the researcher in me, and introduced me to the field of programming languages and compilers. Most importantly, he convinced me to follow that direction and pursue a Ph.D. degree - without him this dissertation would not exist. I am even more grateful to Jeff, who took a chance on a young(-ish) foreigner with the dream of a doctoral degree. He stood by me for over half a decade, teaching me how to become an independent researcher and guiding me through graduate life. Jeff is perhaps the smartest and most integrous person I know, and I have greatly benefited in so many ways.

I want to thank my committee members Ram Krishnan, Jianwei Niu, Ravi Sandhu, Xiaoyin Wang, and Gregory White for spending their precious time with my dissertation, asking critical questions and giving helpful suggestions on how to improve the quality of the thesis.

I am most grateful for my best friend and colleague Omar Chowdhury. Looking back at our first meeting, our first day at UTSA, always makes me smile. Omar has been the greatest help and inspiration. He is the wall I can bounce ideas off, he is the one who grounds me with his vast knowledge of and experience with algorithms. Omar always had an open ear for my concerns and helped me overcome many obstacles.

I would like to thank the myriad of friends and colleagues I met at and through UTSA, in no particular order: Giovanni Del Valle, Keith Harrison, Samira Khan, Bazoumana Kone, Apostolos Kotsiolis, Jean-Michel Lehker, Jane Liang, Jeff McAdams, Arsen Melkonyan, Keyvan Nayyeri, Murillo Pontual, Hui Shen, Rocky Slavin & family, Arpine Soghoyan, Elvira Teran, Yingying Tian, and Emanuelle Vasconcelos. Good times we had, good memories we made. I deeply appreciate their company and support.

Finally, I want to thank my family. To my parents, for their unconditional love and guidance.

Without their sacrifices and attention to my needs, I would not be here. To my sister for the competition: First!

This Masters Thesis/Recital Document or Doctoral Dissertation was produced in accordance with guidelines which permit the inclusion as part of the Masters Thesis/Recital Document or Doctoral Dissertation the text of an original paper, or papers, submitted for publication. The Masters Thesis/Recital Document or Doctoral Dissertation must still conform to all other requirements explained in the Guide for the Preparation of a Masters Thesis/Recital Document or Doctoral Dissertation at The University of Texas at San Antonio. It must include a comprehensive abstract, a full introduction and literature review, and a final overall conclusion. Additional material (procedural and design data as well as descriptions of equipment) must be provided in sufficient detail to allow a clear and precise judgment to be made of the importance and originality of the research reported.

It is acceptable for this Masters Thesis/Recital Document or Doctoral Dissertation to include as chapters authentic copies of papers already published, provided these meet type size, margin, and legibility requirements. In such cases, connecting texts, which provide logical bridges between different manuscripts, are mandatory. Where the student is not the sole author of a manuscript, the student is required to make an explicit statement in the introductory material to that manuscript describing the students contribution to the work and acknowledging the contribution of the other author(s). The signatures of the Supervising Committee which precede all other material in the Masters Thesis/Recital Document or Doctoral Dissertation attest to the accuracy of this statement.

December 2013

A FRAMEWORK FOR COMPOSING SECURITY-TYPED LANGUAGES

Andreas Robert Gampe, Ph.D.
The University of Texas at San Antonio, 2013

Supervising Professor: Jeffery von Ronne, Ph.D., Chair

Ensuring that software protects its users' privacy has become an increasingly pressing challenge. Requiring software to be certified with a secure type system is one enforcement mechanism. Protecting privacy with type systems, however, has only been studied for programs written entirely in a single language, whereas software is frequently implemented using multiple languages specialized for different tasks.

We present a framework that facilitates reasoning over composite languages. In it, guarantees of sufficiently related component languages can be lifted to the composed language. This can significantly lower the burden necessary to certify that such composite programs are safe. Our simple but powerful approach relies, at its core, on computability. We argue that a composition is secure when we can show that an equivalent single-language program is secure. This reasoning can be applied to noninterference, the standard notion of language-based security, as well as declassification, which is a weaker security guarantee necessary for practical uses.

We introduce Security Completeness as the main technical tool to satisfy our framework requirements. Informally, a security-typed languages is security-complete if every secure and computable function can be implemented as a well-typed program in the language. We formally study security completeness and derive sufficient , and in some cases necessary, requirements for a language to be security-complete. A case study of three seminal languages from the literature investigates the three main paradigms of secure languages: imperative, functional and object-oriented languages. We show that, with reasonable assumptions, all case studies are security complete.

To demonstrate the applicability of this framework, we completely show that a standard secure while language satisfies all necessary requirements of a composition host, and present an expressive, security-typed fragment of SQL for embedding.

We finish the thesis with an investigation of dynamically loaded code. A special interpretation of the framework allows it to be used to lift guaranties to programs containing components that are incrementally loaded and verified. This has three benefits. First, incremental loading means a lower start-up time for programs, as the full program is not necessary to start executing. Second, only code that is really necessary for the current computation will be loaded. This results in a decreased bandwidth or time requirement for the application. Third and finally, incremental verification distributes the verification time over the runtime of the program.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1: INTRODUCTION

This thesis is, at its core, concerned with a mismatch between software engineering practice and security theory. Software has become more and more complex over the decades. Large software projects can routinely reach hundreds of thousands or more lines of code. To manage the rising complexity of creating software, systems are created out of (mostly) independent components where each component can be specified, implemented and verified separately. This divide-and-conquer approach is beneficial in multiple ways. For one, it divides a complex problem into smaller chunks, each of which might be easier to solve, and can be solved concurrently to other sub-problems. Second, each sub-problem may be specified, implemented and verified with its own technique, as the specialized problem domain might call for specialized approaches. This, in a sense, is a matter of allowing choice of tools for each separated problem. Last, but not least, by separating sub-problems and developing them indepedently, assumptions and guarantees are established at their boundaries, which allows switching actual implementations as long as they satisfy all the necessary guarantees.

Different fields of computer science have taken up the task to support component-based architectures. We specifically focus on the field of programming languages. Several techniques and paradigms are available to support component-based software that can be broadly categorized into two classes.

Intra-Language Components. On the one hand, there are techniques that allow software systems to be assembled from components that are all written in a single language. This can be done through module support extensions to a core language, as for example in ML. The core of ML is a typed lambda calculus, and is a complete language by itself. The module system allows decomposing software into structures of related values and types, commonly called abstract data types. The three main parts of its module system are signatures, structures and module functors. Signatures are interface descriptions which defines and gives the types of entities that should be provided

by modules. Structures are collections of types, values and sub-structures that may satisfy said signatures. Finally, functors are functions from structures to structures, which can be used to for example implement generic data types.

A second example is the encapsulation and abstraction provided in object-oriented programming languages. In these languages, an (abstract) base class describes the minimum functionality of a component and thus establishes and interface given by the name and types of methods and fields. Sub-classes must implement this interface and are thus components satisfying the given specification.

Inter-Language Components. Another class of composition mechanisms supports building software systems out of components written in different languages. The composition can be shallow or deep. An example of a deep composition is the embedding of a program fragment of one language into a program of another. A prominent example of this technique is the embedding SQL, a domain specific language for querying a relational database, into a web-processing language, for example PHP. In this case, the software has a logic layer and a storage layer, where the logic component communicates with the storage component over a well-defined protocol given by the query language.

Three examples of coupling are common runtimes, e.g., the Java and .Net runtimes, that allow multiple source languages to be compiled to a common intermediate representation that can directly interoperate; foreign-function interfaces that establish how object code of different languages inter-operates; and message-passing, as for example, in inter-process communication.

In our work, we focus on the inter-language component architecture, since it can be used to describe a variety of common settings: from very deeply coupled software made up of object code from different languages, over simple interactions like configuration files, to embeddings of (domain-specific) languages for computation or communication.

In parallel with the need to cope with this rising complexity, privacy is becoming an increasingly important property of software: More and more software handles confidential data and is interconnected over the Internet. The list of prominent examples is ever expanding, but for illustration we will briefly describe two examples here:

Hospitals must manage the confidential *private health information* of their patients. To improve interaction, availability and efficiency, traditional paper-based storage methods are commonly replaced with Electronic Health Record (EHR) systems, software that must manage and store patients' confidential data. An impermissable release of information can lead to heavy financial penalties as mandated by, for example, the Health Insurance Portability and Accountability Act (HIPAA) in the United States. It would be prudent for a hospital to ensure that their EHR system complies with the privacy regulations of HIPAA.

The second example revolves around the ever more-ubiquitous mobile devices in our lifes. Smartphones have become very commonplace and store huge amounts of personal data of their users. For example, emails, text messages, contacts and calendars are commonly stored on them, and the user would likely prefer them to be protected. At the same time, mobile devices support software ecosystems designed around so-called app stores that sell (mostly) inexpensive apps to expand the functionality of the device. It is interesting to note that all three major smartphone operating systems have facilities to separate an app from the storage layer by means of an embedded language: LINQ to SQL in the case of Windows Phone, and SQL to a native SQLite database in both iOS and Android. The prevalent security model of app stores is manual inspection or very coarse-grained permissions. A more fine-grained approach is necessary that can reliably guarantee useful security properties.

A commonly used security property in the area of language-based security is noninterference [43]: program runs differing only in confidential inputs cannot be distinguished by their public outputs. This ensures that an attacker who can only observe public data cannot infer any information about the confidential input values. Typically, this is approximated and enforced as a dependency relation, or more specifically, a lack thereof: there is no dependency of the public output on the pri-

vate input. Two different high-level goals of security can be formulated through noninterference: *confidentiality* and *integrity*. A system ensures confidentiality of data if information can only be read and used by authorized agents (processes or users). It controls where information *flows to*. Here public and confidential represent the notion of secrecy of the data. A system ensures integrity of data if information can only be altered by authorized agents. It controls where information *flows from*. Here public and confidential represent the notion of trust into the data. Overall, systems thus generally control the flows of information (information flow control, IFC). Note that it is standard to only investigate confidentiality because integrity is a dual [19]. As noninterference is the common property enforced in language-based security, in this dissertation, whenever we mention "secure" or "secure systems," we mean this to be "noninterfering" or "noninterferent system."

Two main approaches have been developed to secure traditional monolithic or modular programs. Runtime monitoring performs checks during the execution of a program and stops the execution when a disallowed operation is performed. Two disadvantages of this approach are the potentially significant overhead incurred for the security checks, and label creep, the potential lack of precision due to the locality of checks at runtime that leads to overly restrictive enforcement. Static enforcement, for example static analyses or type systems, approximates the program's behavior ahead of time and a program will only be executed if all checks are known to succeed. While potentially not as precise as runtime monitoring, since actual runtime values are not available at analysis time, static enforcement has the advantage of no (or nearly no) runtime overhead.

Security type systems are a successful and efficient way to guarantee noninterference. There are many proposals for security-typed languages in the literature (e.g., [89, 46, 66, 93, 14, 47, 48], for a slightly dated overview see [78]), starting with the seminal work by Volpano et al.[89] for a simple while language, and all major programming paradigms have been covered. The approach has even been extended to cover entire distributed systems [61].

The two main draws of type systems are (1) that they are a form of static enforcement at compile time, which means low overhead and guidance for a software developer, and (2) they can be formalized and proven secure. All the languages and their type systems refered to above

support compilers that will fail insecure programs, and all have formal proofs of their type systems' soundness, that is, if a program passes the type checker, it is guaranteed secure.

In all of this work, however, exactly one language is treated, while we argued earlier that it is common case in today's software engineering practice to use many languages and components. There are certainly other approaches that already facilitate securing composed software. For example, information flow control on the operating system level [94, 88, 95] can transparently enforce security properties over the communication of processes. However, these approaches treat programs as black boxes, as an analysis of arbitrary programs is not feasible. The granularity is then confined to whole memory regions (e.g., pages) and processes and enforcement must be dynamic, whereas language-based approaches can reason at the level of individual variables and can be static. The generality of the secure OS approach thus comes with a huge lack of precision and severe overhead.

This thesis argues that instead of treating composed programs as black boxes, in many common cases we can do better. It is often possible to mirror the composition process on the type system level. We propose to start with the reasonable assumption that there are security type systems for all languages involved in a composition. (If that is only partially the case, then it is still easier to derive a sound type system for a component language instead of for the complex composed language.) We then extend the host language type system to impose constraints on the embedding of the embedded language.

We show that this simple and straightforward construction is often enough to generically lift the security statements of the component languages to the composed language: If a composed program is well-typed, it is secure.

Our approach is based on the novel property of *Security Completeness*. A language is security-complete iff there exist typable programs for all noninterfering computations in a language. Informally, for every possible computation that respects confidentiality, one can find a valid program that will (a) perform this computation and (b) will pass the type checker of the security type system and can thus be verified to be secure.

In general, security type systems are known to be rather restrictive. As noninterference is a non-trivial property, the problem of checking a program for noninterference is undecidable. Security type systems, on the other hand, need to be decidable to be practical. To resolve this mismatch, type systems either need to accept some insecure programs and become unsound, or reject secure programs and be inprecise. To allow any useful statements, the latter choice is made and the type system restricts the set of secure programs.

It is thus surprising that the requirements for security completeness turn out to be rather easy to fulfill. As this thesis will show, all security type systems known to us are security-complete for computations over integers or similar *primitive* types, that is, computations with inputs and outputs made up of integers or similar atomic types. For *compound* types, it is necessary to reject certain abnormal types that are syntactically valid, but do not have a reasonable semantic meaning.

Turning back to language composition, given a security-complete host language and the constraints imposed on embedding constructs, we can transform a composed program into a purely host language program. This program can be shown to have the same meaning and be typable, so that it is guaranteed to be secure.

On the practical end of the spectrum, we show how to use and extend this setup to secure incrementally loaded code. It is possible to use a structured intermediate code representation and extend it with "holes" into which further code can be embedded. If the holes have to be typed ahead of time, this corresponds tightly to the framework setup described above. On the other hand, we can relax this requirement: we can type-check the incremental code upon reception under the context of the hole's position. If this typing succeeds, it is a valid program fragment to fill the hole and security will be preserved in this case, too.

1.1 Thesis Statement

> It is feasible to control information flow in composed languages with a modular, type-theoretic framework.

1.2 Contributions

The contributions of this thesis are four-fold.

1.2.1 Composition Framework

We detail a formal framework for the composition of security-typed languages. The framework outlines a sufficient process to prove the composition secure under the assumption that the component languages are secure.

To show the practicality of the framework, we formally instantiate the framework for a while language from the literature [89]. We provide all the proofs to complete the framework process and thus show any composition of noninterference-ensuring embedded languages with this while language secure. To demonstrate, we embed a secure SQL fragment to represent a two-tiered software system with a separate storage layer.

1.2.2 Security Completeness

We define a novel property, Security Completeness, for security-typed languages. In short, a language is security-complete iff there exist typable programs for all noninterfering computations in a language. If a language is security-complete, it automatically provides some of the proofs necessary for the composition framework.

Besides the definition of security completeness, we investigate sufficient and necessary properties for a language to be security-complete. Our approach is constructive: if a language fulfills our requirements, we show how to generically transform any noninterfering (but not necessarily typable) program such that the resulting program is both typable and performs an equivalent computation. In case of a Turing-complete language, the sufficient properties are also shown to be necessary.

We formally study three security-typed programming languages under the viewpoint of security completeness. We chose our case studies such that they cover three major programming paradigms:

[89] for procedural programming, [72] for functional programming, and [13] for object-oriented programming.

All languages are shown to be complete with respect to primitive-valued computations. In the case of the latter two, compound types have to be restricted. As a contribution, we formally prove that this restriction is necessary: we formulate a computation using a disallowed type in the system of [72]. This computation is noninterfering, but we prove that it cannot be represented by a typable program in their language.

1.2.3 Security Type Systems for SQL and a dynamic object-oriented language

Domain-specific languages, the main languages to be embedded, have been neglected in the language-based security community. We are not aware of any literature on security type systems for such languages. To show the practicality of our proposed approach, we develop security type systems for two languages typifying two classes of languages that are being embedded in other languages in current software engineering practice.

SQL We investigate an expressive SQL fragment. Our fragment supports arbitrary tables, standard projection, selection, and table joins.

As SQL was our canonical motivating example, we show that it is possible to enforce a security type system regime while supporting most of the features of the data manipulation language part. We formalize the fragment with straightforward syntax and semantics and formally prove that our security type system is sound.

Dynamic object-oriented language We study information flow control for a dynamic object-based language.

We note that a serious complication with type-checking modern scripting languages is the high dynamicity of objects in those languages. In fact, many common idioms are built around the ability to dynamically change the structure of objects at runtime. If a type system is unable to precisely mirror and abstract those changes, it will not be feasible in practice, as it will fail to type practical

programs. As security type systems are simply extended type system, this applies to the security domain as well.

As our part of improving the applicability of a type system-based approach to security, we investigate the security implications of so-called method-not-found errors, which may occur when a non-existing member of an object is invoked. This can be used as a leaking channel.

There are two ways to handle this: the type system could reject the program, if it cannot show that the call always succeeds or always fails, or it can track the information flow of this termination channel. We pursue the latter approach and develop a security type system that enforces security even in the presence of such errors. This reduces the pressure on the type system to be as precise as possible with respect to the structure of objects, and will allow it to type more programs.

1.2.4 Incremental loading and verification of security-typed code

We describe the design and implementation of a novel intermediate code representation (IR) called Nested Control Regions (NCR). This is a low-level structured IR specifically designed to be verified in linear time in a single pass. We provide a compiler from Java to NCR, but the format is independent of the specific input language. On top of the basic representation we layer pluggable type systems [24].

We use this intermediate representation to implement information flow control with dynamic loading. Our security type system is that of [66] based on [67]. Typable programs of [66] can thus be represented by NCR. To support dynamic loading, we extend basic NCR with the concept of holes and a client-server infrastructure to request code to plug a hole when execution reaches that point. The type checker of the NCR runtime support will dynamically check the loaded code and ensure that all safety guarantees remain valid.

As part of the contribution, we formally prove that the approach is sound in an idealized language with dynamic code loading.

1.3 Structure of this dissertation

This dissertation is split into the following chapters. Chapter 2 summarizes required knowledge of domain-specific languages, information flow control, and approaches to formally prove security type systems sound. In Chapter 3, we describe our composition framework, compare it against more traditional approaches, and formally show an instantiation on the case of the embedding of SQL into a standard while language. Following that, in Chapter 4, we define and investigate the key technical tool of Security Completeness. We formalize the property and derive sufficient conditions for primitive languages as well as languages with datatypes and objects & references. The chapter closes with an investigation of three seminal security-typed languages which all turn out to be security-complete. In Chapter 5, we extend the framework and security completeness to cover both nondeterministic languages as well as languages that support two notions of declassification. Chapter 6 describes our efforts to security-type typical embedded languages: both a SQL fragment and an object calculus are furnished with a security type system which are each proven sound. In the following chapter, we explain how the framework approach can be used to facilitate information flow control over incrementally loaded code. We introduce Nested Control Regions, an intermediate representation targetted for simple and efficient verification, and show how to extend it to support incremental loading and verification. Chapter 8 compares our contributions against prior work from the literature.

Chapter 2: BACKGROUND

2.1 Domain-Specific Languages

Most embedded languages are domain-specific, that is, they are each languages made for a particular problem domain or problem representation. This is different from general-purpose languages, which are abstract and always general enough to handle any problem, since they are Turing-complete [1]. Thus they are normally categorized between tiny languages and scripting languages, though the borders are somewhat blurry.

The advantages of restricting a language to a problem domain are three-fold:

Domain-Specific Notation. Since the language is tailored towards a specific domain, problems in that domain can be expressed in a clearer and more concise way. The primitives in the language are geared towards the problem domain, which allows the reuse of rich domain-specific notations.

As an example, a language for mathematical expressions can introduce symbols for sum, product, derivation and integration to allow a more natural way of writing down expressions in a program. With the advent of Unicode, many of these symbols are available natively.

Concise & High-Level. With the domain-specific notation, the abstraction level is adapted to the problems. Grammar and symbols can allow a shorter and more precise specification of a problem, because the language works on a high level and low-level details are omitted.

On the example of the mathematical expression language, a summation symbol Σ is a convenient and concise abbreviation for a regular sum, as is Π for regular products.

End-User Programming. Domain-specific languages may allow more people to create programs. The specialization on a domain and the use of well-known notation allow people not formally trained in programming, but familiar with the domain, to write software.

[1] Note that, however, some embedded languages are Turing-complete and thus general-purpose,e.g., Javascript for NoSQL.

Mathematics is the basis of many science disciplines, and as such many people understand and use math notation. While these people may not be trained programmers, a domain-specific expression language will ease any programming efforts necessary for them.

Domain-specific languages can be used standalone or in concert with other domain-specific or general-purpose languages. In standalone settings, the compiler provides an environment to form whole programs. Sometimes sets of domain-specific languages are used, where each language performs one part of the task of a system. For example, a program might be broken up into a user interface description, a part for interaction with a database, and finally a central part specifying the business processes. Such sets or part thereof are known as fourth-generation languages (4GL), a description commonly used in the eighties.

In many practical instances, domain-specific languages are *embedded* into a *host* general-purpose language. The most prominent examples are querying and interface description languages. Querying languages allow to specify questions or queries to data stored either in the program itself or in the environment. The prototypical example is SQL, which describes queries in a relational database. When embedded in a host language, SQL then forms a connection to an external resource. The exact query syntax can be generalized to also apply to other data sources, for example XML documents and even in-memory arrays, as has been done with LINQ.

A key difference between LINQ and SQL is the standard style of embedding. SQL is traditionally handled in string variables that are handed of to special-purpose commands that execute the query. An example in PHP might look like this:

```
$sql = "SELECT * FROM table";
if (condition) {
  $sql = $sql + " WHERE column > value";
}
$res = mysql_query($sql);
```

This example also shows an advantage of string-based embedding: The embedded code can be

programmatically created and manipulated, which endows the developer with a high flexibility when writing a program. However, this flexibility comes at the price of static checkability. Other popular querying languages often used in string-based embeddings are XPath and XQuery, which describe queries over XML documents.

To overcome the absence of static checks, several techniques try to integrate the embedded language closer into the host language. The Cω project introduces streams into a simplified variant of C$^{\#}$, with the techniques finally being integrated into mainline C$^{\#}$ under the name of Language-integrated Native Queries (LINQ). LINQ syntax is somewhat modeled after SQL and directly integrated into the host language, so that the compiler understands the query.

A stream is a sequence values. Streams are flat, that is, streams are never nested. Operations over values are lifted to streams by application to all elements. In this case, operations also include member selection in records and objects, as well as conditionals for filtering elements. This generalizes the access of various kinds of semi-structured data like relational databases and documents. Streams can be typed, similar to arrays (the Cω notation for a stream with elements of type `int` is `*int`). This allows the validity of operations on streams to be checked. For example, members can't be selected on a stream of `int`, and arithmetic operations can't be performed on a stream of strings.

An advantage of integration into the host language is the ability to perform static checks at compile-time: the sub-program can be syntactically checked as well as type-checked for a (simple) check of well-formedness.

Not only domain-specific languages can be embedded into a host language. It is common to use scripting languages for extensibility in a software architecture. Another use is for executable configuration, where a script programmatically configures a program, instead of the program parsing some configuration files. Furthermore, recently general-purpose scripting languages (or subsets thereof) have been suggested for querying in NoSQL databases. For example, both MongoDB and CouchDB can use Javascript for queries and map-reduce operations.

Most scripting languages are categorized as dynamically typed. As such, establishing a static

type discipline can be hard. Considerable work has been invested to study several scripting languages, most of which are object-oriented. A small list of examples follows.

Diamondback Ruby (DRuby) is a static type discipline for Ruby. It supports type annotations as well as type inference. Supported kinds of types are intersection and union types, simple object and method types, self types and parametric polymorphism. DRuby supports a profile-guided typing mode and dynamic checks for the highly dynamic features of Ruby like `eval`.

Several techniques have been studied for restricted subsets of the Python programming language. RPython is used in the PyPy compiler and fully type-inferable. A similar project is Starkiller. Shed Skin compiles a subset of Python to C++, using type inference for optimizations.

Several approaches try to type significant parts of Javascript. Anderson's inference [8] is an extension of approaches developed for primitive object calculi like [1]. [96] extends the approach with singleton types, while [45] use a recency abstraction to increase precision.

2.2 Information Flow Control

Many programs handle sensitive data, for example in banking, health care and military environments. Early on, the need for security properties was recognized. Three properties are usually used to describe safety:

- *Confidentiality* or privacy describes the secure release of information. Systems only make data available to agents authorized to access it. A common example is the value of a person's bank account. The bank server needs to make this information available to the account holder, and possibly bank employees that work with the account holder, but not to any other client of the bank.

- *Integrity* ensures that data is not compromised and only modified by authorized agents. Continuing with the banking example, a bank balance is only effected by logged, auditable, authorized transactions. That may be as the result of an action authorized by the account holder. For example, the holder may directly withdraw money, or set up automatic payments. On

the other hand, the bank is authorized to impose fees and deposit interest to the accounts. A second, unrelated account holder, however, should never be able to change the first's bank balance without express permission.

- *Availability* or reliability ensures the continued service of a program. A system should be accessible for as much time as possible.

As is usual, only confidentiality and integrity are studied here. Furthermore, as explored by [19], confidentiality and integrity are dual. For the main part of our study concerning noninterference, it thus suffices to focus on confidentiality. Robust declassification in Chapter 5, however, relies on the interplay between integrity and confidentiality, and we will handle both in that part of this thesis. Two orthogonal problems arise for confidentiality: modeling access rights, i.e., a policy, and enforcement of that policy.

Three main access control models exist. In discretionary access control (DAC), the owner of an object decides who is allowed to access the object. Furthermore, having access to an object implies being able to obtain a copy of said object under one own's control. In mandatory access control (MAC), the system determines the policy instead of the owner of an object. Subjects and objects have labels, and rules over labels define access rights. In role-based access control (RBAC), subjects perform roles which endow them with sets of permissions for performing certain actions.

While DAC is the current standard in consumer-grade operating systems, MAC or RBAC are necessary for stronger guarantees. This is the case because DAC does not protect data after the owner allowed another subject access to the information. For stronger security, information needs to be protected even after being initially released to a subject. This is called *information flow control* (IFC), because full flows from source to eventual sinks are traced and controlled.

Several categories of information flows have been defined. *Direct flows* exist because of straight-up data transfer, as for example in assignments:

```
a := b;
```

Here, information flows from `b` to `a` because of a direct data dependency.

Information can also be transferred through control dependencies, for example, conditional execution. This is usually called an *indirect flow*. An example is the following:

```
b := 0;
if (a > 0) then
  b := 1;
end;
```

In that case, one bit of information about `a` is transferred indirectly to b through the seemingly innocuous values `0` and `1`. Several other reasons for indirect flows exist if the language contains more complex structures, for example, dispatch of virtual methods and control flow induced by exceptions.

Other examples of flows are termination and timing. A termination flow or channel exists in the case that confidential data determines if a program terminates or not - an attacker able to observe the program may be able to tell the difference. An example of this is a potentially infinite loop:

```
while (a > 0) do
  nop
```

In the case that `a` is greater than zero, the program will not terminate. Thus the termination gives one bit of information.

Timing channels rely on the attacker being able to observe the program and have the ability to distinguish time. Thus differences in computations may be observed, even though the end result is the same. Examples for this are simple repeated computations, e.g., repeat a computation in a loop decrementing a confidential counter:

```
tmp = a;
while (tmp > 0) do
  compute
  tmp--
```

or more advanced flows based on hardware properties, e.g., caching behavior and the resulting differences in memory access response times.

Termination and timing flows are usually either low bandwidth or hard to exploit. Further, correctly handling either without being excessively conservative is, not surprisingly, very hard. As such, most research focuses exclusively on the different forms of direct and indirect flows.

The goal for information flow control is to define valid flows and enforce that only valid flows are possible in a program. A standard way to define valid flows is lattice-based security [32].

2.2.1 Lattice-based Security

A lattice $(S, \sqsubseteq)$ is a partially ordered set in which every two elements have a least upper bound (join, or supremum) and a greatest lower bound (meet, or infimum). Elements of the lattice can be seen as security levels. The partial order defines in which direction information may flow. For example, assume that $\ell_1, \ell_2 \in S$ and $\ell_1 \sqsubseteq \ell_2$. The meaning of this is that ℓ_2 is at least as restrictive as ℓ_1, and so information is allowed to flow from ℓ_1 to ℓ_2.

A standard security lattice that is minimal and still meaningful is based on the two-element set $LH = \{L, H\}$. Here L stands for low confidentiality or public, and H for high confidentiality or private. Thus the partial (and in this case total) order of elements is $L \sqsubseteq L$, $L \sqsubseteq H$, $H \sqsubseteq H$, but $H \not\sqsubseteq L$. This lattice allows flows in a security level, and from public to private, but not private to public. Often, proofs derived using this specific lattice can be generalized to generic lattices.

While the simplicity of the LH lattice makes it a prime target for theoretical developments, more complex lattices have been proposed in the literature as well as are being used in practice. A prominent example is the Distributed Label Model [67], which is the security lattice used for JFlow [66] and its extensions, e.g., [61].

Given a security lattice, all entities (e.g., values, storage areas, computations, channel sources and sinks) are labeled with lattice elements. In case of dynamic enforcement, actual runtime elements like values are labeled. For example, a normal summation $3 + 5$ will be extended to, e.g., $3^L + 5^H = 8^H$. For static enforcement, it often suffices to label static parts of a program,

e.g., variables and program code, but erase those labels before running the program under a basic semantics. The label of the variable is assumed/guaranteed to be an upper bound on the intended labels of values that will be stored in said variable. That is, we might have $x^L := 3^L;\ y^H := 5^H;\ z^H := x + y$, which resolves to $x; = 3;\ y; = 5;\ z := x + y$ at run time.

A limitation of lattice-based security definitions is that it cannot directly incorporate notions of declassification. Lattices are based on partially ordered sets, so element ordering is transitive. That is, if $\ell_1 \sqsubseteq \ell_2$ and $\ell_2 \sqsubseteq \ell_3$, then it must be the case that $\ell_1 \sqsubseteq \ell_3$. It follows that there cannot be an allowed transition from a secret level to a public level, as this would close a cycle. As an example, assume that information might flow from ℓ_1 to ℓ_2, from ℓ_2 to ℓ_3, and from ℓ_3 being declassified to ℓ_1. In that case, if the elements are supposed to form a lattice, then all three elements are equivalent: as we have $\ell_1 \sqsubseteq \ell_2 \sqsubseteq \ell_3 \sqsubseteq \ell_1$, for example, it follows $\ell_1 \sqsubseteq \ell_3$ and $\ell_3 \sqsubseteq \ell_1$, which by antisymmetry constraints on partial orders implies $\ell_1 = \ell_3$.

2.2.2 Noninterference

A common goal for policies is to establish noninterference. Informally, noninterference states that confidential inputs do not influence non-confidential outputs of a program. In flow notation, this is equivalent of no flows from confidential inputs to non-confidential outputs. This formalizes the expectation that a secure system must not allow an attacker to deduce any confidential information stored or handled in the system, under the assumption that the attacker can only observe public information and behavior.

Formalized noninterference statements have different shapes depending on the setting they are used in. In trace-based settings, for example, the notion of noninterference is based on the closure of the set of traces under the deletion of confidential events/states.

We are interested in language-based noninterference. In this context, noninterference statements connect input and output states. If two inputs are *indistinguishable*, then the outputs must be indistinguishable. Typically, $\sim$ is used for denoting a indistinguishability relation. Indistinguishability can be defined by choosing a lattice element ℓ_a as the attacker/observer level. Two

states are indistinguishable iff all elements that are below or equal to level ℓ_a are equivalent.

Equivalence can have different meanings depending on the language features. In a simple while language, variables store single integers, and integers have a simple equivalence relation given by $=$ (structural equivalence). As an example, given a state which comprises storage for variables a and b, where a is public and b is private. Then states $\langle a : 1, b : 2\rangle$ and $\langle a : 1, b : 3\rangle$ are indistinguishable for an attacker at level L, but states $\langle a : 1, b : 2\rangle$ and $\langle a : 2, b : 3\rangle$ are distinguishable.

In languages with first-order computational elements, like lambda expressions in functional programming, structural equivalence can potentially be relaxed to observational equivalence. This would indicate that attackers are not able to inspect a lambda expression directly, but may invoke and check its results. Under structural equivalence, the two expressions $\lambda x.x$ and $\lambda x.2 \times x - x$ are not the same, as the term x is not equivalent to the term $2 \times x - x$. Under observational equivalence, these expressions are equivalent, however: for any input value v, both functions return the same value v.

Finally, in languages with a heap, e.g., most object-oriented languages, heaps must be related accordingly. Typically it is assumed that references are opaque entities from a user's point of view. If this is not the case, there are very strict requirements on the heap cell allocator to ensure that no leaks through addressing appear. Even if the references are opaque, for a noninterference statement it is necessary to relate states with different allocated cells. A simple example of two allocation sequences will demonstrate the problem.

Assume a simple sequential language with an alloc construct that reserves the next free heap cell, and let heap cells be indexed by natural numbers. We inspect runs of the following program, where variables called x are public and y confidential. $x_1 := \mathrm{alloc};\ \mathrm{if}\ (y_1 > 0)\ \mathrm{then}\ y_2 := \mathrm{alloc}\ \mathrm{end};\ x_2 := \mathrm{alloc}$. Let us start with a state where all variables are initialized to zero[2]. Then the result of the program is the state $\langle x_1 : 1, x_2 : 2, y_1 : 0, y_2 : 0\rangle$, denoting that space for x_1 was reserved in cell 1, and space for x_2 in cell 2. If we, however, start with a state where

[2]For $x_{1/2}$ and y_2 this denotes a null pointer.

$y_1 = k > 0$, the result is $\langle x_1 : 1, x_2 : 3, y_1 : k, y_2 : 2\rangle$, where the space for y_2 is allocated before the respective space for x_2. Note that for an attacker this difference is *not* visible, but a simple value comparison will distinguish those states. A general technique is to declare states distinguishable if there are bijections (mappings) β between memory cells, such that the states become equivalent. For example, such a mapping would be $\beta \equiv 1 \mapsto 1, 2 \mapsto 3$. Mappings between inputs and outputs must then be suitably related.

2.2.3 Declassification

Note that in practice, noninterference is often infeasible. A limited amount of information leaks is necessary for correct system operation. The traditional example is a login process.

A login process takes a username and password as input. It is then supposed to compare this input against some internal database. For simplicity, the process should output a simple "Yes" or "No" depending on the result, indicating whether the login succeeded.

As a user is not authenticated in the beginning, the login process is a public service, so inputs should be considered public. The internal database should be restricted, as it stores the confidential passwords of registered users. The output needs to be reported back to the as-of-yet unauthenticated user. It should thus be public.

The correct answer for the login process depends on both the input and the internal state, as the answer should be "Yes" if and only if public input and private state agree. With noninterference, the login process could not report success or failure of an authentication, i.e., if a username-password combination was correct.

Here it is obvious that to be functional, some systems need to have *leaks*. Such leaks will *declassify* confidential data to non-confidential data. Other examples for systems that need this functionality are auditing systems (e.g., the last four digits of credit cards on receipts) and encryption (where an encrypted text is assumed not to allow deduction of its confidential input).

For such cases, policies may allow limited forms of *declassification*, which transform confidential data to non-confidential data. As a system with declassification operations cannot be

guaranteed to be noninterfering in general, two interesting questions arise.

Validity of Declassification. If a program can arbitrarily use declassification operations, no security guarantees could be established, as any confidential input could be declassified and printed to a public output channel.

There is no general or "best" notion of declassification in the literature. Sabelfeld and Sands [80] survey and classify several versions of declassification. The categories are the orthogonal axes "what," "who," "when," and "where." The *what* category describes which data is allowed to be declassified. The *who* category describes restrictions to which principals may be allowed to declassify or influence (e.g., through control flow) declassification. The *when* category ties timing into declassification and restricts for example by ordering or external time events. Finally, the *where* category restricts declassification to certain parts of a program source or the location of a principal.

Security Guarantees of Declassification. Given the validity of declassification operations in a program, the immediate follow-up is what formal security guarantees can actually be established. This is often a modified, relaxed form of noninterference.

For our investigations, we will focus on two forms of declassification from the literature. Delimited release [79] is an example of the *what* axis (cf. [80], Section 2.1). Robust declassification [92, 68] is an example of the *who* axis (cf. [80], Section 2.2).

Delimited Release. Delimited release defines declassification operations $\mathrm{declassify}(e, \ell)$. The meaning of this operation is to evaluate expression e and declassify the resulting value to security level ℓ[3]. Expressions e that are parameters to a $\mathrm{declassify}$ are called escape hatches, as information in e is allowed to "leak."

[3]The paper fits into the language-based security category. The runtime meaning of $\mathrm{declassify}$ is thus simply to evaluate its expression e.

The security guarantee is a weakened noninterference statement. Instead of guaranteeing indistinguishable outputs for all indistinguishable inputs, delimited release requires a side condition: all escape hatches declassifying to an observable security level must have the same value for respective input states. Formally, a program p satisfies delimited release at level ℓ iff

$$\forall \mu_1, \mu_2, \mu_1', \mu_2'. \begin{array}{c} \mu_1 \rightsquigarrow_p \mu_1' \wedge \mu_2 \rightsquigarrow_p \mu_2' \wedge \mu_1 \sim_\ell \mu_2 \wedge \forall i.(\ell_{e_i} \sqsubseteq \ell \implies \mu_1(e_i) = \mu_2(e_i)) \\ \implies \\ \mu_1' \sim_\ell \mu_2' \end{array},$$

where μ denotes states, $\rightsquigarrow$ is a relation connecting input states to output states parameterized over programs p, e_i are the escape hatches in p with respective declassification to ℓ_{e_i}, and $\mu(e)$ denotes the evaluation of e under state μ. Note that the only difference to a standard termination-insensitive noninterference statement is the term $\forall i.(\ell_{e_i} \sqsubseteq \ell \implies \mu_1(e_i) = \mu_2(e_i))$, which ties in the values of the escape hatches at program start.

In [79], Sabelfeld and Myers show how to use an extended security type system to enforce delimited release in a while language with declassification.

Robust Declassification. In [92], Zdancewic and Myers define *robustness* as the property of a system under attack. A system is robust if an active attacker cannot gain more information than a passive attacker. That is, if a passive attacker cannot distinguish two states, e.g., by running the system over them and not being able to find differences, then an active attacker that can influence the states or behavior of the system cannot, either.

In [92], systems are described as state-based transition systems. Equivalence relations $\approx$, or *views* in their terminology, describe the ability of an attacker to distinguish states. Traces τ, that is, connected sequences of states, are system runs and a view $\approx$ induces canonical traces $\tau/\approx$ defined by the equivalence classes of states induced by the view. An *observation* $O(\sigma, \approx)$ is the set of all canonical traces under the given view starting at a specific state. This formalizes the notion that an attacker does not necessarily have perfect knowledge of a system: A canonical trace

is an equivalence class for concrete traces induced by a view, and given one such concrete trace, an attacker with the observational power given by the view cannot distinguish said trace from any other trace represented by the canonical one; an observation gives all canonical traces starting with a specific state and thus all runs (with respect to observable behavior) that can be distinguished starting at said state. Note that a sequence of equivalent states is taken to be equivalent to a single-state trace with said state. This means all definitions and guarantees assume equivalence under stuttering.

Observations themselves induce views $S[\approx]$ over a system S: under this view two states are equivalent if their respective observations are the same. This is a formalization of observational equivalence. If the observations are the same, then for each concrete trace starting at the first state, it is possible to find a concrete trace starting at the second state such that both traces are equivalent under the given view. As such, an attacker is unable to deduce the concrete start state given a program run.

A passive attacker is an attacker who can run the system. Informally, he is able to compute the observation-induced view. This is used to define basic security of a system: A system S is secure under a view $\approx$ iff a passive attacker cannot gain information, that is, if two states are equivalent under said view, then their respective observations under the view are the same:

$$\forall \sigma_1, \sigma_2 . \sigma_1 \approx \sigma_2 \implies \sigma_1 S[\approx] \sigma_2$$

In the transition-system setting, active attackers are allowed to influence the system by adding state transitions. This formalizes the notion that an active attacker can influence the execution of a program. Note that the state space itself is assumed to be invariant - the attacker may only add new transitions between alreay existing states.

To curb the power of the attacker, [92] restricts added transitions to be, by themselves, secure. The authors argue that this is a reasonable and practical restriction. One can interprete the restriction as the base system being open, e.g., a system that can load plugins, but not executing arbitrary

code. In said system, all dynamically loaded code could be checked to be secure by itself, for example by static analysis or through certificates. The question of robustness is then whether an attacker can use this facility to gain information.

Formally, robustness is then a comparison between observation-induced views of the original system and the system under attack. If two status are equivalent in the original system, a robust system does not allow them to be distinguished even when attacks are performed. Formally, let S be the original system and A an attack, that is, an additional system over the states of S that is secure with respect to $\approx$. Then a robust system satisfies

$$\forall \sigma_1, \sigma_2 . \sigma_1 S[\approx]\sigma_2 \implies \sigma_1(S \cup A)[\approx]\sigma_2$$

where $S \cup A$ denotes the system under attack, which is the set of states of S and all transitions from S and A.

In [68], Myers et al. translate the transition-based definition of [92] to a language-based setting. A simple while language is extended with a declassification operation and "holes." A program with a hole is a context into which other code can be embedded. A context thus defines a family of related programs. Once all holes in a context are filled, the whole program can be executed.

The basic security guarantee in [68] is noninterference. Attack code is thus program code that is noninterfering. Robustness is translated to comparing the runs of programs under substitution of holes with different attacks: if two runs over some start states cannot be distinguished under attack a, they cannot be distinguished under any other attack a', either.

$$\forall c, M_1, M_2, a, a' . \langle M_1, c[a] \rangle \cong \langle M_2, c[a] \rangle \implies \langle M_1, c[a'] \rangle \approx \langle M_2, c[a'] \rangle,$$

where $\langle M_1, c_1 \rangle \cong \langle M_2, c_2 \rangle$ iff the executions of both configurations are both terminating and indistinguishable. On the other hand, $\approx$ relaxes $\cong$ so that configurations are also equivalent if either or both configurations diverge, that is, do not terminate.

Note that there are two differences between the robustness definitions of [92] and [68]. The

first one is nondeterminism. The transition-system based definition in [92] inherently allows nondeterminism. Observations are sets of canonical traces, which themselves stand for potentially many different concrete traces. The definition and base language of [68], on the other hand, are purely deterministic.

The second difference concerns termination-sensitivity. The definition of [92] is termination-sensitive: if an observation with visibly-changing states can be made from one state, the second state must exhibit an equivalent trace. Thus, non-termination is not generally compatible with another terminating trace, and so the definition in [92] is termination-sensitive. On the other hand noninterference and robustness in [68] are explicitly termination-insensitive: two configurations are $\approx$-equivalent if either both converge and the constructed traces are equivalent, or one or both diverge.

To enforce the language-based robustness definition, [68] proposes a type system enforcing both confidentiality and integrity at the same time. Further, declassification can only be performed in a high-integrity environment, attacks cannot change high-integrity state, and holes do only appear in low-confidentiality environment. Under these restrictions, enforced by the security type system, the authors stipulate that programs are guaranteed to be robust[4].

2.2.4 Enforcement Mechanisms

Enforcement mechanisms can be broadly categorized as static or dynamic. Dynamic enforcement monitors a program at runtime and performs checks before sensitive actions. Static enforcement analyzes a program before it is being run - either at compile-time or at load-time - and ensures that no matter what the program inputs are, no execution will ever perform an unallowed operation. Dynamic enforcement has the advantages of flexibility and precision, since runtime values can be inspected. However, that means a certain runtime overhead for security checks is involved. Static enforcement, on the other side, ensures that a program can never fail and has no (or no significant) overhead at runtime.

[4]The proofs outlined in the appendix of [68] have a flaw applying an inductive hypothesis when not all prerequisites are satisfied: in fact, the main issue is one of termination-sensitivity.

Several systems have been proposed for dynamic enforcement. They include monitors that check single applications, up to whole operation systems like HiStar and Flume. The main disadvantage of runtime monitoring is the overhead incurred due to runtime checks. Runtime monitoring needs to find a balance between checks and precision. Either all operations are checked and the security information has very fine granularity, or multiple operations are combined and checks only occur at the block boundaries, which lowers the precision. Further, without a preceding detailed static analysis (or prior knowledge of a program's structure), runtime monitoring is prone to *label creep*.

Static enforcement tries to verify a program at some time prior to execution. Only verified programs are ever executed, and guaranteed to comply with a given policy. Security verification can be done traditionally, for example, with logics (e.g., extended Hoare logic) or model checking. A static analysis can overapproximate all flows of information of a program. An extended type system can give a compiler enough information to check programs before compiling them to actual code.

Examples for logic-based approaches are [6, 7]. A proof of noninterference is established by extending logic primitives to statements over multiple runs of programs, and showing that the runs agree on the values of non-confidential variables. For example, $\bowtie a$ in a pre-condition states the assumption that the value of variable a is not distinguishable to an attacker in two runs of a program, i.e., it is a confidential input. If the proof rules of the logic then allow to establish the post-condition $\bowtie b$, b can be a public output, because it cannot be distinguished over different runs.

The logic-based approach inherits the difficulties of theorem proving from Hoare logic, though. Currently, program verification needs a non-trivial amount of human intervention in proof construction for any non-trivial program.

Information flows describe a set of dependencies. As such, static dependency analysis can be used to compute an approximation of all possible flows in a system. To be safe, the approximation needs to be conservative and overapproximate the flows. An additional, unrealizable flow will in the worst case reject a program, but a missed flow might result in a leak.

A static analysis can be potentially very precise. However, analysis time and modularity are two significant problems. A recent example for a static analysis-based approach is [27], which analyzes Javascript code. The analysis is constraint-based. Set constraints of dependences are established through a pass over the representation of the program, and a constraint solver finds a solution satisfying the constraints. The constructed sets are then checked for violations of noninterference. A significant problem is the runtime of the constraint solver. In fact, in the proposed system, the constraint solving is delegated to a dedicated server. Only a lightweight verification of very simple syntactical checks is done in the browser, which does not give the same security guarantees on the client side as the algorithm on the server.

The other problem is modularity of analysis and verification of results. Modularity is important because most software is composed of libraries. A full analysis may require access to the library code[5], which is not always available, for example, in the case of proprietary commercial libraries. Summaries may be used to alleviate the need, but this has a severe negative impact on precision. Related to modularity, highly precise static analyses cannot be easily verified. For example, many analyses gain precision by repeatedly analyzing a method if it is called from different callsites. This context-sensitivity avoids summaries that have to unite information from potentially very different circumstances. However, now each callsite needs to re-verify a called method, too.

Extended type systems can be used to enforce security on the language level. The seminal work in that area is [89], which is based on lattice-based security by [32]. Types and typing rules allow a compiler to infer or verify that a program does not violate confidentiality constraints. In most cases, the security type systems are based on traditional type systems, which are annotated with security information. For example, a type $\texttt{int}^L$ might describe values of type integer that are public, while $\texttt{String}^H$ describes strings which may be confidential. Typing rules or judgments then enforce that confidential variables are not used in computations stored in public memory.

Type systems have the advantage of (relatively) simple verification and easy integration into compilers. This allows for usually fast checks at compile time and programmer support at the

[5] Analysis is significantly harder on the binary level.

IDE level. Negative points are a somewhat increased verbosity, because type inference is an open problem and so programmers need to use annotations in source programs; and a certain lack of precision compared to static analyses (see above), because types summarize behavior and precise summaries require very complex type systems.

Several proposals for research and practical security-typed languages exist. Of great practical importance are JIF [66] and FlowML [72], two extensions of mainstream languages for software development (Java and ML, respectively). At the moment, language-based systems are the state-of-the-art in matters of practicality and verifiability. For these reasons, the contributions of this dissertation are focused on language-based approaches to information-flow control.

[89] describe a type system for a simple WHILE language. The language contains variables, number literals, arithmetic operations, comparison operations, conditionals and a while loop element. An extension describes how to correctly handle procedures. From that point, over the last two decades numerous languages have been treated with security extensions. A small number of examples follow. For an older survey we refer to [78]. The SLam calculus [46] treats functional languages in a theoretical manner, while FlowML [72] extends a practical functional language (ML). Object-oriented programming and security type systems have been studied in [16, 66, 15, 13]. JIF [66] is the extension of Java with security annotations.

As mentioned earlier, most language-based security is based on a lattice model of security levels. Types are extended to include security labels, with the intuitive meaning that the label is an *upper bound* on the labels of values inhabiting that type. For example, 5^L is a member of $\texttt{int}^L$, as well as $\texttt{int}^H$, while "Hello World"^{H} is not a member of $\texttt{String}^L$. This meaning lifts the lattice order to a subtyping relationship in the type system, and is formalized as a subsumption rule in the security type system.

Further rules besides subtyping capture the semantics of all syntactical elements in the language. In the case of imperative languages, a simple example would be the assignment rule:

$$\frac{\Gamma \vdash a : \texttt{int}^{\phi} \quad \Gamma \vdash b : \texttt{int}^{\psi} \quad \psi \sqsubseteq \phi}{\Gamma \vdash a := b : OK}$$

In this rule, several conventions for security type systems are shown. Typing rules usually derive judgments under certain assumptions, here shown by Γ. A collection of such assumptions is typically called an environment. Usual environments are, for example, variable mappings that store the current types of local variables, and heap mappings that store the types of heap cells. The first two judgments $\Gamma \vdash a : \texttt{int}^{\phi}$ and $\Gamma \vdash b : \texttt{int}^{\psi}$ in the premise derive security types for the expressions used in the assignment statement, where metavariables ϕ and ψ range over security labels. Variable a should be typable as $\texttt{int}^{\phi}$, that is, it stores integers with security at most ϕ. Variable b is similar with an upper bound of ψ. Finally, $\psi \sqsubseteq \phi$ formalizes that values can only be stored in variables with types at least as restrictive as the value. This prevents direct flows, that is, leaks by directly storing confidential information in public memory.

Many of the languages are accompanied by proofs that guarantee the safety of the type system: if a program is well-typed, that is, there exist type environment and type such that a typing judgment of the program can be derived, then it has the property of noninterference. This is one of the practical strengths of security type systems: a typing is a certificate for a formal guarantee that a well-typed program is secure. Security type systems, under this viewpoint, can be seen as instantiations of proof-carrying code [69]. In the next section, we will sketch the approaches to formalize and establish the theoretical foundations.

2.3 Proof Approaches for Type System based Security

There are two commonly used techniques to establish noninterference results. The older one, first used in [89], can be classified as a hybrid syntactical and semantical approach. Noninterference is split into two properties that together imply noninterference. The second one pioneered in [72] is syntactical and adapts the approach of Wright and Felleisen [91] to noninterference. Here, one estables progress and preservation lemmas and deduces noninterference from them.

There are other approaches mentioned in the literature, for example logical relations as used in [46, 13]. We only outline the major approaches here, but note that the shortcommings extend to other approaches, as well.

2.3.1 Mixed Syntactical-Semantical

Volpano et al. [89] derive the properties of Simple Security ([89], Lemma 6.3) and Confinement ([89], Lemma 6.4), that together imply noninterference. Both can be seen as traditional properties from [17, 53] adapted to the language-based security setting. Simple Security states that if an expression e has been typed at level ℓ, then all variables or locations used in e have a level at or below ℓ with respect to the typing environment:

$$\Gamma \vdash e : \ell \implies \forall v \in e.\Gamma(v) \sqsubseteq \ell \wedge \forall l \in e.\Gamma(l) \sqsubseteq \ell$$

This is roughly equivalent to the same-named Simple Security property of Bell & LaPadula, which is commonly summarized as "no read up:" a subject at a a given security level may not read an object at a higher level.

Confinement states that if a statement c is typed at level ℓ, then all assignments in c are to variables v or locations l with at least level ℓ:

$$\Gamma \vdash c : \ell \text{ cmd} \implies \forall v \in \text{assign}_v(c).\Gamma(v) \sqsupseteq \ell \wedge \forall l \in \text{assign}_l(c).\Gamma(l) \sqsupseteq \ell$$

This is roughly equivalent to the $*$-property of Bell & LaPadula, which is commonly summarized as "no write down:" a subject at a given security level may not write to an object at a lower level.

Both Simple Security and Confinement are proved for the security type system by inductions over the structure of the given expression or command, that is, one inspects the expression or command and analyzes a case for each possible constructor. As an example, structural induction on a statement c needs to have cases for assignment, conditional, while loop and sequencing.

The final soundness theorem, stipulating noninterference in the case of typed statements, is proved by induction on the structure of the derivation of a program run. In that case, the induction needs to have cases for every semantic rule of the language. For [89], this means the induction needs cases for variable update (for assignment statements), true and false branches (for condition-

als), iteration and termination (for loops), and "sub-executions" (for sequences).

2.3.2 Purely Syntactical

Wright and Felleisen [91] proposed a purely syntactical approach to traditional type soundness: one first establishes progress and preservation lemmas, and then uses those to argue that the result of a computation satisfies the required constraints.

Progress stipulates that a typed expression or statement is either a value, that is, cannot be reduced further, or progress can be made, that is, the term is not stuck and thus does not denote an error state. A sample formulation is

$$\forall e, \Gamma, \tau : \Gamma \vdash e : \tau \implies (\exists v.e = v) \vee (\exists e'.e \rightsquigarrow e')$$

Preservation, on the other hand, states that if a term is typed and reduced for one step, then the reduct can be typed with the same term. That means that the new term has the same properties (as abstracted by the type) as the original term. A sample formulation is

$$\forall e, e', \Gamma, \tau : \Gamma \vdash e : \tau \wedge e \rightsquigarrow e' \implies \Gamma \vdash e' : \tau$$

Both statements are commonly proven by inductions. As both concern single steps, they are usually either over the structure of the term or the structure of the derivation of the typing.

Traditional soundness states that typed terms do not go wrong, that is, they do not get stuck during reduction. Given progress and preservation lemmas, this result can be deduced by induction over the length of the reduction sequence. If the sequence has length zero, then the term must be a value. Otherwise one must be able to reduce it by progress lemma, and the sequence could not have length zero. If the sequence is not empty, then it has a head and a tail. For the head step, we invoke preservation. This yields that the first element of the tail is typable with the same type. We can now invoke the inductive hypothesis.

Pottier and Simonet [72, 73] adapt this method to information flow control. The key difference

between traditional soundness and noninterference is that noninterference talks about *two* runs that cannot be distinguished. To make this syntactical, Pottier and Simonet extend Core ML, the base language they are treating which is based on ML, to Core ML^2, which has a bracket construct $\langle , \rangle$ that stands for two different computations at once. Now one can show progress and preservation over Core ML^2. Given a typing that says an expression is public, by progress and preservation a final result value must also be public. Brackets, however, cannot by typed low. It follows that the two encoded computations cannot have different results.

Chapter 3: FRAMEWORK

The content of this chapter is based on [40].

3.1 Introduction

To manage the rising complexity of creating software, systems are routinely created out of (mostly) independent components where each component can be specified, implemented and verified separately. In the language domain, one example for this is the composition of fragments from different languages into a complete program. A standard use case is the separation of storage and program logic concerns by embedding SQL queries into program code.

In parallel with the need to cope with this rising complexity, privacy is becoming an increasingly important property of software. Utilizing a security type system is one way to formally and soundly verify software against privacy policies and enforce properties like noninterference [43]. Noninterference ensures that any compliant program cannot leak private information to public channels. Many such type systems exist (e.g., [89, 46, 66, 93, 14, 48], for an overview see [78]) and the approach has been extended to cover entire distributed systems [61]. In all of this work, however, exactly one language is treated.

In contrast, how can we (statically) guarantee the safety of programs that are composed from elements in different languages? We propose to compose security-typed languages into *composed languages*, such that well-typed programs in a composed language can be guaranteed to comply with noninterference. This chapter studies an approach that, under certain assumptions, makes it possible to leverage proofs of non-interference of well-typed host language and well-typed embedded language programs to prove noninterference of well-typed composed language programs. In order to generalize this composition over security-typed host and security-typed embedded languages that use different proofs that well-typed programs are noninterferent, our approach relies on host languages being complete with respect to being able to compute any noninterferent function over its data types. This allows us to establish that executing noninterferent code does not

introduce any behaviors that could not be observed in the host language.

This chapter also validates the approach on a composition of a security-typed while language and a simple security-typed SQL fragment. We demonstrate a constructive technique to prove completeness with respect to noninterfering computations on the example of the while language, and formally prove all requirements to complete our framework approach.

The contributions of this chapter include the description of a general framework for composition of security-typed languages, such that noninterference of the composed language can be established from the proofs of noninterference of the component languages; showing how two notions of declassification fit into and can be enforced by the framework; and demonstrating the framework on a composition of a security-typed while language.

This chapter is structured as follows. In Section 3.2 we detail our goals of composition on the example of a student information system. Our framework approach is outlined in Section 3.4. Section 3.6 describes a case study, in which we compose a while language with a security-typed SQL fragment.

3.2 Motivational Example

Our ultimate goal is to *prove* safe the composition of practical languages. We use the example of a system composed of application logic and backend storage here. The application logic is written in an imperative language, while the storage is accessed with queries written in a SQL dialect. For example, imagine a university system that stores students' data. We can model this with a table that stores a record for each student, e.g., the student's name, room number, and several grades. Mandated by law, the grades are private information and must not be shared with unauthorized personnel, while room numbers can be used in a university directory. We can model this security by assigning low confidentiality to the name and room number, and high confidentiality to the grades. Now general staff can be classified as low, too, so to be able to access a student's room number. We can write a program that reads the database and writes this information to a low output, e.g., a generally accessible website. Note that `eval` will process the nested query and return the

result. In a more practical language, the explicit use of this construct may be hidden by a layer of syntactic sugar.

```
Program list-students;
Schema: students: name=L, room=L, grade1=H;
Code:
 length{L} = eval("SELECT count(*) FROM students");
 i{L} = 0;
 while i < L do
  name{L},room{L},grade1{L} = eval("SELECT name,room, grade1
    FROM students LIMIT $x,$x",i);
  print-public name, room, grade1
  i++;
```

This program should be rejected because of the leak of the grade, namely that the level in the host language(L) does not correspond with the level in the embedded language(H). However, if that part is removed, the program is valid and should pass the information flow checker. Similarly, updates in the database need to be protected, e.g., the following program needs to be rejected.

```
Program update-room;
Schema: students: name=L, room=L, grade1=H;
Code:
 grade1{H} = eval("SELECT grade1 FROM students WHERE name='...'");
 if grade1>3 then
  eval("UPDATE room=3 WHERE name='...'");
```

Assume that our SQL dialect is proven secure (see Section 6.2), as well as the While language (e.g., [89]). We would now like to prove that the composed language is also secure.

3.3 Proof Manipulation vs Framework Approach

Traditional approaches to prove the soundness of a security type system, that is, the formal guarantee that a typable program is secure, have been outlined in the chapter on background material. A first question that this dissertation intends to answer is whether there are shortcomings in the traditional approach when applied to composed languages. Our *positive* answer will be the motivation to introduce a composition framework.

Given a concrete host language and a concrete embedded language that we want to compose, at a first glance the problem seems to be straightforward: we first need to define how an embedding syntactically appears in the (extended) host language and then how it semantically interacts with the other constructs. For simplicity, we introduce a new constructor for an abstract syntax element, and give it intuitive semantics and typing rules.

The interesting part is now the formal *proof* of noninterference for our extended language. In the first and better case, we have a *complete* formal proof for the host language available. We can then study the proof approach. It is likely to include several inductions, both in the main statement and several auxiliary lemmas (see Background). If the induction is related to the program in any way, either directly by being over the structure of a program, or indirectly by being over entities connected to a program like typings or semantics, then we need to extend the cases handled by the induction.

If the statements are known ahead of time, for example, the statement of progress and preservation have a default form, then we might try to generically create the induction case for the `eval` constructor. This seems possible as we define syntax, semantics and typing for that element.

There are two technical problems with this approach. They both arise from it being possible that simply adding another case to this induction does not form a valid, complete proof.

Auxiliary Lemmas The approach breaks down once auxiliary lemmas are used. It is in general impossible to know ahead of time what auxiliary statements the main proof needs. A common lemma is Substitution, arguing that the substitution of a variable for a typed term of the same type

preserves the typing. This should be itself an induction again, so a case for `eval` is again necessary. However, other lemmas may be necessary. [72], for example, also requires at least a weakening and a projection lemma, whereas [89] uses, among others, lemmas that establish state invariants like domain preservation.

Nested Case Analysis or Induction Under a certain viewpoint, lemmas are modular components for the whole proof to improve reuse and readability. One might argue that lemmas could thus be "inlined" into the main proof. If the lemmas are inlined, we do not have the problem of unknown lemma statements that cannot be handled without detailed proof knowledge and understanding.

However, inlining lemmas will lead to problems of the second category: nested case analysis. If we, for example, inline a substitution lemma, all top-level induction cases that used this lemma will now explicitly include a nested induction. This means that cases are *not* cleanly isolated anymore: it is not enough to simply add the `eval` case. All other cases need to be inspected and potentially fixed up, if the new `eval` construct might appear.

Please note that inlining lemmas is not required to raise this problematic case. The problem already appears if there is a rule for the host language that inspects the children of an element, that is, if it doesn't treat the child elements opaquely. This can for example happen if the abstract syntax is not flat, that is, it contains wrapper elements.

Even if we could solve these technical problems, two challenges remain: First, it is necessary to *understand* the proofs of host language and embedded language noninterference. As such formal proofs are intricate and specialized, we expect only a small number of people, experts in the domain of security-typed languages, to be able to make the necessary extensions.

Second, we now have a formal proof of the composition of host language A and embedded language B. It is not obvious if and how much of that effort can be reused when combining language A with another embedded language C.

As a result, it seems impossible to devise a generic proof manipulation strategy that works in

all cases; and there is considerable overhead and required knowledge involved in concrete proof manipulation that this approach seems not practical.

Our solution is to treat the original proofs as a black box, and derive a proof independently. But note that we do not have to re-prove everything. Instead, we try to reuse the already proven statements about the component languages The trick lies in reducing the composed problem to a purely-host problem, and argueing in the host-language space.

3.4 Framework for Composition

To establish that well-typed composite program are secure, that is, noninterfering, we view composite programs as host-language contexts (a context is a program with holes), where all holes have been filled with some eval statements. Our goal is to show that we can fill the context with well-typed host program fragments such that each fragment matches the effect of the corresponding function implemented in the embedded language, and the new complete program remains well-typed. If this is possible, then the composite program has an equivalent well-typed host program, which is guaranteed to be noninterfering by the host-level type system guarantees. It follows that the composite program must be noninterfering, too. The following subsections describe sufficient requirements guaranteeing such a transformation.

3.4.1 Eval Setup

For our framework we assume a certain computational setup for the composed language. The flow of computation is, in idealized form, as follows.

1. A host parameter is evaluated and reduced to a *value*
2. The host value is translated to an embedded value
3. The embedded computation is run with the embedded value as input
4. The output of the embedded computation is translated to a host result

We can formalize this in a functional interpretation as $\gamma \circ f_E \circ \alpha \circ f_H$, where f_H is the host parameter evaluation, f_E is the embedded computation, and α and γ are so-called *transfer functions*. Transfer functions translate values between the two languages: α in a forward (host $\rightarrow$ embedded) direction, and γ in a backward (embedded $\rightarrow$ host) direction.

We also require a certain shape of typing of an `eval` statement or expression. Informally, an `eval` should be typable only if the embedded computation can be typed with types corresponding to the host's input and output types. An abstract formalization that covers all type systems seems not possible, as they differ too widely in syntax and semantics, and the specific side conditions to enforce noninterference.

As such, we only give an incomplete example here. Following the above computational view of $f_{eval} = \gamma \circ f_E \circ \alpha \circ f_H$, we assume that a typing is established by $f_{eval} : \tau_i \rightarrow \tau_o = (\gamma^\tau : \tau_r \rightarrow \tau_o) \circ (f_E : \tau_e \rightarrow \tau_r) \circ (\alpha^\tau : \tau_h \rightarrow \tau_e) \circ (f_H : \tau_i \rightarrow \tau_h)$. The functions α^τ and γ^τ are *type transfer functions*: they relate types (or just security levels) of the host language and the embedded language.

For a type-based enforcement, transfer functions and type transfer functions need to be suitably related.

Requirement 1 (Transfer Functions Requirement)**.** *The transfer functions α and γ and type transfer functions α^τ and γ^τ of a language composition need to satisfy the following constraints.*

(a) The type transfer functions are monotonous.

$$\forall \tau_1, \tau_2.\ \tau_1 \sqsubseteq_H \tau_2 \implies \alpha^\tau(\tau_1) \sqsubseteq_E \alpha^\tau(\tau_2)$$
$$\forall \tau_1, \tau_2.\ \tau_1 \sqsubseteq_E \tau_2 \implies \gamma^\tau(\tau_1) \sqsubseteq_H \gamma^\tau(\tau_2)$$

(b) Composition of type transfer functions is non-decreasing.

$$\forall \tau.\ \tau \sqsubseteq (\gamma^\tau \circ \alpha^\tau)(\tau)$$
$$\forall \tau.\ \tau \sqsubseteq (\alpha^\tau \circ \gamma^\tau)(\tau)$$

(c) Transferred values can be typed according to the transferred type.

$$\forall v, \tau.\ \vdash_H v : \tau \implies \vdash_E \alpha(v) : \alpha^\tau(\tau)$$
$$\forall v, \tau.\ \vdash_E v : \tau \implies \vdash_H \gamma(v) : \gamma^\tau(\tau)$$

(d) If input values are indistinguishable, then transferred values are indistinguishable.

$$\forall v_1, v_2, \tau.\ v_1 \sim_\tau v_2 \implies \alpha(v_1) \sim_{\alpha^\tau(\tau)} \alpha(v_2)$$
$$\forall v_1, v_2, \tau.\ v_1 \sim_\tau v_2 \implies \gamma(v_1) \sim_{\gamma^\tau(\tau)} \gamma(v_2)$$

The requirement 1(a) ensures that if a type is considered more confidential in the source language than another, then the corresponding target language type is also more confidential than the other. Otherwise, a translation to the other language would be a declassification. The requirement 1(b) ensures that just passing a value around cannot inadvertently launder, that is, declassify it. Requirement 1(c) ensures typability of transferred values. Without this requirement, type-based enforcement is not possible. Finally, requirement 1(d) ensures that transferring values preserves indistinguishability. Else a simple transfer will leak information.

These requirements are carefully chosen. In a composed language with transfer functions and type transfer functions that satisfy requirement 1 we can abstractly show the following lemma.

Lemma 2 (Transfer Functions Imply Noninterference)**.** *Given a typed* `eval` *statement with a functional interpretation and typing given by* $f_{eval} : \tau_i \to \tau_o = (\gamma^\tau : \tau_r \to \tau_o) \circ (f_E : \tau_e \to \tau_r) \circ (\alpha^\tau : \tau_h \to \tau_e) \circ (f_H : \tau_i \to \tau_h)$*, and the transfer functions and type transfer functions satisfy requirement 1, then the* `eval` *statement is noninterfering.*

$$\forall v_1, v_2.\ v_1 \sim_{\tau_i} v_2 \implies f_{eval}(v_1) \sim_{\tau_o} f_{eval}(v_2)$$

Proof. The proof of this is straightforward. The functional interpretation of `eval` is $\gamma \circ f_E \circ \alpha \circ f_H$. By typing and host-level noninterference, we have $v_1^h = f_H(v_1) \sim_{\tau_h} f_H(v_2) = v_2^h$. By requirement

1, it follows that $v_1^e = \alpha(v_1^h) \sim_{\alpha^\tau(\tau_h)=\tau_e} \alpha(v_2^h) = v_2^e$. As the embedded computation f_E is typed, embedded-level noninterference applies. This yields $v_1^r = f_E(v_1^e) \sim_{\tau_r} f_E(v_2^e) = v_2^r$. Finally, again by requirement 1, we have $v_1^o = \gamma(v_1^r) \sim_{\gamma^\tau(\tau_r)=\tau_o} \gamma(v_2^r) = v_2^o$. As $v_1^o = f_{eval}(v_1)$ and $v_2^o = f_{eval}(v_2)$, this concludes the proof. □

3.4.2 Simulation

Simulation is sufficient to find a host-language fragment that matches the effects of embedded language programs. Simulation requires that for each each function of interest computed in the embedded language, there exists a program in the host language that simulates its behavior. If $\mathcal{I} \subseteq \mathcal{L}_\mathcal{E}$ is the class of programs of interest, a subset of all programs of the embedded language, $\mathcal{L}_H$ is the class of all host programs, and $[\cdot]$ is the computed function of a program, we may formalize this as $\forall p \in \mathcal{I}.\ \exists p' \in \mathcal{L}_H.\ [\![p]\!] \equiv [\![p']\!]$, where $\equiv$ defines some equivalence over semantic functions. In short, the host language is computationally at least as powerful as the embedded language.

In practice, we consider host languages that are Turing-complete, that is, every Turing-computable function can be computed using a single security level, and the embedded language might be Turing-computable, that is, every embedded-language program computes a Turing-computable function. Then the host language can obviously simulate any embedded-language program. Note that any other specific level of computability is acceptable, as long as it allows covering all embedded-language behaviors.

For security-typed composite languages we are only interested in the class of noninterfering computations of the embedded language. If a composed program is typed, then the definition of `eval` and its typing rules ensure that the embedded code fragment is typable in the embedded language. This means that the embedded program is noninterfering by virtue of the embedded language's noninterference statement. Now since the simulation is required to compute an equivalent function, and noninterference is a semantic property that only relies on inputs and outputs, the simulation is also noninterfering. Note that this of course only holds if the interface established by `eval` satisfies constraints about the values and types involved, for example, preserving

indistinguishability of values.

Finally, note that only this step is specific to a certain composition. For each composition, it needs to be ensured that the embedded language is at most as powerful as the host language. All following steps are specific to the host language and can be reused for other compositions. However, we think that the simulation step is broadly applicable, because most host languages are expected to be Turing-complete general-purpose languages that are at the top of any realistic computability hierarchy.

Formalization

We start formalizing our simulation requirements by defining simulations.

Definition 3 (Simulation)**.** *A simulation S is a function mapping* `eval` *statements to pure-host statements. We write $S[c]$ for the simulation of* `eval` *statement c.*

Informally, a simulation is correct if it maps an `eval` statement to a functionally equivalent pure-host statement. The eval statement and pure-host statement are formally related by an $\mathrm{encodes}$ predicate connecting a partial host state and the embedded state. This abstracts that a simulation needs to encode the embedded state. Correctness then maps execution of the original `eval` statement and the simulation.

Definition 4 (Correct Simulation)**.** *A simulation S is correct iff* $\mathrm{encodes}$ *is a bisimulation: Given μ, μ', ν, ν' and* `eval` *statement c where $\mu, \nu, c \Downarrow \mu', \nu'$, there exist μ_e and μ'_e such that* $\mathrm{encodes}(\mu_e, \nu)$*, $\mu \oplus \mu_e, S[c] \Downarrow \mu' \oplus \mu'_e$ and* $\mathrm{encodes}(\mu'_e, \nu')$*.*

Finally, the framework requirement is that there is a correct simulation.

Requirement 5 (Correct Simulation)**.** *There exists a correct simulation S for the composed language.*

3.4.3 Typability

While we now know that the simulation itself is noninterfering, it remains to be seen whether this holds for the whole program the simulation is a part of. We reduce this problem to two steps, the first of which is typability. If we can show that the simulation is typable, host-level noninterference will apply and secure the computation.

Note that the framework is a theoretical tool to prove composed languages secure. As such, the actual simulation is not important for running the composed program. This means that we can use *any* simulation that is equivalent to the embedded code fragment, no matter how complex, if it helps us find a typing.

Definition 6 (Environment Mapping)**.** *An environment mapping is a function T from embedded-language typing environments Δ to host-language typing environments Γ.*

Definition 7 (Type-Correct Simulation)**.** *A simulation S with associated encoding* encodes *is type-correct with respect to environment mapping T iff*

1. $\forall \mu, \nu, \Delta.\ \Delta \vdash \nu \wedge \mathrm{encodes}(\mu, \nu) \implies T(\Delta) \vdash \mu$
2. $\forall \Gamma, \Delta, c.\ \Gamma, \Delta \vdash c : \tau \implies \Gamma \oplus T(\Delta) \vdash S[c] : \tau$

Definition 8 (Indistinguishability-Preserving Encoding)**.** *An encoding* encodes *is indistinguishability-preserving with respect to environment mapping T iff indistinguishable state pairs are mapped to indistinguishable state pairs:*

$$\forall \Delta, \mu_1, \mu_2, \nu_1, \nu_2.\ \mathrm{encodes}(\mu_1, \nu_1) \wedge \mathrm{encodes}(\mu_2, \nu_2) \implies (\mu_1 \sim_{T(\Delta)} \mu_2 \iff \nu_1 \sim_{\Delta} \nu_2)$$

We call a simulation S indistinguishability-preserving if its encoding is indistinguishability-preserving.

Definition 9 (Typable Encoding)**.** *An encoding μ_e of ν (i.e.,* $\mathrm{encodes}(\mu_e, \nu)$*) is typable with Γ_e with respect to Δ iff $\Delta \vdash \nu$ and $\Gamma_e \vdash \mu_e$ and for all ν' such that $\Delta \vdash \nu'$ and $\nu \sim_{\Delta} \nu'$ we have for all μ' with* $\mathrm{encodes}(\mu', \nu')$ *that $\Gamma_e \vdash \mu'$ and $\mu_e \sim_{\Gamma_e} \mu'$.*

This definition ensures that a typable encoding complies with the indistinguishability of the embedded state. Informally, a pair of indistinguishable embedded-level states should be simulated by a pair of indistinguishable host-level states.

Definition 10 (Typable Simulation)**.** *A correct simulation S is typable with respect to T, denoted by S_T, iff S is type-correct and indistinguishability-preserving with respect to T.*

Finally, the framework requirement is that there is a typable simulation.

Requirement 11 (Typable Simulation)**.** *The correct simulation S of the framework is typable.*

For technical reasons, it can be easier to actually show a stronger property for the host language.

Definition 12 (Security Completeness)**.** *A security-typed language $\mathcal{L}$ is security-complete if and only if for every program c that computes a function f, where f is noninterfering with respect to security signature $\mathcal{S}$, there exists a program c' that also computes f and is typable corresponding to $\mathcal{S}$.*

In the case of a security-complete host language, the typability step is immediately obvious: A correct simulation preserves noninterference. In a security-complete language, this guarantees the existence of a typable simulation. The While language we investigate here is security-complete, as we will show in Theorem 71.

Note that all practical general-purpose security-typed languages seem to be security-complete for functions over integers. In Chapter 4 we study the limits of security-completeness and basic requirements on a security type system. There we show sufficient conditions for versions of security-completeness for functional languages with nonrecursive and recursive algebraic datatypes, as well as languages with heaps and side effects including a simple class-based object-oriented language.

3.4.4 Replacement

Last, if we can replace all instances of `eval` in the composite program with the corresponding simulations, and can show that typings and meaning are properly preserved, we have found a pure

host-language program that is host-typable, which implies its noninterference. Again, noninterference can then be reflected onto the composite program, which is functionally equivalent. This is the third step, which is also only host-language specific.

Note that this is not a traditional substitution lemma. Standard substitution lemmas are concerned with replacing a variable with a term, and preserving a typing in the process. In our case, a whole construct, namely `eval`, must be replaced, and the typing of the replaced term may be under a different environment to account for the possible temporaries and side effects of the simulation. Furthermore, one has to show that the programs before and after substitution are functionally equivalent. However, as mentioned, this has to be proven only once and can be reused for other compositions involving this host language.

Formalization

Requirement 13 (Replacement Semantics)**.** *If* $c = E[\overrightarrow{e}]$ *with* $\overrightarrow{e}$ `eval` *statements, and* $\overrightarrow{s}$ *are correct simulations of* $\overrightarrow{e}$*, then*

$$\begin{aligned}&\forall \mu, \mu', \mu_e, \mu'_e, \nu, \nu.\\ &\quad \mu, \nu, E[\overrightarrow{e}] \Downarrow \mu', \nu' \wedge \mathrm{encodes}(\mu_e, \nu) \wedge \mathrm{encodes}(\mu'_e, \nu') \implies\\ &\quad\quad \mu \oplus \mu_e, E[\overrightarrow{s}] \Downarrow \mu' \oplus \mu'_e\end{aligned}$$

This requirement formalizes the notion that replacing all `eval` instructions should create a simulation of the whole composed program, that is, the transformed program satisfies the correctness requirements.

Requirement 14 (Replacement Typing)**.** *If* $c = E[\overrightarrow{e}]$*,* $\Gamma, \Delta \vdash c : \tau$ *and for all elements of* $\overrightarrow{e}$ *we have* $\Gamma_i, \Delta \vdash e_i : \tau_i$ *in the derivation of* $\Gamma, \Delta \vdash c : \tau$*, then for all* $\overrightarrow{s}$ *and* Γ' *with* $\Gamma_i \oplus \Gamma', \Delta \vdash s_i : \tau_i$ *for all elements* s_i *of* $\overrightarrow{s}$ *we have* $\Gamma \oplus \Gamma', \Delta \vdash E[\overrightarrow{s}] : \tau$*.*

This requirement formalizes our extended substitution. If one replaces an `eval` statement under an environment extended with the encoding environment, the program remains typed.

Requirement 15 (Reduction to Host)**.** *If* $\Gamma, \Delta \vdash c : \tau$*, and* c *does not contain* `eval` *statements, then* $\Gamma \vdash c : \tau$*.*

Finally, to be able to argue with host-level noninterference for a pure-host composed-language program, we have to be able to derive a host-level typing. Note that this requirement is easily shown if the composed language was generated as in Section 3.4.1.

3.4.5 Completing the Framework Approach

Finally, we show abstractly how the previous requirements ensure that soundness of the composed language.

Theorem 16 (Framework Correctness)**.** *If a composed language satisfies requirements 5, 11, 13, 14, and 15, then its security type system is sound.*

Proof. Assume $\Gamma, \Delta \vdash c : \tau$. Let S be the correct, typable simulation guaranteed by requirements 5 and 11. If c does not contain `eval` statements, then by Requirement 15 we have $\Gamma \vdash c : \tau$, so c is noninterfering by host language noninterference.

Now assume that c does contain `eval` statements. Let $\overrightarrow{e}$ denote the list of `eval`s, and let $E[\bullet]$ be the context such that $E[\overrightarrow{e}] = c$. As c is typed, each `eval` in c is typed. Then by definition of correctness, we have a list $\overrightarrow{s}$ of simulations generated by S. Let Γ_e be the typing of the encoding as given by simulation typability. Note that Γ_e is shared by all elements of $\overrightarrow{s}$, as they are typed over the same embedded state typing Δ. By simulation typability, for all corresponding elements e_i and $s_i = S[e_i]$ we have that if $\Gamma', \Delta \vdash e_i : \tau_i$, then $\Gamma' \oplus \Gamma_e, \Delta \vdash s_i : \tau_i$. By requirement 14, this implies $\Gamma \oplus \Gamma_e, \Delta \vdash E[\overrightarrow{s}] : \tau$. That is, replacing all `eval` statements with corresponding simulations yields a typable program, where the typing is related to the original judgment. Now according to requirement 15, as all `eval` statements have been replaced, it follows that we have a pure host-level typing judgment $\Gamma \oplus \Gamma_e \vdash E[\overrightarrow{s}] : \tau$. Thus, the program with simulations $E[\overrightarrow{s}]$ is guaranteed to be noninterfering by the host-level noninterference statement.

Analogously, requirement 13 ensures that the program with simulations $E[\overrightarrow{s}]$ performs a re-

lated computation. The requirement guarantees that

$$\forall \mu, \mu', \mu_e, \mu'_e, \nu\nu'.\ \mu, \nu, c \Downarrow \mu', \nu' \wedge \mathrm{encodes}(\mu_e, \nu) \implies \mu \oplus \mu_e, E[\overrightarrow{s}] \Downarrow \mu' \oplus \mu'_e \wedge \mathrm{encodes}(\mu'_e, \nu')$$

As the simulation program $E[\overrightarrow{s}]$ is noninterfering, computes a related function, and $\mathrm{encodes}$ preserves indistinguishability, it follows that the composed program c is noninterfering. □

3.5 Applicability

One might ask if embedding has a purpose if the embedded code is required to be able to be simulated in the host. In practice many embedded languages are only powerful in certain domains (i.e., domain specific languages) and excel in conciseness and expressivity there, while general-purpose languages are usually Turing-complete and can do the same work. Most embedded languages show their advantages in the conciseness and expressivity in just this limited domain. For example, SQL is a query language for databases and is (in its basic incarnation) not Turing-complete, but can describe a complex set of relational queries with a relatively small amount of code. The SQL semantics could be simulated precisely in a general-purpose language, albeit with a lot of simulation overhead. Thus, SQL can clarify the meaning and intention of some part of a program, improving that and only that part over a general-purpose implementation. Furthermore, most host languages are general-purpose languages that are Turing-complete and thus as powerful as realistically possible.

Though we require a simulation exists, we would like to stress that its efficiency does not effect the practicality of out framework. The simulation is a tool for guaranteeing noninterference of the extended semantics of the composed language. Thus, the size of the simulation would not matter, since it would *not* be used in practice.

Another relevant question is whether our requirements are too strong. Namely, we require noninterferent input programs. Type systems are still important once one already knows that a function complies with noninterference. For example, a typing can act as a certificate for a program, such

that remote clients can check for actual noninterference. Also, our overall goal tries to formally establish the safety of a composed language from its components. Typing for a known noninterferent embedded program still needs to be liftable to the overall language, which our approach provides.

3.6 Case Study

This section formalizes a case study for applying our framework. In Subsection 3.6.1 we introduce While. The following subsection sketches our fragment of SQL, gives its type system, and states noninterference. Subsection 3.6.3 formalizes the composed language. In the last subsection, a full development of the proofs required for applying the framework to the composed language can be found.

3.6.1 Host: WHILE

We base our exposition on a simplified version of the while language in [89]. This is a simple While language, with the following expressions and statements.

$$e ::= n \mid x \mid e \odot e \qquad c ::= x := e \mid c; c' \mid \text{if } e \text{ then } c \text{ else } c' \mid \text{while } e \text{ do } c$$

The language only supports integer values. A state μ binds variables to integers. We use a natural semantics. Expressions are assumed side-effect free, and evaluated by $\mu(e)$.

$$\mu(e) = \begin{cases} n & e = n \\ \mu(x) & e = x \\ \mu(e_1)[\![\odot]\!]\mu(e_2) & e = e_1 \odot e_2 \end{cases}$$

As is usual, $\mu[x := v]$ denotes the state μ' that is identical to μ except for mapping x to v:

$$\mu[x := v](y) = \begin{cases} v & y = x \\ \mu(y) & \text{else} \end{cases}$$

The semantics connects an input state and a statement with an output state and is fully standard.

$$\frac{}{\mu, x := e \Rightarrow \mu[x := \mu(e)]}\ S\text{-}Ass \qquad \frac{\mu, c \Rightarrow \mu' \quad \mu', c' \Rightarrow \mu''}{\mu, c; c' \Rightarrow \mu''}\ S\text{-}Seq$$

$$\frac{\mu(e) \neq 0 \quad \mu, c \Rightarrow \mu'}{\mu, \text{if } e \text{ then } c \text{ else } c' \Rightarrow \mu'}\ S\text{-}IfT \qquad \frac{\mu(e) = 0 \quad \mu, c' \Rightarrow \mu'}{\mu, \text{if } e \text{ then } c \text{ else } c' \Rightarrow \mu'}\ S\text{-}IfF$$

$$\frac{\mu(e) \neq 0 \quad \mu, c \Rightarrow \mu' \quad \mu', \text{while } e \text{ do } c \Rightarrow \mu''}{\mu, \text{while } e \text{ do } c \Rightarrow \mu''}\ S\text{-}WhileT \qquad \frac{\mu(e) = 0}{\mu, \text{while } e \text{ do } c \Rightarrow \mu}\ S\text{-}WhileF$$

Since the language only supports integers, it is not necessary to have a ground type system. A typing environment Γ binds variables to security levels. Judgments have the form $\Gamma \vdash e : \ell$ and $\Gamma \vdash c : \ell\,\text{ok}$ and are also standard.

$$\frac{}{\Gamma \vdash n : \ell}\ T\text{-}Lit \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}\ T\text{-}Var$$

$$\frac{\Gamma \vdash e : \ell \quad \Gamma \vdash e' : \ell}{\Gamma \vdash e \odot e' : \ell}\ T\text{-}Exp \qquad \frac{\Gamma \vdash e : \ell \quad \ell \sqsubseteq \ell'}{\Gamma \vdash e : \ell'}\ T\text{-}ESub$$

$$\frac{\Gamma \vdash x : \ell \quad \Gamma \vdash e : \ell' \quad \ell' \sqsubseteq \ell}{\Gamma \vdash x := e : \ell}\ T\text{-}Ass \qquad \frac{\Gamma \vdash c : \ell\,\text{ok} \quad \Gamma \vdash c' : \ell\,\text{ok}}{\Gamma \vdash c; c' : \ell\,\text{ok}}\ T\text{-}Seq$$

$$\frac{\Gamma \vdash e : \ell \quad \Gamma \vdash c : \ell\,\text{ok} \quad \Gamma \vdash c' : \ell\,\text{ok}}{\Gamma \vdash \text{if } e \text{ then } c \text{ else } c' : \ell\,\text{ok}}\ T\text{-}If \qquad \frac{\Gamma \vdash e : \bot \quad \Gamma \vdash c : \ell\,\text{ok}}{\Gamma \vdash \text{while } e \text{ do } c : \ell\,\text{ok}}\ T\text{-}While$$

$$\frac{\Gamma \vdash c : \ell\,\text{ok} \quad \ell \sqsubseteq \ell'}{\Gamma \vdash c : \ell'\,\text{ok}}\ T\text{-}SSub$$

Two states are indistinguishable if they agree on all observable variables: $\mu_1 \sim_{\Gamma,\ell} \mu_2 \iff \forall x.\Gamma(x) \sqsubseteq \ell \implies \mu_1(x) = \mu_2(x)$. Noninterference states that a typed computation, when started with indistinguishable states, results in indistinguishable states. For a proof we refer to [89] or the sketches in the background chapter.

$$\forall \ell, \Gamma, c, \mu_1, \mu_2, \mu_1', \mu_2'.\Gamma \vdash c : \ell \,\mathrm{ok} \wedge \mu_1 \sim_{\Gamma,\ell} \mu_2 \wedge \mu_1, c \Rightarrow \mu_1' \wedge \mu_2, c \Rightarrow \mu_2' \implies \mu_1' \sim_{\Gamma,\ell} \mu_2'$$

3.6.2 Embedded Language: SQL

As an embedding, we use an expressive fragment of SQL, the Structured Query Language. The language, its syntax and semantics and its secure type system are detailed in Section 6.2.

For composition purposes, the details of said language are not important. We only stress two points here. First, terms and variables in the language are denoted by $\dot{c}$ and $\dot{x}$, with the dot clearly separating them from host-level statements and variables. The embedded state is denoted by ν, and a typing environment by Δ.

Second, the embedded type system makes judgments of the form $\Delta \vdash \dot{c} : \ell_\bullet^{\ell_r}/\ell_s \,\mathrm{ok}$. The type $\ell_\bullet^{\ell_a}/\ell_s$ is to be read as:

- The result of this query are rows with columns typed according to $\ell_\bullet$, that is, column one is typed according to ℓ_1, column two is typed according to ℓ_2, and so on.
- The overall result is typed additionally with a confidentiality level of ℓ_r.
- The query has side effects with a lower bound of ℓ_s.

3.6.3 Composed Language

This section formalizes the extension of the host language, to create a composed language of WHILE and SQL. This consists of adding an evaluation statement type, extending the semantics, and giving it a typing that allows reasoning from the components' noninterference theorems. We

restrict ourselves to transfer simple integers. We add the statement $x_1, \ldots, x_n := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$ for evaluation, where x_i designate the variables that will hold the result, and x' is a host-level variable that is a parameter for the embedded computation $\dot{c}$. We will use $x_\bullet$ to denote a list of variables.

We extend the host semantics in the following two ways. First, the embedded state ν is threaded through the original reduction rules, which do not update the embedded states themselves. In the case of WHILE, this is unambiguous. As an example, the sequence rule update looks like the following.

$$\frac{\mu, c_1 \Rightarrow \mu' \quad \mu', c_2 \Rightarrow \mu''}{\mu, c_1; c_2 \Rightarrow \mu''} \quad \rightarrow \quad \frac{\mu, \nu, c_1 \Rightarrow \mu', \nu' \quad \mu', \nu', c_2 \Rightarrow \mu'', \nu''}{\mu, \nu, c_1; c_2 \Rightarrow \mu'', \nu''}$$

In languages with multiple nested reductions this threading will induce a certain evaluation order.

Second, reduction of eval is given by the following new rules.

$$\frac{\nu, \dot{c}[\dot{x} := \mu(x')] \Rightarrow \nu', \emptyset}{\mu, \nu, x_\bullet := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c} \Rightarrow \mu[x_1 := 0][\ldots][x_n := 0], \nu'}$$

$$\frac{\nu, \dot{c}[\dot{x} := \mu(x')] \Rightarrow \nu', s \quad s \neq \emptyset \quad s(0) = (n_1, \ldots, n_n)}{\mu, \nu, x_\bullet := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c} \Rightarrow \mu[x_1 := n_1][\ldots][x_n := n_n], \nu'}$$

Typing rules are changed accordingly. That is, Δ is threaded through the original typing rules, and statement types are extended by the lower bound of changes in the embedded state. Furthermore, we add the following typing rule for eval.

$$\frac{\Gamma \vdash x' : \ell \quad \Delta, [\dot{x} : \ell] \vdash \dot{c} : \ell_\bullet^{\ell_2} / \ell_s \,\mathrm{ok} \quad \forall i. \ell_i \sqcup \ell_2 \sqsubseteq \Gamma(x_i)}{\Gamma, \Delta \vdash x_\bullet := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c} : (\bigsqcap \Gamma(x_i)) \sqcap \ell_S \,\mathrm{ok}}$$

Indistinguishability is lifted component-wise, that is, $\mu_1, \nu_1 \ddot{\sim}_{\Gamma,\Delta,\ell} \mu_2, \nu_2 \iff \mu_1 \sim_{\Gamma,\ell} \mu_2 \wedge \nu_1 \dot{\sim}_{\Delta,\ell} \nu_2$. This suffices because SQL values, that is, result sets, do not occur as values in the

composed language. Then noninterference is stated analogously to the WHILE case.

$$\forall \ell, \Gamma, c, \mu_1, \nu_1, \mu_2, \nu_2, \mu_1', \nu_1', \mu_2', \nu_2'.$$
$$\Gamma \vdash c : \ell \,\mathrm{ok} \wedge \mu_1, \nu_1 \ddot{\sim}_{\Gamma,\Delta,\ell} \mu_2, \nu_2 \wedge \mu_1, \nu_1, c \Rightarrow \mu_1', \nu_1' \wedge \mu_2, \nu_2, c \Rightarrow \mu_2', \nu_2'$$
$$\Longrightarrow \mu_1', \nu_1' \ddot{\sim}_{\Gamma,\Delta,\ell} \mu_2', \nu_2'$$

3.6.4 Proofs

This section applies our framework approach outlined in section 3.4 and formally verifies that the composition of While and SQL from the previous section is safe, that is, typable composed programs are noninterfering. For this, recall the main steps:

1. Embedded-language programs can be simulated in the host (Requirement 5)
2. The simulation is noninterferent and can be typed (Requirement 11)
3. Replace embeddings with the simulation; the now pure-host program is typable, implying noninterference of the composed program (Requirements 13 & 14)

Simulation

For the first step, we need to establish that the host language is computationally at least as powerful as the embedded language. In our case, we note that WHILE is Turing-complete, and SQL is Turing-computable. Thus, any program $\dot{c}$ can be simulated by a program c.

For the further steps, we also need to argue about the shape of the simulation. This is important because the simulation works with encodings, at least of the embedded-level state. Most generally, the database might be encoded into a single integer. Then functionally, the embedded part of eval corresponds to an integer function with two inputs and two outputs. However, WHILE's type system cannot give the database, thus encoded, a correct type, as all columns are collapsed into a single number.

Instead, we use multiple integers to encode the embedded state. Namely, each column is en-

coded into an integer. For example, we could use Church's encoding of lists into nested pairs, and then a pairing function like that of Cantor. The number of necessary integers is known statically, since the database "schema" is known. Now each column representation can be given the level of the corresponding column in the database.

Traversing the database can be implemented as iterative decoding and unpacking of the head pair, and other list manipulations can be implemented in that fashion, too. select can be implemented by traversing the database, evaluating the condition, potentially computing and adding the result. update iterates, checks the condition and updates the current pair. insert can simply add a new entry at the head of the lists, since the order does not matter, while delete deletes corresponding elements in all columns.

Formalization For simplicity, we assume that for any given composed program, the names of the columns, as described by Δ, do not appear in the program. We can then define the encoding μ_e of embedded state ν as $\mu_e(t.c) = \mathrm{encode\text{-}column}(\nu, t, c)$, where t ranges over table names and c over respective columns. μ_e also contains space for temporaries necessary in the simulation of operations.

The code fragments in Figure 3.1 through Figure 3.4 show the simulations of SQL statements. The first simulation (Figure 3.1) is of a simple select statement of the form $\mathrm{select}\ \dot{e}\ \mathrm{from}\ \dot{t}\ \mathrm{where}\ \dot{e}'$ over our encoding. The code assumes that table $\dot{t}$ has columns $c_1,\ldots,c_n$. empty_list is the integer value encoding an empty list, while head, tail and append are macros for the list operations over an encoding. Finally, expr is a macro for SQL expression evaluation. As our languages agree on integer computations, this is a straightforward translation, replacing column references with the corresponding temporary variables, e.g., c_1 with $\mathrm{h\_c}_1$.

The second code fragment (Figure 3.2) simulates insertion. The code follows our semantics, where one column is filled with the given value, and all other columns of the new row are initialized to zero. The third code fragment (Figure 3.3) simulates update. The code iterates over the encoding, rebuilding the storage and updating a column when necessary. The fourth and last

```
result := empty_list;
tmp_c1 := t_c1; tmp_c2 := t_c2; ...
while tmp_c1 <> empty_list do
  h_c1 := head(tmp_c1); h_c2 := head(tmp_c2); ...
  tmp_c1 := tail(tmp_c1); tmp_c2 := tail(tmp_c2); ...
  tmp_e' := expr(ė', h_c1, h_c2, ...);
  if tmp_e' <> 0 then
    tmp_e := expr(ė, h_c1, h_c2, ...);
    result := append(result, tmp_e);
```

Figure 3.1: select simulation

```
tmp_e := expr(ė);
t_ci := append(t_ci, tmp_e);
t_c1 := append(t_c1, 0); ...
```

Figure 3.2: insert simulation

```
tmp_c1 := t_c1; tmp_c2 := t_c2; ...
t_c1 := empty_list; t_c2 := empty_list; ...
while tmp_c1 <> empty_list do
  h_c1 := head(tmp_c1); h_c2 := head(tmp_c2); ...
  tmp_c1 := tail(tmp_c1); tmp_c2 := tail(tmp_c2); ...
  tmp_e' := expr(ė', h_c1, h_c2, ...);
  if tmp_e' <> 0 then
    h_ci := expr(ė, h_c1, h_c2, ...);
  t_c1 := append(t_c1, h_c1); ...
```

Figure 3.3: update simulation

```
tmp_c1 := t_c1; tmp_c2 := t_c2; ...
t_c1 := empty_list; t_c2 := empty_list; ...
while tmp_c1 <> empty_list do
  h_c1 := head(tmp_c1); h_c2 := head(tmp_c2); ...
  tmp_c1 := tail(tmp_c1); tmp_c2 := tail(tmp_c2); ...
  tmp_e' := expr(ė, h_c1, h_c2, ...);
  if tmp_e' == 0 then
    t_c1 := append(t_c1, h_c1); ...
```

Figure 3.4: delete simulation

fragment (Figure 3.4) simulates deletion. Similar to update, the code iterates over the encoding, rebuilding the storage and skipping rows where the expression evaluates to a non-zero value.

It is straightforward to show that the simulations are correct on states that are encodings derived with $\mathrm{encode\text{-}column}$. The mapping S of SQL statements $\dot{c}$ to the corresponding programs above is that a correct simulation and satisfies Requirement 5.

Noninterference

The second step of our approach is split into two parts: noninterference and typability. Noninterference of the simulation is derived from the conditions of the eval.

In our case, both α and α^τ are simply identity functions. Integers of the while language are translated to equivalent integers of SQL: $\alpha(n) = n$. Similarly, we assume the same lattice for host and embedded language, so we translate a level from the host to the same level in the embedding: $\alpha^\tau(\ell) = \ell$.

γ takes a list of tuples and projects it such that the result is the first tuple, if such exists, or zeros otherwise.

$$\gamma(\nu) = \begin{cases} (0, \ldots, 0) & \nu = \emptyset \\ \nu(1) & \text{else} \end{cases}$$

γ^τ takes the element and length labels of SQL and combines them into their upper bound to account for the type of the data values, as well as for the result being empty or not: $\gamma^\tau(\ell_\bullet^{\ell_t}/\ell_s)(i) = \ell_i \sqcup \ell_t$. The join $\sqcup$ is monotonous, so γ^τ is. Indistinguishability of two lists of tuples implies that they are either both empty or agree on their first element (or observable components thereof). Thus, indistinguishable results sets will be projected to indistinguishable tuples at the host level.

With these definitions, noninterference applies as established by Lemma 2.

Noninterference is a semantical property defined over inputs and outputs. Since the simulation is functionally equivalent, that is, produces the same outputs for the same inputs, the simulation is also noninterferent.

Typing

Now that we have shown that there exists a simulation of the embedded program, we need to show that the simulation is typable with respect to the types of the eval.

We start by giving an environment mapping T. Given a typing environment Δ of the embedded language, giving security levels ℓ_c to columns c, we define $T(\Delta)(\mathrm{t\_c}) = \Delta(\mathrm{c})$. We will show typability of S with respect to this mapping.

Part one of the Type-Correct Simulation Requirement requires that typed embedded states are mapped to typed host-level encodings. This is trivially given. States are typable if they structurally conform to the typing environment: all and only those columns mapped in the environment exist in the state. As columns are encoded into variables, and those variables are mapped in $T(\Delta)$, it follows that when the embedded state is typed with respect to Δ, then the encoded state is typed with respect to $T(\Delta)$.

Part two requires that whenever an eval statement is typed with respect to environments Γ and Δ, then the simulation is typed with respect to Γ and $T(\Delta)$. For our simulation, this is given as our helper macros head, tail and append are generic over one level, and inputs and outputs are at that level in all simulation fragments. Furthermore, expr computes the exact same integer expression of $\dot{e}$, and column/variable levels are identical. As integer expressions have the same typing judgments in our while and SQL languages, expr can be typed, too. Thus, the code fragments are all typable with respect to $T(\Delta)$.

Next, we need to show that the encoding is fine-grained enough, that is, indistinguishability is preserved. For clarity, we will assume a simple table instead of a set of tables. We have $\mu_1 \sim_{\Gamma,\ell} \mu_2 \iff \forall x.\Gamma(x) \sqsubseteq \ell \implies \mu_1(x) = \mu_2(x)$ and $\nu_1 \sim_{\Delta,\ell} \nu_2 \iff \Delta(table) \sqsubseteq \ell \implies \downarrow_\ell^\Delta(\nu_1) = \downarrow_\ell^\Delta(\nu_2)$, where $\Gamma = T(\Delta)$. Let us consider $\Delta(table) \sqsubseteq \ell$, as the other case is trivial. Here $\downarrow_\ell^\Delta(\nu)$ erases columns where $\Delta(c) \not\sqsubseteq \ell$. So if $\nu_1 \sim_{\Delta,\ell} \nu_2$, then for all c such that $\Delta(c) \sqsubseteq \ell$ and all rows i $\nu_1(i).c = \nu_2(i).c$. As the equality rows for all rows, it follows that the encodings of the columns are also equivalent, since encodings must be bijections. Thus, $\mu_1(v_c) = \mu_2(v_c)$ for all column

variables v_c such that $\Delta(c) \sqsubseteq \ell$. Thus, for all v such that $\Gamma(v) \sqsubseteq \ell$ we have $\mu_1(v) = \mu_2(v)$, and thus $\mu_1 \sim_{\Gamma,\ell} \mu_2$. This derivation guarantees that the encoding is both indistinguishability-preserving and typable.

It thus follows that the simulation is typable with respect to T, satisfying requirement 11.

Replacing eval

The last step of our approach ties the previous subsections together and shows how to replace the eval for the simulation. We first need a technical lemma about typing judgments of extended typing environments.

Lemma 17 (Weakening)**.** *Given typing environments* Γ *and* Γ' *such that* Γ' *is an extension of* Γ*, then for all statements* c *and levels* ℓ*, if* $\Gamma \vdash c : \ell$ ok*, then* $\Gamma' \vdash c : \ell$ ok*.*

Proof. This is a standard lemma. First, we show by induction of on the derivation an equivalent statement for expressions: if $\Gamma \vdash e : \ell$, then $\Gamma' \vdash e : \ell$.

Case T-Lit is immediate.

Case T-Var. As Γ' is an extension of Γ, it agrees with Γ on x. Thus, an we can appeal to T-Var which yields $\Gamma' \vdash x : \Gamma(x) = \Gamma'(x) = \ell$.

Case T-Exp. We have $\Gamma \vdash e_1 : \ell$ and $\Gamma \vdash e_2 : \ell$. By inductive hypothesis, $\Gamma' \vdash e_1 : \ell$ and $\Gamma' \vdash e_2 : \ell$. An application of T-Exp yields the result.

Case T-$ESub$ is analogous to the previous case.

The main proof proceeds by induction on the derivation of $\Gamma \vdash c : \ell$ ok.

Case T-Ass. Then $\Gamma \vdash x : \ell$, $\Gamma \vdash e : \ell'$ and $\ell' \sqsubseteq \ell$. By the previous statement about expressions, $\Gamma' \vdash x : \ell$, $\Gamma' \vdash e : \ell'$. An application of T-Ass yields the result.

Case T-Seq. Then $\Gamma \vdash c : \ell$ ok and $\Gamma \vdash c' : \ell$ ok. By inductive hypothesis, $\Gamma' \vdash c : \ell$ ok and $\Gamma' \vdash c' : \ell$ ok. Thus by T-Seq $\Gamma' \vdash c; c' : \ell$ ok.

Cases $T\text{-}If$, $T\text{-}While$ and $T\text{-}SSub$ are analogous to the previous case.

□

Lemma 18 (Substitution Typable)**.** *Given an eval statement, let c_{eval} be its typable host simulation. Assume a context $E[\bullet]$, that is, a composed program with a statement hole. Furthermore assume an $x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$, Γ, Δ and ℓ such that $\Gamma, \Delta \vdash E[x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}] : \ell\,\mathrm{ok}$. Then there exists an extension Γ' of Γ such that $\Gamma', \Delta \vdash E[c_{eval}] : \ell\,\mathrm{ok}$.*

Proof. First note that this is not a traditional substitution lemma, since the substituted element is not just a variable. Furthermore, the typing environment is actually growing since we need to type the temporaries of the simulation. The proof is an induction over the derivation of the typing of $E[x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}]$.

$E = \bullet$: Then the statement is correct by the proofs in the previous subsections.

$E = x := e$ or $E = x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$ Vacuous, no nested (composed-level) statement.

$E = c; E'$ or $E = E'; c$: In both cases typing ends with the sequencing rule, which yields that both c and $E'[x := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}]$ are typed under Γ and Δ. By inductive hypothesis, $E'[c_{eval}]$ is typed under a Γ' and Δ, where Γ' is an extension of Γ. By Lemma 17, c can be typed under Γ'. By the sequencing rule, we gain a complete typing of the sequence again. The cases for conditionals $\mathrm{if}\, e \,\mathrm{then}\, c \,\mathrm{else}\, E'$/$\mathrm{if}\, e \,\mathrm{then}\, E' \,\mathrm{else}\, c$ and loop $\mathrm{while}\, e \,\mathrm{do}\, E'$ are analogous.

□

We can use this lemma to iteratively replace all eval statements in the original composed program with their respective simulations. The result is a typing of a pure-host statement under the composed-language rules. The next lemma states that such a typing induces the corresponding host-level typing of the statement.

Lemma 19 (Eval-free Composed To Host)**.** *If a statement c that does not contain* eval *is typed under Γ and Δ as $\Gamma, \Delta \vdash c : \ell\,\mathrm{ok}$, then c can be typed as $\Gamma \vdash c : \ell\,\mathrm{ok}$.*

Proof. By induction on the derivation of $\Gamma, \Delta \vdash c : \ell\,\mathrm{ok}$. Since c does not contain eval, no substatement contains an eval, either. Thus the inductive hypothesis applies. Furthermore, the eval typing rule cannot appear in the derivation, because it only appears to eval. We can thus reconstruct a host-level typing corresponding to the (copied) composed-level typing rule. □

Corollary 20 (Simulation Pure-Host Typable)**.** *If c is a composite program that is typable as $\Gamma, \Delta \vdash c : \ell\,\mathrm{ok}$, then there exists an extension Γ' of Γ that encodes Δ such that the simulation program c_{eval}, where all* eval *statements have been substituted for their simulation, is typable as $\Gamma' \vdash c_{eval} : \ell\,\mathrm{ok}$.*

This formally proves that the simulation program is noninterferent by noninterference of typed host-language programs. Now, the final step needs to formally show that the simulation program is equivalent to the original program. This is obviously modulo the behavior of temporaries of the simulation, which are exposed to the host.

Theorem 21 (Simulation Equivalence up to Γ)**.** *Given an eval statement, let c_{eval} be its typable host simulation. If $\Gamma, \Delta \vdash c : \ell\,\mathrm{ok}$, then there exists an extension Γ' such that $\Gamma' \vdash c_{eval} : \ell\,\mathrm{ok}$, and for all $\mu_1, \mu_1', \nu, \nu', \mu_2$ where $\Gamma \vdash \mu_1\,\mathrm{ok}$, $\Gamma' \vdash \mu_2\,\mathrm{ok}$ and μ_2 is an extension of μ_1 such that the extension encodes ν, and $\mu_1, \nu, c \Rightarrow \mu_1', \nu'$, then there exists μ_2' an extension of μ_1' such that $\mu_2, c_{eval} \Rightarrow \mu_2'$ and the extension encodes ν'.*

Proof. We find Γ' by Lemma 18. Now induction on the derivation of $\mu_1, \nu, c \Rightarrow \mu_1', \nu'$. We show select cases.

- $c \equiv x := e$. This base case results in $c_{eval} = c$, as c does not contain an eval statement. Then we can set $\mu_2' = \mu_2[x := \mu_2(e)]$, which is an extension of $\mu_1' = \mu_1[x := \mu_1(e)]$, because e is restricted to variables in μ_1 and μ_2 agrees with μ_1 on those. The extension also encodes ν' correctly, because $\nu = \nu'$ and x is the only changed mapping.

- $c \equiv \mathrm{while}\, e\, \mathrm{do}\, c'$. We treat the case that $\mu_1(e) \neq 0$ ($S\text{-}WhileT$), the other is analogous and simpler. First, $c_{eval} = \mathrm{while}\, e\, \mathrm{do}\, c'_{eval}$. e can only refer to variables in the domain of μ_1

because of c being well-typed. μ_2 agrees with μ_1 on all variables in μ_1. Thus $\mu_2(e) = \mu_1(e) \neq 0$. Now, by inductive hypothesis, because $\mu_1, \nu, c' \Rightarrow \mu_1'', \nu''$ and c is typed, there exists μ_2'' such that $\mu_2, c'_{eval} \Rightarrow \mu_2''$, where μ_2'' is an extension of μ_1'' that encodes ν''. Also, by inductive hypothesis, because $\mu_1'', \nu'', \mathrm{while}\, e \,\mathrm{do}\, c' \Rightarrow \mu_1', \nu'$, there exists μ_2' such that $\mu_2'', \mathrm{while}\, e \,\mathrm{do}\, c'_{eval} \Rightarrow \mu_2'$, where μ_2' is an extension of μ_1' that encodes ν'. The cases for if and sequence are similar.

$c \equiv x := \mathrm{eval}\, e \,\mathrm{in}\, \dot{c}$. Then $c_{eval} = c'; x := x'$, where c' is the simulation of eval and the assignment writes back the value from temporary x'. By the properties of the simulation we have that μ_2'', the state after evaluating c', is identical to μ_1 on the domain of μ_1, because the pure simulation is constructed to not affect original variables. The extension part however encodes ν' by property of being a simulation. Finally, $\mu_2' = \mu_2''[x := \mu_2''[x']]$, and we have $\mu_2''[x'] = \mu_1'[x]$ by virtue of the simulation. Thus, the extension part of μ_2' over μ_1' encodes ν', and μ_2' agrees with μ_1' on the domain of μ_1'.

□

Corollary 22 (Replacement)**.** *For a typable composed program c, the program c_{eval} created by replacing all* `eval` *statements with a corresponding simulation, is typable and functionally equivalent to c.*

This proves the requirements 13 and 14. This concludes all steps of our framework. Formally, we have proven all the requirements of Theorem 16. This means that the theorem applies, and guarantees the soundness of the security type system of the composed language. Any typable program composed of While and SQL code is guaranteed to be noninterfering.

Chapter 4: SECURITY COMPLETENESS

The content of this chapter is based on [41].

4.1 Introduction

The previous chapter introduced our framework approach to proving security-typed composed languages secure under certain assumptions to the composition. The framework, informally, replaces all embedded programs with simulations and then argues over a pure host-language program. One of the requirements is that the simulation of an embedded program fragment is typable with respect to the embedded program's original typing.

In this chapter, we introduce and study *Security Completeness*. If a security-typed language is security-complete, *all* noninterfering computations have a typable representation. Thus a security-complete language satisfies the typability requirement trivially, as simulations are proven noninterfering by Lemma 2 and simulation correctness.

Suppose one has a noninterferent function $f(x, y)$ that produces a public output from public input x and secret input y. To obtain a typeable version f' of f, one can first define a function g to be the same function as f, but type both of g's inputs as public, and its output also as public. Intuitively, assuming that the underlying language is Turing complete, and that any function that involves only a single security level is typeable, g can be expressed and typed. Now, one can define $f'(x, y) = g(x, c)$, for some suitable constant c, which is well-typed since, usually, constants are public. A typical noninterference theorem guarantees that $f(x, y_1) = f(x, y_2)$, for all y_1, y_2, which ensures that $f = f'$.

This basic argument may seem trivial; actually showing that it holds for classes of languages providing operations over certain classes of data types (rather than a specific language with integer data) brings up several issues. Our basic approach for establishing this property is first described in Section 4.2 and formalized in Section 4.3. We then, in Sections 4.4 and 4.5, show how more complex data types and references can be supported, which adds additional requirements on the

host language that would generally be expected to be satisfied by languages that support such entities. Note that the current work does not support extending the types of inputs and outputs to functions. Finally, in Section 4.6, we show that the requirements placed on host languages are reasonable by showing how three languages, the system of [89], FlowML [72], and the work in [14] satisfy the requirements.

In this chapter we show how arbitrary noninterferent functions can be computed in certain classes of security-typed languages, and show that these classes of languages are reasonable.

4.2 Approach

This section details our approach in the simplified case of output indistinguishability being output equivalence. This is, for example, the case if the output is just a single integer value that is assumed to be public, which is a common form to formalize noninterference (e.g., [72]). We will formalize this setting in the next section, and the following sections will detail generalizations to more complicated values. A subsection treats the differences between termination-sensitive and termination-insensitive noninterference.

4.2.1 Basic Approach

If a function f is computable, then there exists a program p that computes f, that is, the output of p agrees with f under the same inputs for the right meaning of inputs and output. Noninterference is a dependency problem, If a program is noninterferent, then the (low) result does not depend on high inputs. This means that, for any high inputs, the low output value will be the same. We are thus able to substitute arbitrary constants for those inputs when computing only low outputs. However, we need to prove the existence or wellformed-ness of said program. We approach this from a computability direction, where constant functions and function composition are guaranteed by primitive recursion.

We prefer a composition requirement over more direct manipulations because it abstracts the exact syntax and semantics of the language involved. Note that we do not need to inspect programs

at all, as required by, for example, a slicing approach. Instead, we show the existence of some separate program that is typable and computes an equivalent function. This allows us a generalization that can accept, for example, both imperative and functional languages. Note that it is important to find an *equivalent* function: for our simulation argument, it is not enough to compute correctly up to indistinguishability.

The final step is showing typability according to the noninterference signature of this intuitive construction. We observe that, trivially, every function is noninterferent if all inputs are considered public - noninterference resolves to determinism (or some similar notion in the case of nondeterministic languages). This generalizes to any single security level. It seems reasonable to require that a non-trivial security-type system should be able to type a program with a security typing assigning a single level to everything. This is our first requirement for a security-typed language.

Next, projection and constant functions should be typable at the respective levels. Under projection we understand here functions of multiple inputs that return one of those inputs. For example, projection $\pi_1(x, y) = x$ should be typable as $\ell_x \times \ell_y \to \ell_x$ for any ℓ_x and ℓ_y. The constant function is noninterferent no matter the security typing, since the output is always the same and does not depend on the inputs. Our requirement is that it can be typed with any result level, including public.

Last, we require that composition is typable if the components are typable and agree on input and output types. By construction, the input types of the composition agree with the security typing for the original program, and the output type with the output type of said program. Then a typing states the same (or extended) noninterference property that we intended for the original program, and the construction guarantees functional equivalence.

A demonstration is shown in Figure 4.1. The original is shown in part 4.1a: it is the program $p = \mathrm{a} := (\mathrm{x} + \mathrm{y}) - (\mathrm{y} - \mathrm{x})$ computing $f(x, y) = 2x$. The result of the transformation is shown in part 4.1b. The first block projects x, the second block is the constant 0 function, the third block represents the original program in a low-typable version, and the fourth block is the final result. The overhead comes from the necessity of renaming variables when composing in imperative languages

Program	*Typing*
$\mathrm{a} := (\mathrm{x} + \mathrm{y}) - (\mathrm{y} - \mathrm{x});$	$[\mathrm{x} : L,\ \mathrm{y} : H,\ \mathrm{a} : L]$

(a) Original Program

Code	*Typing*	*High-level equivalent*
$\mathrm{t}_1^{\mathrm{x}} := \mathrm{x};\ \mathrm{t}_1^{\mathrm{y}} := \mathrm{y};\ \mathrm{t}_1^{\mathrm{o}} := \mathrm{t}_1^{\mathrm{x}};$	$[\mathrm{t}_1^{\mathrm{x}} : L,\ \mathrm{t}_1^{\mathrm{y}} : H,\ \mathrm{t}_1^{\mathrm{o}} : L]$	$\mathrm{t}_1^{\mathrm{o}} = \pi_1(x, y)$
$\mathrm{t}_2^{\mathrm{x}} := \mathrm{x};\ \mathrm{t}_2^{\mathrm{y}} := \mathrm{y};\ \mathrm{t}_2^{\mathrm{o}} := 0;$	$[\mathrm{t}_2^{\mathrm{x}} : L,\ \mathrm{t}_2^{\mathrm{y}} : H,\ \mathrm{t}_2^{\mathrm{o}} : L]$	$\mathrm{t}_2^{\mathrm{o}} = c_0(x, y)$
$\mathrm{t}_3^{\mathrm{x}} := \mathrm{t}_1^{\mathrm{o}};\ \mathrm{t}_3^{\mathrm{y}} := \mathrm{t}_2^{\mathrm{o}};$ $\mathrm{t}_3^{\mathrm{o}} := (\mathrm{t}_3^{\mathrm{x}} + \mathrm{t}_3^{\mathrm{y}}) - (\mathrm{t}_3^{\mathrm{y}} - \mathrm{t}_3^{\mathrm{x}});$	$[\mathrm{t}_3^{\mathrm{x}} : L,\ \mathrm{t}_3^{\mathrm{y}} : L,\ \mathrm{t}_3^{\mathrm{o}} : L]$	$\mathrm{t}_3^{\mathrm{o}} = p^L(\mathrm{t}_1^{\mathrm{o}}, \mathrm{t}_2^{\mathrm{o}})$ $= p^L(x, 0)$
$\mathrm{a} := \mathrm{t}_3^{\mathrm{o}};$		

(b) Transformed Program

Figure 4.1: Simple Program with Security Context

to prevent side effects. The original program is not typable under the signature that assumes x low and y high, because the assignment cannot be typed. The new construction, however can be typed and computes essentially the same function.

A more complicated example is shown in Figure 4.2. Here we have a program with two outputs, one public and one confidential. In that case essentially we run the program twice, once for each output level. The first run computes for the public output. As before, by noninterference the *public part* of the output will be computed correctly. The confidential output, however, is not guaranteed to be correct. This is the reason we run the original program another time. This time we treat all inputs as confidential. Then we can run a single-level version with level H. *All* outputs of this run will be confidential. We ignore the formerly public output and only extract the originally confidential output. Combining the public result from the first run and the confidential result from the second run yields the final result.

4.2.2 Termination Sensitivity

The development in the previous subsection only holds if we consider termination-sensitive non-interference. In that case, two runs on indistinguishable inputs have to agree on their termination

Program	*Typing*
if $\mathrm{y} > 0$ then	$[\mathrm{x} : L,\ \mathrm{y} : H,\ \mathrm{a} : L\ \mathrm{b} : H]$
$\mathrm{a} := 0; \mathrm{b} = 1;$	
else	
$\mathrm{a} := 0; \mathrm{b} = 2;$	

(a) Original Program

Code	*Typing*	*High-level equiv.*
$\mathrm{t}_1^{\mathrm{x}} := \mathrm{x}; \mathrm{t}_1^{\mathrm{y}} := \mathrm{y}; \mathrm{t}_1^{\mathrm{o}} := \mathrm{t}_1^{\mathrm{x}};$	$[\mathrm{t}_1^{\mathrm{x}} : L,\ \mathrm{t}_1^{\mathrm{y}} : H,\ \mathrm{t}_1^{\mathrm{o}} : L]$	$\mathrm{t}_1^{\mathrm{o}} = \pi_1(x, y)$
$\mathrm{t}_2^{\mathrm{x}} := \mathrm{x}; \mathrm{t}_2^{\mathrm{y}} := \mathrm{y}; \mathrm{t}_2^{\mathrm{o}} := 0;$	$[\mathrm{t}_2^{\mathrm{x}} : L,\ \mathrm{t}_2^{\mathrm{y}} : H,\ \mathrm{t}_2^{\mathrm{o}} : L]$	$\mathrm{t}_2^{\mathrm{o}} = c_0(x, y)$
$\mathrm{t}_3^{\mathrm{x}} := \mathrm{t}_1^{\mathrm{o}}; \mathrm{t}_3^{\mathrm{y}} := \mathrm{t}_2^{\mathrm{o}};$	$[\mathrm{t}_3^{\mathrm{x}} : L,\ \mathrm{t}_3^{\mathrm{y}} : L,\ \mathrm{t}_3^{\mathrm{o}} : L,\ \mathrm{t}_3^{\mathrm{p}} : L]$	$\mathrm{t}_3^{\mathrm{o}} = p^L(\mathrm{t}_1^{\mathrm{o}}, \mathrm{t}_2^{\mathrm{o}}).a$
if $\mathrm{t}_3^{\mathrm{y}} > 0$ then		$= p^L(x, 0).a$
$\mathrm{t}_3^{\mathrm{o}} := 0; \mathrm{t}_3^{\mathrm{p}} = 1;$		$\mathrm{t}_3^{\mathrm{p}} = p^L(\mathrm{t}_1^{\mathrm{o}}, \mathrm{t}_2^{\mathrm{o}}).b$
else		$= p^L(x, 0).b$
$\mathrm{t}_3^{\mathrm{o}} := 0; \mathrm{t}_3^{\mathrm{p}} = 2;$		
$\mathrm{t}_4^{\mathrm{x}} := \mathrm{x}; \mathrm{t}_4^{\mathrm{y}} := \mathrm{y}; \mathrm{t}_4^{\mathrm{o}} := \mathrm{t}_4^{\mathrm{y}};$	$[\mathrm{t}_4^{\mathrm{x}} : L,\ \mathrm{t}_4^{\mathrm{y}} : H,\ \mathrm{t}_4^{\mathrm{o}} : H]$	$\mathrm{t}_4^{\mathrm{o}} = \pi_2(x, y)$
$\mathrm{t}_5^{\mathrm{x}} := \mathrm{t}_1^{\mathrm{o}}; \mathrm{t}_5^{\mathrm{y}} := \mathrm{t}_4^{\mathrm{o}};$	$[\mathrm{t}_5^{\mathrm{x}} : H,\ \mathrm{t}_5^{\mathrm{y}} : H,\ \mathrm{t}_5^{\mathrm{o}} : H,\ \mathrm{t}_5^{\mathrm{p}} : H]$	$\mathrm{t}_5^{\mathrm{o}} = p^H(\mathrm{t}_1^{\mathrm{o}}, \mathrm{t}_4^{\mathrm{o}}).a$
if $\mathrm{t}_5^{\mathrm{y}} > 0$ then		$= p^H(x, y).a$
$\mathrm{t}_5^{\mathrm{o}} := 0; \mathrm{t}_5^{\mathrm{p}} = 1;$		$\mathrm{t}_5^{\mathrm{p}} = p^H(\mathrm{t}_1^{\mathrm{o}}, \mathrm{t}_4^{\mathrm{o}}).b$
else		$= p^H(x, y).b$
$\mathrm{t}_5^{\mathrm{o}} := 0; \mathrm{t}_5^{\mathrm{p}} = 2;$		
$\mathrm{a} := \mathrm{t}_3^{\mathrm{o}};$		
$\mathrm{b} := \mathrm{t}_5^{\mathrm{p}};$		

(b) Transformed Program

Figure 4.2: Dual-Output Program with Security Context

behaviour, that is, the first run terminates if and only if the second run terminates. Termination-insensitive noninterference, on the other hand, only makes a statement over two terminating runs. The assumption underlying termination-insensitive noninterference is that an attacker might not be able to observe (non-)termination, or that the one bit of information leaked through termination is acceptable. In that case, the approach outlined in the previous subsection cannot be guaranteed to simulate correctly only up to termination, because termination may depend on the high input, and thus the choice of constants.

The construction outlined above, however, can be extended so that the constructed code correctly mimics termination-insensitive noninterfering programs. Our solution imposes further requirements that allow the application of a standard technique in computability: dovetailing (interleaving computations). If we require the set of values valid for the high inputs to be recursively enumerable, we can *test* the function on all possible inputs. For this test to succeed, we have to be able to simulate the function in a stepwise manner, e.g., as in a small-step semantics. We will interleave the simulations of the different input values, such that if there is at least one value that forces termination, we will find that case. As an example, assume that $f_k(x)$ denotes a computation of $f(x : \mathbb{N})$ for k steps. Then an interleaving could be $f_1(0)$, $f_2(0)$, $f_1(1)$, $f_3(0)$, $f_2(1)$, $f_1(2)$, ... If for any x, $f(x) = v$ is defined, there is a k such that $f_k(x) = v$, and the interleaving contains this computation.

Such an interleaving will terminate if there is at least one terminating high value. To complete correctness with respect to the original, we compute the original function in a high setting in sequence. This will ensure that the simulation does not terminate when it should not.

We demonstrate this approach in Figure 4.3. To interleave computations, our "state" is a pair of natural numbers $\langle \mathrm{t^y}, \mathrm{t^n} \rangle$, where $\mathrm{t^y}$ denotes the input to the program and $\mathrm{t^n}$ denotes the number of steps to execute the program. We will use the Cantor pairing function to encode this pair of natural numbers into a single natural number (in the program denoted by $\mathrm{t^i}$). The function is defined as

$$C(x, y) = \frac{1}{2}(x + y)(x + y + 1) + y$$

and its inverse is

$$C^{-1}(z) = \langle w - z + \frac{w^2 + w}{2}, z - \frac{w^2 + w}{2} \rangle, \qquad \text{where } w = \left\lfloor \frac{\sqrt{8z+1} - 1}{2} \right\rfloor$$

The Cantor pairing function has three properties that are important for our application:

- It is computable, that is, there are while programs computing the numbering.
- It is reversible and the reverse is computable.
- The pairing is bijective, i.e., each pair is mapped to one natural number, and each natural number corresponds to one pair.

These properties allow us to use the single counter t^{i} to exhaustively search the space of all pairs. We use $\underline{\text{cantor}}_1$ to denote a macro of the first component of the reverse mapping, and $\underline{\text{cantor}}_2$ to denote a macro for the second component. An implementation can be derived from the mathematical notation above.

We assume step-wise simulation is given by a macro $\underline{\text{sim}}$ with six parameters: an encoding of the program to run, the two program inputs, the number of steps to simulate, the potential result variable, and a flag variable to denote whether the run was complete after the indicated number of steps.

Note that such a program is guaranteed to exist as the while language is Turing-complete: There is a general Turing machine that can be used to simulate any Turing machine. Accordingly, any Turing-complete language has a general evaluation program that can simulate any (sufficiently encoded) program in the language. This reasoning can be extended to step-wise simulation: The general Turing machine can be extended by another tape with a unary encoding of the number of steps, and the transition function can be extended such that simulation steps only happen as long as the counter tape hasn't run out of steps. Again by Turing-completeness this implies that a corresponding evaluation program for the while language exists, and we denote it by $\underline{\text{sim}}$.

The transformation is shown in Figure 4.3. The first part gives the original program. The

typable simulation is given in the second part. The first three blocks iteratively compute the values of the program for increasing values of y, where we use sim to simulate the given program for t^{n} steps and assign $\mathrm{t}^{\mathrm{s}} = 0$ if the program finished. If the loop terminates, a result for some y will be in $\mathrm{t}_2^{\mathrm{o}}$. Note that the loop only involves L variables and can thus be typed as L. The following three blocks compute an H version of the program for termination correctness: The original program would not terminate for any input $\mathrm{y} > 10$, but the simulation so far will find that the program terminates for input $\mathrm{y} = 0$. So to make the simulation non-terminating in such cases, we run an H version of the original program over the complete original input (but will not use its output). The last block assigns the result of the low simulation as the overall result.

4.3 Formalization

In this section we formalize the approach and requirements outlined in the previous section. We start by introducing generic notation for the security-typed language and its security-type system, and formally defining our requirements in the first subsection. In the second subsection, we formally show how these requirements lead to our revised hypothesis.

Note that in this section we restrict ourselves to functions with a single output. The following sections will generalize to nonrecursive and recursive datatypes, covering functions with multiple outputs by functions with tuple outputs.

4.3.1 Definitions & Requirements

We assume a security-typed language $\mathcal{L}$ and its security-type system $\mathcal{T}$ with associated lattice $\mathcal{S}$. The languages provides a set of values, ranged over by v, a set of programs, ranged over by p, and state or input, ranged over by μ. The language has associated semantics that reduces a program and state to a value and state. We denote the semantics by $(p, \mu) \rightsquigarrow_s (v, \mu')$. We define $\Downarrow$ to include nontermination, such that $(p, \mu) \Downarrow (v, \mu')$ if $(p, \mu) \rightsquigarrow_s (v, \mu')$, and $(p, \mu) \Downarrow (\bot, \bot)$, if there is no such (v, μ').

We connect the semantics to a functional interpretation through two predicates defined by the

Program	*Typing*
$\mathrm{a} := \mathrm{x};$	$[\mathrm{x} : L,\ \mathrm{y} : H,\ \mathrm{a} : L]$
while $\mathrm{y} \neq 10$ **do**	
$\quad \mathrm{a} := \mathrm{x};\ \mathrm{y} := \mathrm{y} + 1;$	

(a) Original Program

Code	*Typing*	*High-level equiv.*
$\mathrm{t}_1^{\mathrm{x}} := \mathrm{x};\ \mathrm{t}_1^{\mathrm{y}} := \mathrm{y};\ \mathrm{t}_1^{\mathrm{o}} := \mathrm{t}_1^{\mathrm{x}}$	$[\mathrm{t}_1^{\mathrm{x}} : L,\ \mathrm{t}_1^{\mathrm{y}} : H,\ \mathrm{t}_1^{\mathrm{o}} : L]$	$\mathrm{t}_1^{\mathrm{o}} = \pi_1(x, y)$
$\mathrm{t}^{\mathrm{i}} := 0;\ \mathrm{t}^{\mathrm{s}} := 1;$	$[\mathrm{t}^{\mathrm{i}} : L,\ \mathrm{t}^{\mathrm{s}} : L]$	
while $\mathrm{t}^{\mathrm{s}} > 0$ **do**		
$\quad \underline{\mathrm{cantor}}_1(\mathrm{t}^{\mathrm{i}}, \mathrm{t}_2^{\mathrm{y}});$	$[\mathrm{t}_2^{\mathrm{y}} : L,\ \mathrm{t}^{\mathrm{n}} : L]$	$C^{-1}(\mathrm{t}^{\mathrm{i}}) = \langle \mathrm{t}_2^{\mathrm{y}}, \mathrm{t}^{\mathrm{n}} \rangle$
$\quad \underline{\mathrm{cantor}}_2(\mathrm{t}^{\mathrm{i}}, \mathrm{t}^{\mathrm{n}});$		
$\quad \mathrm{t}_2^{\mathrm{x}} := \mathrm{t}_1^{\mathrm{o}};$	$[\mathrm{t}_2^{\mathrm{x}} : L,\ \mathrm{t}_2^{\mathrm{o}} : L]$	$p^L(\mathrm{t}_2^{\mathrm{x}}, \mathrm{t}_2^{\mathrm{y}}) \rightarrow^{(\mathrm{t}^{\mathrm{n}})} \mathrm{t}_2^{\mathrm{o}}\ ?$
$\quad \underline{\mathrm{sim}}(p^L, \mathrm{t}_2^{\mathrm{x}}, \mathrm{t}_2^{\mathrm{y}}, \mathrm{t}^{\mathrm{n}}, \mathrm{t}_2^{\mathrm{o}}, \mathrm{t}^{\mathrm{s}});$		
$\quad \mathrm{t}^{\mathrm{i}} := \mathrm{t}^{\mathrm{i}} + 1;$		
$\mathrm{t}_3^{\mathrm{x}} := \mathrm{x};\ \mathrm{t}_3^{\mathrm{y}} := \mathrm{y};\ \mathrm{t}_3^{\mathrm{o}} := \mathrm{t}_3^{\mathrm{x}};$	$[\mathrm{t}_3^{\mathrm{x}} : L,\ \mathrm{t}_3^{\mathrm{y}} : H,\ \mathrm{t}_3^{\mathrm{o}} : H]$	$\mathrm{t}_3^{\mathrm{o}} = \pi_1(x, y)$
$\mathrm{t}_4^{\mathrm{x}} := \mathrm{x};\ \mathrm{t}_4^{\mathrm{y}} := \mathrm{y};\ \mathrm{t}_4^{\mathrm{o}} := \mathrm{t}_4^{\mathrm{y}};$	$[\mathrm{t}_4^{\mathrm{x}} : L,\ \mathrm{t}_4^{\mathrm{y}} : H,\ \mathrm{t}_4^{\mathrm{o}} : H]$	$\mathrm{t}_4^{\mathrm{o}} = \pi_2(x, y)$
$\mathrm{t}_5^{\mathrm{x}} := \mathrm{t}_3^{\mathrm{o}};\ \mathrm{t}_5^{\mathrm{y}} := \mathrm{t}_4^{\mathrm{o}};$	$[\mathrm{t}_5^{\mathrm{x}} : H,\ \mathrm{t}_5^{\mathrm{y}} : H,\ \mathrm{t}_5^{\mathrm{o}} : H]$	$\mathrm{t}_5^{\mathrm{o}} = p^H(x, y)$
$\mathrm{t}_5^{\mathrm{o}} := \mathrm{t}_5^{\mathrm{x}};$		
while $\mathrm{t}_5^{\mathrm{y}} \neq 10$ **do**		
$\quad \mathrm{t}_5^{\mathrm{o}} := \mathrm{t}_5^{\mathrm{x}};\ \mathrm{t}_5^{\mathrm{y}} := \mathrm{t}_5^{\mathrm{y}} + 1;$		
$\mathrm{a} := \mathrm{t}_2^{\mathrm{o}};$		

(b) Transformed Program

Figure 4.3: Example Termination-Insensitive Program

$$
\begin{aligned}
&ni(f, \tau_1 \times \cdots \times \tau_n \to \tau_r) \iff \\
&\left(\begin{array}{l}
\forall l \in \mathcal{S}.top(\tau_r) \sqsubseteq l \implies \\
\quad \forall x_1^1, x_1^2 : \tau_1, x_2^1, x_2^2 : \tau_2, \ldots . \\
\quad\quad (\forall i.top(\tau_i) \sqsubseteq l \implies x_i^1 = x_i^2) \implies \\
\quad\quad\quad f(x_1^1, \ldots) = f(x_1^2, \ldots)
\end{array}\right)
\end{aligned}
$$

Figure 4.4: Noninterference

language. A program p computes function $f(x_1, \ldots, x_n)$, if for all μ such that $in_{p,f}(x_1, \ldots, x_n, \mu)$, and $(p, \mu) \Downarrow (v, \mu')$, we have $f(x_1, \ldots, x_n) = res_{p,f}(v, \mu')$, where in and res abstract how a language defines input and output in program p with respect to function f[1]. We denote this by $comp(p, f)$.

The type system provides a set of types, ranged over by τ, and type judgements of the form $\Gamma \vdash_s p : \tau$, $\Gamma \vdash_s v : \tau$[2]and $\Gamma \vdash_s \mu$. Note that for our purposes, it is not necessary to explicitly include a program counter in the notation. If a specific language needs a pc, it can be treated as part of Γ or τ. Security completeness is about whole programs, so low side effects are permissible as long as the resulting function is noninterferent[3]. We can extract a type level $l \in \mathcal{S}$ from a type τ through the function top. We use two predicates to connect a type judgment and function signature, similar to the semantic connection. A judgment $\Gamma \vdash_s p : \tau$ is typed according to f with signature $S = \tau_1 \times \cdots \times \tau_n \to \tau_r$, if $in^t_{p,f}(\Gamma, \tau, \tau_1, \ldots, \tau_n)$ and $\tau_r = res^t_{p,f}(\Gamma, \tau)$. We denote this as $typed(p, f, S, \Gamma, \tau)$.

Our definition of noninterference ni for a function f with respect to (security) signature $\tau_1 \times \cdots \times \tau_n \to \tau$ can be found in Figure 4.4. A program p is noninterferent with respect to f and a signature $\tau_1 \times \cdots \times \tau_n \to \tau_r$, if p computes f and f is noninterferent. The type system guarantees that a typable program is noninterferent with respect to all functions it computes and is typed

[1]For an example, recall how in the example of Figure 4.1 the inputs were bound to variables `x` and `y` and the output to variable `a`.

[2]Note that values v are not required to be programs, but need a type judgment.

[3]This is sufficient since security completeness is about the function represented by the beginning and ending states of a program; it does not need to explicitly consider low side-effects that do not change the final result. The pc, thus, does not need to be considered explicitly in the general results on noninterference.

accordingly[4]:

$$\Gamma \vdash_s p : \tau \wedge comp(p, f) \wedge typed(p, f, S, \Gamma, \tau) \implies ni(f, S)$$

Associated with the security-typed versions we expect ground-typed versions, denoted by a g subscript or by $\lfloor \bullet \rfloor$ (which can be seen as an erasure function removing all security annotations), that is, the security-typed language is based on a standard language and type system with regular soundness guarantees, that is, ground-typed programs do not go wrong.

We require the following manipulation functions for annotations.

Requirement 23 (Erasure & Lift)**.** *There exist an erasure function $\lfloor \bullet \rfloor$ and a lift function $\lceil \bullet \rceil^l$ such that*

$$\forall p, v, \mu \in \mathcal{L}, \Gamma, \tau \in \mathcal{T}.\lfloor p \rfloor \in \lfloor \mathcal{L} \rfloor, \lfloor v \rfloor \in \lfloor \mathcal{L} \rfloor, \lfloor \mu \rfloor \in \lfloor \mathcal{L} \rfloor, \vdash \Gamma \Rightarrow \vdash \lfloor \Gamma \rfloor, \lfloor \tau \rfloor \in \lfloor \mathcal{T} \rfloor$$

$$\forall p_g, v_g, \mu_g \in \lfloor \mathcal{L} \rfloor, \Gamma_g, \tau_g \in \lfloor \mathcal{T} \rfloor, l \in \mathcal{S}.\lceil p_g \rceil^l \in \mathcal{L}, \lceil v_g \rceil^l \in \mathcal{L}, \lceil \mu_g \rceil^l \in \mathcal{L}, \vdash \Gamma_g \Rightarrow \vdash \lceil \Gamma_g \rceil^l, \lceil \tau \rceil^l \in \mathcal{L}$$

$$\forall \tau_g, l \in \mathcal{S}.top(\lceil \tau_g \rceil^l) = l$$

We define a complete relabeling $\lceil \bullet \rceil^l = \lceil \lfloor \bullet \rfloor \rceil^l$. The identity $\tau = \lceil \tau \rceil^{top(\tau)}$ is required to hold.

We use the requirements on relabeling to form a partial order on types lifted from their security levels.

$$\tau_1 \sqsubseteq \tau_2 \iff \exists \tau \in \mathcal{T}, l_1, l_2 \in \mathcal{S}.l_1 \sqsubseteq l_2 \wedge \tau_1 = \lceil \tau \rceil^{l_1} \wedge \tau_2 = \lceil \tau \rceil^{l_2}$$

Security and ground languages are suitably related:

[4]The notion that a program might compute multiple functions might be surprising. But computation here is defined with respect to what parts of the output state are of interest. For example, consider projection $\pi_1(x, y) = x$ in a WHILE language.

Requirement 24 (Security to Ground.)**.**

$$
\begin{aligned}
&\forall p, v, \mu, \mu', \Gamma, \tau. \\
&\quad \Gamma \vdash_s p : \tau \implies \lfloor \Gamma \rfloor \vdash_g \lfloor p \rfloor : \lfloor \tau \rfloor \\
&\quad \Gamma \vdash_s v : \tau \implies \lfloor \Gamma \rfloor \vdash_g \lfloor v \rfloor : \lfloor \tau \rfloor \\
&\quad \Gamma \vdash_s \mu \implies \lfloor \Gamma \rfloor \vdash_g \lfloor \mu \rfloor \\
&\quad (p, \mu) \rightsquigarrow_s (v, \mu') \implies (\lfloor p \rfloor, \lfloor \mu \rfloor) \rightsquigarrow_g (\lfloor v \rfloor, \lfloor \mu' \rfloor)
\end{aligned}
$$

Furthermore, it holds that

$$
\begin{aligned}
&\mathit{typed}(p, f : \tau_1 \times \cdots \times \tau_n \to \tau, \Gamma, \tau) \implies \\
&\quad \mathit{typed}(\lfloor p \rfloor, f : \lfloor \tau_1 \rfloor \times \cdots \times \lfloor \tau_n \rfloor \to \lfloor \tau \rfloor, \lfloor \Gamma \rfloor, \lfloor \tau \rfloor)
\end{aligned}
$$

and $\mathit{comp}(p, f) \implies \mathit{comp}(\lfloor p \rfloor, \lfloor f \rfloor)$.

Furthermore, we can gain security-type system judgments from ground-type judgments for any security level from $\mathcal{S}$.

Requirement 25 (Single-Level.)**.**

$$
\begin{aligned}
&\forall l \in \mathcal{S}. \Gamma_g \vdash_g p_g : \tau_g \implies \lceil \Gamma_g \rceil^l \vdash_s \lceil p_g \rceil^l : \lceil \tau_g \rceil^l \\
&\forall l \in \mathcal{S}. \Gamma_g \vdash_g v_g : \tau_g \implies \lceil \Gamma_g \rceil^l \vdash_s \lceil v_g \rceil^l : \lceil \tau_g \rceil^l \\
&\forall l \in \mathcal{S}. \Gamma_g \vdash_g \mu_g \implies \lceil \Gamma_g \rceil^l \vdash_s \lceil \mu_g \rceil^l \\
&\forall l \in \mathcal{S}. (p_g, \mu_g) \rightsquigarrow_g (v_g, \mu'_g) \implies (\lceil p_g \rceil^l, \lceil \mu_g \rceil^l) \rightsquigarrow_s (\lceil v_g \rceil^l, \lceil \mu'_g \rceil^l)
\end{aligned}
$$

Furthermore, it holds that

$$
\begin{aligned}
&\mathit{typed}(p_g, f_g, S = \tau_1 \times \cdots \times \tau_n \to \tau, \Gamma_g, \tau_g) \implies \\
&\quad \mathit{typed}(\lceil p_g \rceil^l, f_g, \lceil S \rceil^l = \lceil \tau_1 \rceil^l \times \cdots \times \lceil \tau_n \rceil^l \to \lceil \tau \rceil^l, \lceil \Gamma_g \rceil^l, \lceil \tau_g \rceil^l)
\end{aligned}
$$

and $\mathit{comp}(p_g, f_g) \implies \mathit{comp}(\lceil p_g \rceil^l, f_g)$.

Similar to the base cases of primitive recursive functions. we need programs that compute

projection and constants.

Requirement 26 (Projection)**.** *Let $\pi_i^n(x_1, \ldots, x_n) = x_i$ be the i-th projection function of n inputs. Let $\pi_i^n : \tau_1 \times \cdots \times \tau_n \to \tau_i$ be a signature of π_i^n. Then there exists a program p_i^n, a Γ and τ such that*

$$\Gamma \vdash_s p_i^n : \tau \wedge comp(p_i^n, \pi_i^n) \wedge$$
$$typed(p_i^n, \pi_i^n : \tau_1 \times \cdots \times \tau_n \to \tau_i, \Gamma, \tau)$$

Requirement 27 (Constant Function)**.** *Let $c_i^{x,l}(x_1, \ldots, x_n) = x$ be the x-constant function of n inputs. We have $\tau_1 \times \cdots \times \tau_n \to \tau_r$ a signature of $c_i^{x,l}$, that is $x : \tau_r$ and $\tau_r = [\tau_i]^l$. Then there exists a program p^x, a Γ and τ such that*

$$\Gamma \vdash_s p^x : \tau \wedge comp(p^x, c_x) \wedge$$
$$typed(p^x, c_i^{x,l} : \tau_1 \times \cdots \times \tau_n \to \tau_r, \Gamma, \tau)$$

Note that the existence of constant functions is necessary. Security completeness stipulates the existence of a functionally equivalent *program* for a given noninterfering computation. A program computing a constant does *not* follow from erasure and single-level lifting of constant $\lceil \lfloor x \rfloor \rceil^l$, as this is a value and not necessarily a program. The program requirement is important, since only programs need to be able to be composed. This restriction allows us to easily include imperative languages into the framework.

Finally, we want to compose typed programs. We decided to formulate a general composition requirement, instead of a special-cased one.

Requirement 28 (Composition)**.**

$$
\begin{array}{l}
\forall p_\bullet, p. \\
\left(\begin{array}{l}
\forall 1 \leq i \leq n. \Gamma_i \vdash_s p_i : \tau^i \wedge \mathit{comp}(p_i, f_i) \wedge \\
\quad \mathit{typed}(p_i, f_i : \tau_1 \times \cdots \times \tau_n \to \tau_r^i, \Gamma_i, \tau^i)
\end{array}\right) \wedge \\
\left(\begin{array}{l}
\Gamma \vdash p : \tau^p \wedge \mathit{comp}(p, f) \wedge \\
\quad \mathit{typed}(p, f : \tau_r'^1 \times \cdots \times \tau_r'^m \to \tau_r, \Gamma, \tau^p)
\end{array}\right) \wedge \\
\forall 1 \leq i \leq n. \tau_r^i \sqsubseteq \tau_r'^i \\
\Longrightarrow \\
\exists p_c, \Gamma_c, \tau_c. \begin{array}{l}
\Gamma_c \vdash_s p_c : \tau_c \wedge \mathit{comp}(p_c, f \circ \overrightarrow{f_i}) \wedge \\
\mathit{typed}(p_c, f \circ \overrightarrow{f_i} : \tau_1 \times \cdots \times \tau_n \to \tau_r, \Gamma_c, \tau_c)
\end{array}
\end{array}
$$

where $(f \circ \overrightarrow{f_i})(x_1, \ldots, x_n) = f(f_1(x_1, \ldots, x_n), \ldots, f_m(x_1, \ldots, x_n))$.

4.3.2 Revised Theorem & Proof

Theorem 29 (Security-Typability Completeness)**.** *Assume a language $\mathcal{L}$ and corresponding ground language that fulfill requirements 23, 24, 25, 26, 27, and 28. Such language is security-complete.*

Proof. Assume p a program that computes noninterferent $f : \tau_1 \times \cdots \times \tau_n \to \tau$. Since p is ground-typable, we have $p_g = \lfloor p \rfloor$ such that there is a ground typing $\Gamma_g \vdash_g p_g : \tau_g$, such that $\mathit{typed}(p_g, f : \lfloor \tau_1 \rfloor \times \cdots \times \lfloor \tau_n \rfloor \to \lfloor \tau \rfloor, \Gamma_g, \tau_g)$ by requirement 24. Let $l = \mathit{top}(\tau)$. Define $g_\bullet$ as

$$
g_i = \begin{cases}
\pi_i^n : \tau_1 \ldots \tau_n \to \tau_i & \text{if } \mathit{top}(\tau_i) \sqsubseteq l \\
c_i^{x_i, l} : \tau_1 \ldots \tau_n \to [\tau_i]^l & \text{else, with arbitrary } x_i : [\tau_i]^l
\end{cases}
$$

which exist by requirements 26 and 27. Noninterference of f with respect to signature $\tau_1 \times \cdots \times$

$\tau_n \to \tau$ and level l states that

$$\begin{aligned}\forall x_1^1, x_1^2 : \tau_1, \ldots, x_n^1, x_n^2 : \tau_n.& \\ (\forall i. top(\tau_i) \sqsubseteq l \implies x_i^1 = x_i^2) &\implies \\ f(x_1^1, \ldots) = f(x_1^2, \ldots)&\end{aligned}$$

Take any set of inputs $x_\bullet$ for f. Let $y_\bullet$ be defined as $y_i = g_i(x_1, \ldots, x_n)$. Then $\forall i. top(\tau_i) \sqsubseteq l \implies x_i = y_i$ by construction. Now, by noninterference, we have

$$f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_n(x_1, \ldots, x_n))$$

By requirements 26 and 27, there exist $p_\bullet$, $\Gamma_\bullet$ and $\tau_\bullet^f$ such that

$$\begin{aligned}\forall i. \Gamma_i \vdash_s p_i : \tau_i^g \wedge comp(p_i, g_i) \wedge& \\ typed(p_i, g_i : \tau_1 \ldots \tau_n \to \tau^i, \Gamma_i, \tau_i^g).&\end{aligned}$$

Furthermore, by construction we have $\forall i. \tau^i \sqsubseteq [\tau_i]^l$, by requirements 23 and 27.

By the Single-level requirement, we can lift the typing of p_g such that $\lceil \Gamma_g \rceil^l \vdash_s \lceil p_g \rceil^l : \lceil \tau_g \rceil^l$ and $comp(\lceil p_g \rceil^l, f)$ and $typed(\lceil p_g \rceil^l, f : [\tau_1]^l \times \cdots \times [\tau_n]^l \to [\tau]^l, \lceil \Gamma_g \rceil^l, \lceil \tau_g \rceil^l)$. This allows us to use the Composition requirement, composing $p_\bullet$ into $\lceil p_g \rceil^l$, which is functionally equivalent to composing $g_\bullet$ into f. This results in a program p_c and typing $\Gamma_c \vdash_s p_c : \tau_c$ such that $comp(p_c, f \circ \overrightarrow{g_i})$ and $typed(p_c, f \circ \overrightarrow{g_i} : \tau_1 \times \cdots \times \tau_n \to [\tau]^l, \Gamma_c, \tau_c)$. Previous deductions and identity requirement on relabeling permit us to simplify this to $comp(p_c, f)$ and $typed(p_c, f : \tau_1 \times \cdots \times \tau_n \to \tau, \Gamma_c, \tau_c)$. Thus, the program p_c is typable with the required signature and computes f, which concludes the proof. □

4.3.3 Sufficient vs. Necessary Conditions

Our derivation in the previous subsections concludes that the requirements established are sufficient for a language to be security-complete. The requirements are, however, not generally neces-

sary for security completeness. The leeway that the definition of security completeness allows us, i.e., that *another* equivalent program exists that is typable, makes a reverse deduction impossible in general.

In the special case of a Turing-complete language, we can show that a slight weakening of requirement 25 leads to a set of necessary conditions. The weakening is straightforward:

Requirement 30 (Single-Level, weak)**.**

$$\begin{aligned}
&\forall l \in \mathcal{S}.\Gamma_g \vdash_g p_g/v_g/\mu_g : \tau_g \implies \\
&\exists p'_g.\lceil\Gamma_g\rceil^l \vdash_s \lceil p'_g/v_g/\mu_g\rceil^l : \lceil\tau_g\rceil^l \\
&\forall l \in \mathcal{S}.(p_g, \mu_g) \rightsquigarrow_g (v_g, \mu'_g) \implies \\
&(\lceil p'_g\rceil^l, \lceil\mu_g\rceil^l) \rightsquigarrow_s (\lceil v_g\rceil^l, \lceil\mu'_g\rceil^l)
\end{aligned}$$

Furthermore, it holds that

$$\begin{aligned}
&typed(p_g, f_g : \tau_1 \times \cdots \times \tau_n \to \tau, \Gamma_g, \tau_g) \implies \\
&typed(\lceil p'_g\rceil^l, f_g : \lceil\tau_1\rceil^l \times \cdots \times \lceil\tau_n\rceil^l \to \lceil\tau\rceil^l, \lceil\Gamma_g\rceil^l, \lceil\tau_g\rceil^l)
\end{aligned}$$

and $comp(p_g, f_g) \implies comp(\lceil p'_g\rceil^l, f_g)$.

This states that we can find an alternate program (p'_g) that computes the same value, but is typable. Note how this definition is closer to the intention behind security completeness.

The basic set of requirements is now *necessary*: Projections and constants are guaranteed members of a Turing-complete language, and are, with the right signature, noninterfering. Thus, requirements 27 and 26 must hold in a Turing-complete and security-complete language. Also, composition is a guaranteed operator for Turing-complete languages, and the definition of composition preserves noninterference. Thus, requirement 28 must hold in a Turing-complete and security-complete language. Now finally, take an arbitrary computable function. That function is noninterfering under the assumption of a signature with just one single level ℓ. By Turing-completeness, there exists a program that computes said function. By Security completeness, there

must exist a typable program then that computes said function. As the signature must match, this program is typed with the single level ℓ. This means the Turing-complete and security-complete language satisfies requirement 30 (but not necessarily requirement 25).

4.4 Datatypes

We can extend the formalization of the previous section to data types. For security-typed languages, compound values imply the possibility of more complicated indistinguishability relations, e.g., different parts of a value may have different security levels and need to be treated differently. A statement of noninterference may then use this complex indistinguishability both for inputs and outputs. That is, noninterferent programs create outputs that agree on low parts if the low input parts are equivalent. A simple case for demonstration follows. Assume that the language in questions supports *pairs*. Let $f(x, y) = \langle x, y + 1\rangle$, where x and the first component of the output pair are public, and y and the second component of the output pair are confidential. A sample noninterference statement for this function is

$$\forall x, y_1, y_2, x'_1, x'_2, y'_1, y'_2. \quad f(x, y_1) = \langle x'_1, y'_1\rangle \wedge f(x, y_2) = \langle x'_2, y'_2\rangle \implies x'_1 = x'_2.$$

This is equivalent to an indistinguishability relation for pairs that only forces equivalence on the public component:

$$\langle x_1, y_1\rangle \sim \langle x_2, y_2\rangle \iff x1 = x2$$

Notice that the construction in the previous subsection required the whole output to be equivalent, whereas now only the public part is. Also, the confidential output may depend on the confidential input, as in the given example function.

4.4.1 Assumptions

We assume some structure of complex values. First, complex values can be described as algebraic, that is, are of the form $v = c\ v_1, \ldots, v_n$ for an n-ary constructor c. Note that we require values

to be made up of sub-values. To ensure termination of our recomputation, we require all treated values to be finite. We require typing to structurally match values: if a value $v = c\ v_1, \ldots, v_n$ can be typed as τ under Γ, then there are types $\tau_1, \ldots, \tau_n$ such that $v_1, \ldots, v_n$ are typed under Γ, and for all $v'_1, \ldots, v'_n$ typable in that way, $c\ v'_1, \ldots, v'_n$ can be typed as τ. This is a standard consequence in rule-based type systems.

Matching

Furthermore, we need functions to decompose values to their components. To unify product and variant treatment, we assume a matching construct in the language. Formally, if a type system can type values $v_i = c_i\ v_1^i, \ldots, v_{n_i}^i$ with $c_i \neq c_j$ (for $i \neq j$) as τ, where v_k^i can be typed with τ_k^i, then there exist matching functions with the signature $\mathrm{match} : \tau \to (\tau_1^1 \times \cdots \times \tau_{n_1}^1 \to \tau') \to \cdots \to \tau'$ for all τ', with the semantics that $\mathrm{match}(c_i\ v_1^i, \ldots, v_{n_i}^i, f_1, \ldots, f_m) = f_i(v_1^i, \ldots, v_{n_i}^i)$.

Definition 31 (Exhaustive)**.** *Sequences $c_\bullet$ and $n_\bullet$ are called exhaustive for type τ iff all and only those values that are typable as τ have a constructor in $c_\bullet$, and $n_\bullet$ gives the arity of the corresponding constructors.*

$$\forall v.\ \vdash v : \tau \implies \exists i, v_1, \ldots, v_{n_i}.\ v = c_i\, v_1, \ldots, v_{n_i}$$
$$\forall i \in \mathrm{dom}(c_\bullet).\ \exists v_1, \ldots, v_{n_i}.\ \vdash c_i\, v_1, \ldots, v_{n_i} : \tau$$

Requirement 32 (Matching)**.**

$$\forall \tau.\ \left(\exists c_\bullet, n_\bullet, v_\bullet^\bullet, \tau_\bullet^\bullet . \forall i, j.\ \vdash c_i\ v_1^i, \ldots, v_{n_i}^i : \tau \wedge\ \vdash v_j^i : \tau_j^i\right) \implies$$
$$\forall \tau'. \exists \mathrm{match}.\quad \mathrm{match} : \tau \to (\tau_1^1 \times \cdots \times \tau_{n_1}^1 \to \tau') \to \cdots \to \tau' \wedge$$
$$\mathrm{match}(v, f_1, \ldots, f_n) = \begin{cases} f_1(v_1^1, \ldots, v_{n_1}^1) & \text{if } v = c_1\ v_1^1, \ldots, v_{n_1}^1 \\ \vdots & \\ f_n(v_1^n, \ldots, v_{n_n}^n) & \text{if } v = c_n\ v_1^n, \ldots, v_{n_n}^n \end{cases}$$

where $c_\bullet$ and $n_\bullet$ are exhaustive for τ.

This is standard for pattern-matching languages, and can be simulated in languages without explicit pattern matching (e.g., by branching on tag values encoding variants). We will use the common syntax, that is, "$\lambda x \ldots$" for functions and "$\mathrm{match}\ x\ \mathrm{with} \ldots$" for matching.

Note that this existential requirement is very weak: to compute with datatypes, one form or another of matching is required. In the case of standard encoding of algebraic datatypes into lambda calculus, the value itself is a matching construct. In the case of higher level languages like ML and Haskell, matching functionality can be given as a construct or on the function definition level.

We assume that each type τ involved has one immediate security-level annotation, which we denote by $tp(\tau)$. Multiple immediate annotations can be handled by complex security lattices. For matching, we require match to be typable, if $tp(\tau) \sqsubseteq tp(\tau_j^i)$ for all i and j, that is, the immediate annotations on sub-values are above the immediate annotations on the value, $tp(\tau) \sqsubseteq tp(\tau')$, and all f_i are typable according to the signature of match. This might seem restrictive, but is powerful enough to capture all cases outside the limitations outlined in the following subsection.

Indistinguishability

We also need to make minimum requirements on what indistinguishability means for values of a type τ and observer level ϕ. Our single requirement is that if two values v_1 and v_2 of type τ are indistinguishable at level ϕ, and $tp(\tau) \sqsubseteq \phi$, then both values have the same root constructor, and all immediate sub-values are indistinguishable with respect to their corresponding types at level ϕ.

4.4.2 Limitations

It turns out that our approach to proving security completeness is not generally applicable to languages with complex datatypes.. As an example, take a language with pairs which have three security annotations: one for each component and one to signal the security level of the identity of the pair. Now take a pair that has a public and a private component, and is itself private. This leads to the public component not being accessible by an attacker (cf. [72, 14, 66]). Indistinguishability

might thus be defined as:

$$\langle x_1, y_1\rangle \sim \langle x_2, y_2\rangle \text{ at } \langle \phi_1, \phi_2\rangle^{\phi_3} \iff$$
$$\phi_3 = H \vee \left(\begin{array}{r} (\phi_1 = L \implies x_1 = x_2) \wedge \\ (\phi_2 = L \implies y_1 = y_2) \end{array} \right)$$

With this, the following computation is noninterferent:

$$\texttt{if } \mathrm{h} > 0 \texttt{ then } \langle 3, 5\rangle \texttt{ else } \langle 4, 5\rangle \ : \ H \to \langle L, H\rangle^H$$

However, this computation has a dependency between high input h and the low output component.

We argue that this is a degenerate case. While the label on the component might be public, for all intents and purposes the pair behaves like it is typed all-confidential. There is no way, neither on the language level, nor the semantics or indistinguishability definition, to access the public component and retain its security level. As such, that typing is syntactically valid, but semantically not meaningful.

Note that there are two obvious problems to overcome. First, one might try to include more context in typing judgments: if the type system knew that one has a dependency between a confidential input and a nominally public output, where the output will be embedded into a confidential container, then one could allow this dependency. However, this seems to break locality and modularity of the type system. Even if this problem can be overcome, more complex systems present aliasing problems on top of the modularity: just because a local container reference is confidential does not automatically imply that all references to the same container in the program are also confidential. If there is even one alias that is public (so that the local alias is typed by subsumption), then allowing the assignment constitutes a definite leak.

We are not aware of a security-typed language that can handle these problems in general, and specifically the above program. In fact, in the following we will show that FlowML, a practical non-trivial security-typed language, cannot type any program that computes this function. We thus limit the theorems to security types such that levels of sub-types are at least as high as those of

enclosing types.

Requirement 33 (Sublevel Monotonicity)**.**

$$\forall \tau, \tau'.\tau' \in \tau \implies tp(\tau) \sqsubseteq tp(\tau'),$$

where $\bullet \in \bullet$ *describes the structural relationship of types.*

FlowML

For details of the setup of FlowML we refer to [72, 73]. We only show the proof regarding the limitation here.

FlowML does not annotate pairs. For the proof, we will thus use sums, which does not change the overall implications. We use a sum of two integers. Instances of sum types are created with the constructors inj_1 for the left sub-type and inj_2 for the right sub-type. This means we are looking for a *Core ML* expression e with free variable h such that:

Requirement 34 (*Core ML* Expression)**.**

- $e[h/0]/\emptyset \rightsquigarrow^* inj_1 0/\mu'$
- $e[h/1]/\emptyset \rightsquigarrow^* inj_1 1/\mu''$
- $\exists M.L, [h : \texttt{int}^H], M \vdash e : (\texttt{int}^L + \texttt{int}^L)^H$

As outlined in Chapter 2, FlowML uses a syntactic approach to prove soundness. Two *Core ML* runs are encoded into one *Core ML*^{2} run by the syntactic construct of brackets $\langle , \rangle$. Each component of the bracket is computed separately.

By soundness and completeness of *Core ML*^{2} our setup implies that $e[h/\langle 0, 1\rangle]/\emptyset$ $\rightsquigarrow^* \langle inj_1 0, inj_1 1\rangle/\mu'''$. Note how the separate inputs ($h/0 \equiv h = 0$ and $h/1 \equiv h = 1$) to e are encoded into the substitution $h/\langle 0, 1\rangle$, and the different outputs into the bracket $\langle inj_1 0, inj_1 1\rangle$.

We denote $e[h/\langle 0, 1\rangle]$ by e_h. By substitution, we have $\exists M.L, \emptyset, M \vdash e_h : (\texttt{int}^L + \texttt{int}^L)^H$. We assume that the heap does not contain brackets in the beginning. This can be argued from the *Core*

ML nature of e and common semantics - the *Core ML*^{2} heap can be represented without brackets because all mapped values are the same in the beginning for both executions. In the remaining part, we will ignore the heap when it is not touched.

We will now show that a typing implies that e is not part of *Core ML*. Namely, if e satisfies all conditions above, it must contain a $\langle, \rangle$ bracket. First an auxillary lemma.

Lemma 35 (Bracket Source)**.** *If $e \rightsquigarrow^* v$, then if v has brackets, then e has brackets.*

Proof. By simple inspection of the semantics. No rule explicitly introduces brackets. □

To actually prove our statement, we need a stronger hypothesis. Let $\tau^{\uparrow}$ denote type τ where we know that security annotations are non-decreasing in type components. For example, $\tau = (\text{int}^L + \text{int}^H)^L$ is $\tau^{\uparrow}$. On the other hand, a type is $\tau^{\downarrow}$ if it is not $\tau^{\uparrow}$, that is, there is a component type relation such that the child has level L and the parent has level H.

Lemma 36 ($e \notin CoreML$)**.** *If $h : \tau_i^{\uparrow} \vdash e : \tau^{\downarrow}$, $\vdash v_1 : \tau_i^{\uparrow}$ and $\vdash v_2 : \tau_i^{\uparrow}$, v_1 differs from v_2 in the high part, $\vdash v_1' : \tau^{\downarrow}$ and $\vdash v_2' : \tau^{\downarrow}$, v_1' differs from v_2' in the low part protected by the high part, μ is $\downarrow$-free, and $e_h = e[h/\langle v_1, v_2\rangle]/mu \rightsquigarrow < v_1', v_2' >$, then e contains $\langle, \rangle$.*

Proof. The proof proceeds by induction over the length of a derivation of $e_h/\mu \rightsquigarrow^* \langle v_1', v_2'\rangle/\mu'''$.

Case Reflexivity of $\rightsquigarrow$: Then e_h is a value. The proof proceeds by inspection of all cases of typing for e_h. The literal cases contradict the execution requirements. The variable case contradicts typing, since $\tau_i^{\uparrow} \neq \tau^{\downarrow}$. The bracket case means that either e is a bracket, or $e = h$, which contradicts typing. For complex values, we inspect the value at the position of the decreasing type. Then at the decreasing type, the component needs to be typed low. If it is typed as a non-bracket, then it violates the execution requirements. If it is a bracket, it violates typing, for brackets are always typed H.

Case β: Then e_h is of the shape $(\text{fix } f.\lambda x.e')v$ and reduces to $e'[x/v][f/\text{fix } f.\lambda x.e']$. It follows that e is of the shape $(\text{fix } f.\lambda x.e'')v'$. By rules of substitution, we have $e'[x/v][f/\text{fix } f.\lambda x.e'] =$

$e''[x/v'][f/\text{fix } f.\lambda x.e''][h/\langle v_1, v_2\rangle]$. By inductive hypothesis, $e''[x/v][f/\text{fix } f.\lambda x.e'']$ contains a bracket. Then there is a bracket in e'' or in v. But then there was a bracket in e.

Case ref & assign: Then e_h is of the shape $\text{ref } v$ or $m := v$. Then either v contains a bracket, or this contradicts the execution requirements.

Case deref: Then the heap must have contained $\langle v_1', v_2'\rangle$, which is a contradiction to our assumption of a $\downarrow$-bracket-free heap at the start of execution.

Case proj: Then e_h is of the form $\text{proj}_j(v_1, v_2)$. Then e is of the form $\text{proj}_j(v_1', v_2')$. By rules of substitution, we have $v_j = v_j'[h/\langle v_1, v_2\rangle]$. By inductive hypothesis, this means that v_j' contains a bracket (see first case), which means that e contains a bracket.

Case case: Then e_h is of the form $(\text{inj}_j v)\ \text{case}\ x \succ e_1\ e_2$, which means that e is of the form $(\text{inj}_j v')\ \text{case}\ x \succ e_1'\ e_2'$, and e_h reduces to $e_j[x/v]$. By the rules of substitution, we have $e_j[x/v] = e_j'[x/v'][h/\langle v_1, v_2\rangle]$. By inductive hypothesis, there is a bracket in $e_j'[x/v']$, which means there is a bracket in e_j' or v'. But that means there is a bracket in e.

Case let: Then e_h has the shape $\text{let } x = v \text{ in } e'$ and reduces to $e'[x/v]$. Then e has the form $\text{let } x = v' \text{ in } e''$. By the rules of substitution, we have $e'[x/v] = e''[x/v'][h/\langle v_1, v_2\rangle]$. By inductive hypothesis, there is a bracket in $e''[x/v']$, which means there is a bracket in e'' or v'. But that means there is a bracket in e.

Case bind: Like the let case.

Case handle: Then $e_h = \text{raise } \epsilon\ v\ \text{handle } \epsilon\ x \succ\ e'$ and reduces to $e'[x/v]$. Then e is of the form $\text{raise } \epsilon\ v'\ \text{handle } \epsilon\ x \succ\ e''$ and $e' = e''[h/\langle v_1, v_2\rangle]$. Then, by rules of substitution, $e'[x/v] = e''[x/v'][h/\langle v_1, v_2\rangle]$, so that the inductive hypothesis can be applied and yields that $e''[x/v']$ contains a bracket. Then e'' or v' contain a bracket, which means that e contains a bracket.

Case $\mathrm{handle} - \mathrm{done}$: Then $e_h = \mathrm{raise}\ \epsilon\ v\ \mathrm{handle}\ e'\ \mathrm{done}$ and reduces to e'. Then e is of the form $\mathrm{raise}\ \epsilon\ v'\ \mathrm{handle}\ e''\ \mathrm{done}$ and $e' = e''[h/\langle v_1, v_2\rangle]$. Then by inductive hypothesis e'' contains a bracket, so that e contains a bracket.

Case $\mathrm{handle} - \mathrm{raise}$: Contradicts the semantical requirement that execution does end in a value.

Case $\mathrm{finally}$: Then $e_h = a\ \mathrm{finally}\ e'$ and reduces to $e'; a$. Note that e' may not raise an exception, or the semantical requirements would be violated. Thus, $a = \langle v_1', v_2'\rangle$. Then e must be of the form $a\ \mathrm{finally}\ e''$, or substitution with h cannot match. But then e includes a bracket.

Case pop: Then a must be $\langle v_1', v_2'\rangle$ by semantical requirements. But then $e_h = E[a]$ contains brackets which cannot be from substituting h for $\langle v_1, v_2\rangle$. Thus, e contained a, which has brackets.

Lifting: A careful inspection shows that the brackets in the specialized lifting cases that match in e_h cannot be from substituting h in e, since the typing does not match. But then e contains brackets.

Case $\mathrm{lift} - \mathrm{context}$: *Note that no other sequencing case is allowed to be applicable.* Then e_h is either a handle or a bind context. Assume first that e_h is a handle context $E[\langle a_1, a_2\rangle]$, that is, $e_h = \langle a_1, a_2\rangle\ \mathrm{handle}\ \epsilon\ x \succ e'$, $e_h = \langle a_1, a_2\rangle\ \mathrm{handle}\ e'\ \mathrm{done}$, $e_h = \langle a_1, a_2\rangle\ \mathrm{handle}\ e'\ \mathrm{raise}$, or $e_h = \langle a_1, a_2\rangle\ \mathrm{finally}\ e'$. Note that *pop* does not apply, so e_h handles a_1 or a_2, so $\exists j.a_j = \mathrm{raise}\ \epsilon\ v$. Then e cannot be of the form $E[h]$, or else substitution of h would not match, so e is of the form $e = \langle a_1, a_2\rangle\ \mathrm{handle}\ \epsilon\ x \succ e''$, $e = \langle a_1, a_2\rangle\ \mathrm{handle}\ e''\ \mathrm{done}$, $e = \langle a_1, a_2\rangle\ \mathrm{handle}\ e''\ \mathrm{raise}$, or $e = \langle a_1, a_2\rangle\ \mathrm{finally}\ e''$. But then e includes a bracket.

Analogously, if e_h is a bind context, then e_h is of the form $\mathrm{bind}\ x = \langle a_1, a_2\rangle\ \mathrm{in}\ e'$, and since *bind* does not apply, $\langle a_1, a_2\rangle$ cannot be a value, so at least one a_j is a $\mathrm{raise}\ \epsilon\ v$. Then

e cannot be of the form $E[h]$, or else substitution of h would not match, so e is of the form $\mathrm{bind}\ x = \langle a_1, a_2\rangle\ \mathrm{in}\ e''$. But then e includes a bracket.

Case $\mathrm{context}$: Then $e_h = E[e_1]$ and reduces to $E[e_2]$, where e_1 reduces to e_2. We only show the bind case here, the handle cases are analogous. We have $e_h = \mathrm{bind}\ x = e_1\ \mathrm{in}\ e'$ reduces to $\mathrm{bind}\ x = e_2\ \mathrm{in}\ e'$. Then e is of the form $\mathrm{bind}\ x = e_1'\ \mathrm{in}\ e''$ with $e_1 = e_1'[h/\langle v_1, v_2\rangle]$ and $e' = e''[h/\langle v_1, v_2\rangle]$. By typing of bind, and subject reduction, e_2 does not contain a free h. Thus, $e_2[h/\langle 0, 1\rangle] = e_2$. So $E[e_2] = (\mathrm{bind}\ x = e_2\ \mathrm{in}\ e'')[h/\langle v_1, v_2\rangle]$. By inductive hypothesis, e_2 or e'' contain brackets. If e'' contains brackets, we are immediately done.

Now, since $E[e_2] : \tau^{\downarrow}$ by subject reduction, we have $e_2 : \tau_2$ for some τ_2. Either $\tau_2^{\uparrow}$ or $\tau_2^{\downarrow}$. In the first case, we can use this with the reduction of e', which yields by induction brackets in e', which is a contradiction. In the second case, we can use the inductive hypothesis on the nested execution $e_1'[h/\langle v_1, v_2\rangle] \to e_2$. This results in brackets in e_1', which means there was a bracket in e.

Case $\mathrm{bracket}$: Analogously to values, it follows that e is a bracket or $e = h$.

This proves that, to satisfy all requirements, e must be a proper *Core ML*^{2} expression including a bracket. □

A simple corollary of Lemma 36 is now that there is no *CoreML* expression that satisfies *all* requirements, that is, is security-typability and semantically equivalent.

Corollary 37 (No Typable Program)**.** *There is no typable Core ML program that takes an H integer as input and produces a sum-value output satisfying requirement 34.*

4.4.3 Security-typed Simulation with Datatypes

The intuition behind our approach is to split computations by output level, allowing a level-separated computation. The final result then needs to be composed from the parts. Separability is a known result for trace-based security. We re-use and extend it to complex datastructures.

In a language-based environment, directly separating by security level is complicated. Since levels are connected to types, which are connected to the structure of values, we instead separate structurally, which implies a level separation. E.g., with the example above, we will find a program that represents $f(x, y)$ as a composition of computations for each pair component:

$$f(x, y) = \text{match } f^L(x, y) \text{ with } \langle x_t, y_t \rangle \Rightarrow \langle f^1(x, y), f^2(x, y) \rangle$$

where $f^1(x, y) = \pi_1(f(x, y))$ and $f^2(x, y) = \pi_2(f(x, y))$. Intuitively, the matching will compute a single (sub-)value at the level of the immediate annotation of that type. Noninterference will enforce that at least the variant chosen is correctly computed at this level. The corresponding matched case will re-compute all sub-values, at their correct levels, and reconstruct the correct value in a typable fashion.

An example of this construction is given in Figure 4.5. The program p on top computes $f(x, y) = \langle x, y + 1 \rangle$, but is not typable. In the transformed program, for brevity we use π to extract components of a pair. The first four blocks compute the low component, while the next three blocks compute the high component, and finally the pair is reconstituted. Note the conceptual similarity to [33]. They perform a similar process at runtime to enforce noninterference, compared to our approach of showing typability in the case when noninterference is given.

4.4.4 Nonrecursive Datatypes

For nonrecursive datatypes, a type τ can be matched statically to any value v it types. We will recompute a (sub-)value corresponding to the structure of its (sub-)type, ensuring typability along the way.

We can recursively generate a function for this whole computation. Note that underlined functions are meta-level functions defining a language-level construct - in a sense they are macros to construct the language-level computation. Assume that f is noninterferent with respect to signature $\tau^i \to \tau$. Here, we assume τ^i is not complex to simplify the presentation. Also, let f^ϕ denote

Program	*Typing*
if $\mathrm{y} = 0$ then $\mathrm{a} := \langle \mathrm{x}, \mathrm{y} \rangle$	$[\mathrm{x} : L,\ \mathrm{y} : H,\ \mathrm{a} : L \times H]$
else $\mathrm{a} := \langle \mathrm{x} + 0, \mathrm{y} + 1 \rangle$;	

(a) Original Program

Code	*Typing*	*High-level equiv.*
$\mathrm{t}_1^{\mathrm{x}} := \mathrm{x}$; $\mathrm{t}_1^{\mathrm{y}} := \mathrm{y}$; $\mathrm{t}_1^{\mathrm{o}} := \mathrm{t}_1^{\mathrm{x}}$	$[\mathrm{t}_1^{\mathrm{x}} : L,\ \mathrm{t}_1^{\mathrm{y}} : H,\ \mathrm{t}_1^{\mathrm{o}} : L]$	$\mathrm{t}_1^{\mathrm{o}} = \pi_1(x, y)$
$\mathrm{t}_2^{\mathrm{x}} := \mathrm{x}$; $\mathrm{t}_2^{\mathrm{y}} := \mathrm{y}$; $\mathrm{t}_2^{\mathrm{o}} := 0$;	$[\mathrm{t}_2^{\mathrm{x}} : L,\ \mathrm{t}_2^{\mathrm{y}} : H,\ \mathrm{t}_2^{\mathrm{o}} : L]$	$\mathrm{t}_2^{\mathrm{o}} = c_0(x, y)$
$\mathrm{t}_3^{\mathrm{x}} := \mathrm{t}_1^{\mathrm{o}}$; $\mathrm{t}_3^{\mathrm{y}} := \mathrm{t}_2^{\mathrm{o}}$; if $\mathrm{t}_3^{\mathrm{y}} = 0$ then $\mathrm{t}_3^{\mathrm{o}} := \langle \mathrm{t}_3^{\mathrm{x}}, \mathrm{t}_3^{\mathrm{y}} \rangle$ else $\mathrm{t}_3^{\mathrm{o}} := \langle \mathrm{t}_3^{\mathrm{x}} + 0, \mathrm{t}_3^{\mathrm{y}} + 1 \rangle$;	$[\mathrm{t}_3^{\mathrm{x}} : L,\ \mathrm{t}_3^{\mathrm{y}} : L,\ \mathrm{t}_3^{\mathrm{o}} : L \times L]$	$\mathrm{t}_3^{\mathrm{o}} = p^L(\mathrm{t}_1^{\mathrm{o}}, \mathrm{t}_2^{\mathrm{o}})$
$\mathrm{t}_4^{\mathrm{o}} := \pi_1\ \mathrm{t}_3^{\mathrm{o}}$;	$[\mathrm{t}_4^{\mathrm{o}} : L]$	
$\mathrm{t}_5^{\mathrm{x}} := \mathrm{x}$; $\mathrm{t}_5^{\mathrm{y}} := \mathrm{y}$; $\mathrm{t}_5^{\mathrm{o}} := \mathrm{t}_5^{\mathrm{y}}$;	$[\mathrm{t}_5^{\mathrm{x}} : L,\ \mathrm{t}_5^{\mathrm{y}} : H,\ \mathrm{t}_5^{\mathrm{o}} : H]$	$\mathrm{t}_5^{\mathrm{o}} = \pi_2(x, y)$
$\mathrm{t}_6^{\mathrm{x}} := \mathrm{t}_1^{\mathrm{o}}$; $\mathrm{t}_6^{\mathrm{y}} := \mathrm{t}_5^{\mathrm{o}}$; if $\mathrm{t}_6^{\mathrm{y}} = 0$ then $\mathrm{t}_6^{\mathrm{o}} := \langle \mathrm{t}_6^{\mathrm{x}}, \mathrm{t}_6^{\mathrm{y}} \rangle$ else $\mathrm{t}_6^{\mathrm{o}} := \langle \mathrm{t}_6^{\mathrm{x}} + 0, \mathrm{t}_6^{\mathrm{y}} + 1 \rangle$;	$[\mathrm{t}_6^{\mathrm{x}} : H,\ \mathrm{t}_6^{\mathrm{y}} : H,\ \mathrm{t}_6^{\mathrm{o}} : H \times H]$	$\mathrm{t}_6^{\mathrm{o}} = p^H(\mathrm{t}_1^{\mathrm{o}}, \mathrm{t}_5^{\mathrm{o}})$
$\mathrm{t}_7^{\mathrm{o}} := \pi_2\ \mathrm{t}_6^{\mathrm{o}}$;	$[\mathrm{t}_7^{\mathrm{o}} : H]$	
$\mathrm{a} := \langle \mathrm{t}_4^{\mathrm{o}}, \mathrm{t}_7^{\mathrm{o}} \rangle$;		

(b) Transformed Program

Figure 4.5: Example Pair-Result Program

the function that results from single-level typing as outlined in the previous section. p is a path to a sub-value/sub-type, which is encoded by a list of pairs for the choice of constructor and immediate sub-value. We denote the type in τ relative to path p by τ'. Note that the concept of paths is purely meta-level, since nonrecursive types can be fully statically described - we only need it to describe the computation recursively.

$$\texttt{let } \tau' = \texttt{nestedSubTree}(p,\ \tau)$$

$$\underline{\text{match}}^f(p,\tau) = \lambda x : \tau^i.\ \underline{\text{extract}}^f(p,\tau)\ f^{tp(\tau')}(x) \quad \texttt{// if } \tau' \texttt{ is not a datatype}$$

$$\underline{\text{match}}^f(p,\tau) = \lambda x : \tau^i. \quad \texttt{// else}$$

$$\text{match } (\underline{\text{extract}}(p,\tau)\ f^{tp(\tau')}(x)) \text{ with}$$

$$\vdots$$

$$c_i\ t^i_1,\ldots,t^i_{n_i} \Rightarrow c_i\ (\underline{\text{match}}^f(p{+\!+}(i,1),\tau)\ x),\ldots,$$
$$(\underline{\text{match}}^f(p{+\!+}(i,n_i),\tau)\ x)$$

The key point of $\underline{\text{match}}$ is the recomputation of f at the level of the currently inspected sub-value denoted by p. To avoid inspection of f, we do a full recomputation, which then requires to extract the sub-value in question - this is the job of $\underline{\text{extract}}$. With the restrictions on τ and indistinguishability, it follows that this recomputation is correct up to the choice of constructor, but not necessarily the sub-values $t^i_1,\ldots,t^i_{n_i}$. We thus recompute the sub-values recursively by extending the path for $\underline{\text{match}}$.

Extraction itself does not need to recompute at each step. To be typable as needed, $\underline{\text{extract}}$ refers to a default value v_{def} for type τ' when the given path does not lead to such a case (we use "default" to stand for the finite number of other cases).

$$\underline{\text{extract}}((),\tau) = \lambda x : \tau.\ x$$

$$\underline{\text{extract}}((i,j) :: p,\tau) = \lambda x : \tau.\ \text{match } x \text{ with}$$
$$c_i\ t^i_1,\ldots,t^i_{n_i} \Rightarrow \underline{\text{extract}}(p,\tau^i_j)\ t^i_j$$
$$\texttt{default} \Rightarrow v_{def}$$

We can now formalize our conjecture for nonrecursive datatypes.

Theorem 38 (Nonrecursive Datatypes)**.** *Assume a language $\mathcal{L}$ and corresponding ground language that fulfill requirements 23, 24, 25, 26, 27, 28 and additionally requirements 32 and 33. This language is security-complete.*

4.4.5 Proof of Nonrecursive Case

We need a range of auxillary predicates and functions. First, to generically handle datatypes we need the following:

Datatype Predicate: $\mathrm{dt}(\tau)$ is true if τ is representing a datatype

Root Constructor: $\mathrm{rt}(c\ t_1, \ldots, t_n) = c$, the root constructor of a datatype term

Parameters: $\mathrm{par}(c\ t_1, \ldots, t_n) = (t_1, \ldots, t_n)$ and $\mathrm{par}(k, c\ t_1, \ldots, t_n) = t_k$

To argue over the structure and execution of the recursive constructions of $\underline{\mathrm{extract}}$ and $\underline{\mathrm{match}}$, we also need predicates and functions over paths and datatypes. Recall that paths are sequences (lists) of choices. A choice is a pair, where the first component denotes the choice of constructor/subtype, and the second component the component thereof. For example, in a value, $(1, 2)$ is the choice of constructor c_1, and parameter 2 thereof. Thus, $(1, 2)$ would match, for example, $c_1\ x, y, z$ (more specifically, the y of it), but not $c_2\ a$. Note that all following definitions are constrained to types τ that are datatypes.

Valid paths: We use the following predicate to denote valid paths in a type or value. Valid paths describe an acceptable sequence of choices. For types, we have

$$\mathrm{vp}((), \tau)$$
$$\mathrm{vp}((i, j) :: r, \tau) \iff \mathrm{dt}(\tau) \wedge i \in \{1, \ldots, n\} \wedge \mathrm{vp}(r, \tau_j^{c_i})$$

For values, we have

$$\mathrm{vp}((), t)$$
$$\mathrm{vp}((i,j) :: r, t) \iff t : \tau \wedge \mathrm{dt}(\tau) \wedge t = c_i^\tau\ t_1, \ldots, t_n \wedge \mathrm{vp}(r, t_j)$$

Type at path: A valid path in a type denotes a subtype. This can be extracted by the following function:

$$\mathrm{pt}((), \tau) = \tau$$
$$\mathrm{pt}((i,j) :: r, \tau) = \mathrm{pt}(r, \tau_j^{c_i})$$

Value at path: Analogously, a valid path in a value denotes a sub-value. (Note that the sub-term is a value by requirements on treated values.)

$$\mathrm{pv}((), t) = t$$
$$\mathrm{pv}((i,j) :: r, c_i\ t_1, \ldots, t_n) = \mathrm{pv}(r, t_j)$$

We first prove some auxillary lemmas for $\underline{\mathrm{extract}}$.

Lemma 39 (Typing of $\underline{\mathrm{extract}}$)**.** $\forall \tau, p. \mathrm{vp}(p, \tau) \implies \underline{\mathrm{extract}}(p, \tau) : \tau \to \mathrm{pt}(p, \tau)$

Proof. By induction on the length of p.

Empty path: This is the identity function and obviously typed.

$p = (i,j) :: p'$*:* The path p is valid in τ. Thus, (i,j) is a valid choice, making the extraction case a valid pattern in the match construct. Furthermore, p' is a valid path in $\tau_j^{c_i}$, so that by inductive hypothesis the type of the nested $\underline{\mathrm{extract}}(p', \tau_j^{c_i}) : \tau_j^{c_i} \to \mathrm{pt}(p', \tau_j^{c_i})$, which by definition is the same as $\tau_j^{c_i} \to \mathrm{pt}(p, \tau)$. Furthermore, by requirement on typing of datatype values, we have $t_j : \tau_j^{c_i}$, so that the whole pattern case has type $\mathrm{pt}(p, \tau)$. Finally, the default case is a constant of type $\mathrm{pt}(p, \tau)$, such that by requirements on match typing, the match can be typed with result type $\mathrm{pt}(p, \tau)$. Thus, $\underline{\mathrm{extract}}(p, \tau) : \tau \to \mathrm{pt}(p, \tau)$.

□

In the following, for brevity, we will abbreviate language-level executions and use functional notation. For example, $\underline{\text{extract}}(p, \tau)\ t$ denotes the result of executing $\underline{\text{extract}}(p, \tau)$ with input t.

Lemma 40 (Valid-Path Extraction)**.**

$$\forall \tau, p, t : \tau. \text{vp}(p, t) \implies \underline{\text{extract}}(p, \tau)\ t = \text{pv}(p, t)$$

Proof. By induction on the length of p.

Empty path: Then $\text{pv}(p, t) = t$ and $\underline{\text{extract}}(p, \tau) = \text{id}$, so that $\underline{\text{extract}}(p, \tau)\ t = t$.

$p = (i, j) :: p'$*:* Then $\text{vp}(p', \tau_j^{c_i})$, $t = c_i\ t_1, \ldots, t_{n_i}$, $\text{vp}(p', t_j)$, and $t_j : \tau_j^{c_i}$. By inductive hypothesis, $\underline{\text{extract}}(p', \tau_j^{c_i})\ t_j = \text{pv}(p', t_j)$. By construction of pv and semantics of matching, we have $\underline{\text{extract}}(p, \tau)\ t = \underline{\text{extract}}(p', \tau_j^{c_i}) = \text{pv}(p', t_j) = \text{pv}(p, t)$.

□

Next, we prove some auxillary results for noninterference that we need for proofs about $\underline{\text{match}}$. Recall that we require for values of type τ to be indistinguishable at level l that $\forall v_1, v_2 : \tau. v_1 = c\ t_1, \ldots, t_n \sim_{\tau,l} c'\ t'_1, \ldots, t'_{n'} = v_2 \wedge tp(\tau) \sqsubseteq l \implies c = c' \wedge n = n' \wedge \forall i. t_i \sim_{t_i,l} t'_i$.

Lemma 41 (Valid-Path Noninterference)**.**

$$\forall v_1, v_2 : \tau, p, l. \text{vp}(p, \tau) \wedge tp(\text{pt}(p, \tau) = \tau') \sqsubseteq l \wedge \text{vp}(p, v_1) \implies \text{vp}(p, v_2) \wedge \text{pv}(p, v_1)\ \sim_{\tau',l}\ \text{pv}(p, v_2)$$

Proof. By induction on the length of p, "adding" at the tail.

Empty path: Then we have $tp(\tau' = \tau) \sqsubseteq l$, $\text{pv}(p, v_1) = v_1$, $\text{vp}(p, v_2)$, and $\text{pv}(p, v_2) = v_2$. Now the basic noninterference definition applies and yields the indistinguishability result.

$p = p' + +(i, j)$*:* Then p' is a prefix and thus valid in τ and v_1. Furthermore, by restrictions on types, namely that security levels are increasing, $tp(\text{pt}(p', \tau) = \tau'') \sqsubseteq l$. It follows that p' is valid in v_2 and $v'_1 = \text{pv}(p', v_1)\ \sim_{\tau'',l}\ \text{pv}(p', v_2) = v'_2$.

We know that $\mathrm{dt}(\tau'')$ (else p would not be valid). That means that the requirement above applies and $\mathrm{rt}(v_1') = \mathrm{rt}(v_2') = c_i$, making p valid for v_2, and $\forall k.\mathrm{par}(k, v_1') \sim_{\tau_k'', l} \mathrm{par}(k, v_2')$, especially for $k = j$. A case decision on $\mathrm{dt}(\tau')$ to check possible paramaters and arguing with $tp(\tau') \sqsubseteq l$ concludes.

□

Lemma 42 (Valid Paths in Single-Level)**.** *Let $f : \tau^i \to \tau$ be noninterferent.*

$$\forall x : \tau^i, p.\mathrm{vp}(p, \tau) \wedge \mathrm{pt}(p, \tau) = \tau' \implies \mathrm{vp}(p, f(x)) \iff \mathrm{vp}(p, f^{tp(\tau')})(x)$$

Proof. By the above lemma. Instantiate $v_1 = f(x)$ and $v_2 = f^{tp(\tau')}$. Use noninterference of f and definition of $\bullet^l$ to gain $v_1 \sim_{\tau, tp(\tau')} v_2$. Now use the lemma to prove both directions. □

Lemma 43 (Roots at a Path)**.** *Let $f : \tau^i \to \tau$ be noninterferent. Then for all $x : \tau^i$ and p,*

$$\mathrm{vp}(p, \tau) \wedge \mathrm{pt}(p, \tau) = \tau' \wedge \mathrm{vp}(p, f(x)) \implies \mathrm{rt}(\mathrm{pv}(p, f(x))) = \mathrm{rt}(\mathrm{pv}(p, f^{tp(\tau')}(x)))$$

Proof. Simple corollary of the above. □

Last, we prove two auxillary lemmas for $\underline{\mathrm{match}}$. Note that we assume $f : \tau^i \to \tau$ here, and denote $\mathrm{pt}(p, \tau)$ with τ'.

We begin by arguing that we can allow proofs by structural induction on types. Namely, we can establish a partial order on datatypes by the length of the longest valid path in them, and then do an induction on the length of such paths.

Lemma 44 (Typing of $\underline{\mathrm{match}}$)**.** *Let $f : \tau^i \to \tau$ a noninterfering function satisfying the requirements of Theorem 38. Let τ' be the type denoted by a path p in τ. Then $\forall p.\mathrm{vp}(p, \tau) \implies \underline{\mathrm{match}}^f(p, \tau) : \tau^i \to \tau'$*

Proof. By induction on τ'.

$\neg\mathrm{dt}(\tau')$*:* Since p is valid in τ, by Lemma 39 we have $\underline{\mathrm{extract}}(p,\tau) : \tau \to \tau'$. By single-level typing, we gain $\underline{\mathrm{extract}}(p,\tau) : [\tau]^{tp(\tau')} \to [\tau']^{tp(\tau')} = [\tau]^{tp(\tau')} \to \tau'$. By construction, $f^{tp(\tau')} : \tau^i \to [\tau]^{tp(\tau')}$. Thus, by composition, $\underline{\mathrm{extract}}(p,\tau) : \tau^i \to \tau'$.

$\mathrm{dt}(\tau')$*:* Analogously to before, and with composition, we gain a typing of the matched expression as $\underline{\mathrm{extract}}(p,\tau)\ f^{tp(\tau')}(x) : [\tau']^{tp(\tau')}$. Thus, all matching cases apply, since they were generated for τ'.

Take an arbitrary pattern in $\underline{\mathrm{match}}(p,\tau)$. For each parameter, we have $p{+}{+}(i,j)$ is a valid path in τ. Furthermore, $\mathrm{pt}(p{+}{+}(i,j),\tau) = \tau^{c_i} + j$ denotes a smaller type than τ'. Thus the inductive hypothesis applies, and we gain a typing $\underline{\mathrm{match}}(p{+}{+}(i,j),\tau) : \tau^i \to \tau_j^{c_i}$. By requirements on typing of complex values, namely that any set of typable parameters makes the construction typable, we have $c_i\ (\underline{\mathrm{match}}(p{+}{+}(i,j),\tau)\ x), \ldots (\underline{\mathrm{match}}(p{+}{+}(i,n_i),\tau)\ x) : \tau'$. Thus, by typing of match, we have $\underline{\mathrm{match}}(p,\tau) : \tau^i \to \tau'$.

□

Lemma 45 (Result of Match)**.**

$$\forall f : \tau^i \to \tau, x : \tau^i.\mathrm{ni}(f) \implies$$
$$\forall \tau'.\forall p.\mathrm{vp}(p,\tau)) \wedge \mathrm{pt}(p,\tau) = \tau' \wedge \mathrm{vp}(p, f(x)) \implies \underline{\mathrm{match}}(p,\tau)\ x = \mathrm{pv}(p, f(x))$$

Proof. Fix f and x. Now induction on τ'.

$\neg\mathrm{dt}(\tau')$*:* Since p is valid in τ and $f(x)$, By Lemma 42, p is valid in $f^{tp(\tau')}(x)$. By Lemma 40, $\underline{\mathrm{match}}(p,\tau)\ x = \underline{\mathrm{extract}}(p,\tau)\ f^{tp(\tau')}(x) = \mathrm{pv}(p, f^{tp(\tau')}(x))$. By Lemma 41, this is indistinguishable to $\mathrm{pv}(p, f(x))$. Since τ' is not a datatype, this means the values are equivalent.

$\mathrm{dt}(\tau')$*:* Fix a valid p. By Lemma 42, p is valid in $f^{tp(\tau')}(x)$. By Lemma 40, $\underline{\mathrm{match}}(p,\tau)\ x = \underline{\mathrm{extract}}(p,\tau)\ f^{tp(\tau')}(x) = \mathrm{pv}(p, f^{tp(\tau')}(x)) = c\ t_1, \ldots, t_n : \tau'$. Since p is valid in $f(x)$, we have $\mathrm{pv}(p, f(x)) = c'\ t'_1, \ldots, t'_{n'} : \tau'$. By Lemma 43, $c = c'$ and $n = n'$.

By definition of $\underline{\text{match}}$ and semantics of match, we have $\underline{\text{match}}(p, \tau)\ x = c\ (\underline{\text{match}}(p + +(i, 1), \tau)\ x), \ldots, (\underline{\text{match}}(p + +(i, n), \tau)\ x)$. It remains to show that the parameters are equivalent to their respective t'_j. Take an arbitrary j. Then by construction, $p{+}{+}(i, j)$ is valid in τ. Also, as seen above, $p{+}{+}(i, j)$ is valid in $f(x)$. Furthermore, $\text{pt}(p{+}{+}(i, j), \tau) = \tau''$ is smaller than τ'. Thus, the inductive hypothesis applies and yiels $\underline{\text{match}}(p + +(i, j), \tau)\ x = \text{pv}(p + +(i, j), f(x)) = t_j$.

□

Now the proof of Theorem 38 is a simple corollary of Lemma 45, instantiated with $p = ()$ and $\tau' = \tau$.

4.4.6 Example

Here we will give a simple example of the construction. We assume a functional, ML-style language with **let** and match (in ML syntax "**case of**") constructs and integers and sums. Anonymous functions are created with **fn**. Let f be a noninterfering function with signature $L \times H \to (L+H)^L$, that is, it takes a public and a confidential input, and produces a public disjoint sum of a public and a confidential option. Elements of sum type are either inl or inr. We annotate local variables with their types for exposition.

```
fn (x:L) (y:H) =>
  let f1^0 : (L + L)^L = (fn x => x) f(x,0) in
  case f1^0 of
    inl _ =>
      let m = fn (x:L) (y:H) =>
        let e = fn (z:(L + L)^L) =>
          case z of
            inl r => (fn x => x) r
          | inr _ => 0
```

```
      in e f(x,0)
    in m x y
  inr _ =>
    let m = fn (x:L) (y:H) =>
      let e = fn (z:(H + H)^H) =>
        case z of
          inl _ => 0
          | inr r => (fn x => x) r
      in e f(x,y)
    in m x y
```

4.4.7 Recursive Datatypes

The approach of the previous section can be extended to recursive datatypes. Recursive datatypes complicate the recomputation process. Namely, the structure of a value cannot exactly be matched statically to the type. Instead, certain structures may be repeated. The key difference is that now the path in a value needs to be handled dynamically. We can model the path with a list of integers, which is a recursive type and thus allowed by the language in question. We use μ types to guide the construction of the corresponding code. μ types allow a binding construct $\mu\, x.\tau$, where x may appear in τ. A common interpretation is that μ types are finite representations of regular trees that are generated when "unfolding" μ types, i.e., replacing x with $\mu\, x.\tau$ in τ. μ types are not explicitly labeled. They inherit to label of τ, such that the tree interpretation is unambiguous. Thus, all "recursions" of a binder have the same security level.

We capture the recursive nature of values with a recursive recomputation in the language. Given a μ type with binders exhibiting variables $x_1, \ldots, x_n$ (for simplicity we assume all variables are unique), we create functions to recompute values of the corresponding μ type. For example, for $t = \mu\, x_1.\texttt{int} * x_1 + (\mu\, x_2.\texttt{bool} + x_2)$, we generate mutually recursive functions f_{x_1} and f_{x_2}

to compute the corresponding parts. To distinguish the current part under computation, these functions take as a parameter a path to the component. Different from the nonrecursive case, this path must actually be maintained at runtime. We can model the path with a list of integers, a list being a recursive type and thus allowed by the language in question. To make the construction typable, a (mutually) recursive construction needs to be typable if calls are made with the right signatures.

Extraction has to follow the path parameter. To ensure minimum typability requirements, i.e., involved functions can be monomorphically typed, we create specialized mutually recursive functions for each start and end type. The start type denotes the type of the input value, while the end type denotes the type of the overall result, i.e., at the end of the path. For example, with t as above, we create extraction functions for $x_1 \to x_1$, $x_1 \to x_2$, and $x_2 \to x_2$.

The actual recomputation for each component now follows the nonrecursive case: extract the value based on the path to the recursive component and inside it, recompute sub-components, create the compound value and return. The only difference now is that in case of a recursive-type sub-component we call the corresponding f_x function with an updated path.

Formalization

For the following definition, we distinguish static, non-recursive paths p^c from recursive paths p^r. We also denote recursive types by their variables in the corresponding μ type. The construction here is for one recursive (sub-)type τ^r of the overall type τ^t. As before, let τ' denote the subtype in τ^r at path p^c.

$$
\begin{aligned}
\underline{\mathrm{rmatch}}_r(p^c) \;=\; & \lambda p^r.\ \lambda x:\tau^i.\ \mid \mathtt{if}\ \ \tau'\ \ \mathtt{is\ not\ a\ datatype} \\
& \qquad \underline{\mathrm{extract}}(p^c,\tau^r)\ \ (\mathrm{extract}_{t\to r}\ p^r\ f^{tp(\tau')}(x)) \\
\underline{\mathrm{rmatch}}_r(p^c) \;=\; & \lambda p^r.\ \lambda x:\tau^i.\ \mid \mathtt{if}\ \ p^c = ()\ \ \mathtt{or}\ \ \tau'\ \ \mathtt{not\ recursive} \\
& \qquad \mathrm{match}\ \ \underline{\mathrm{extract}}(p^c,\tau^r)\ \ (\mathrm{extract}_{t\to r}\ p^r\ f^{tp(\tau')}(x))\ \ \mathrm{with} \\
& \qquad\qquad \vdots \\
& \qquad\qquad c_i\ t^i_1,\ldots,t^i_{n_i}\ \Rightarrow\ c_i\ (\underline{\mathrm{rmatch}}(p^c{+}{+}(i,1))\ p^r\ x),\ldots,
\end{aligned}
$$

$$(\underline{\text{rmatch}}(p^c + +(i, n_i))\ p^r\ x)$$

$$\underline{\text{rmatch}}_r(p^c)\ =\ \lambda p^r.\ \lambda x : \tau^i.\ |\ \texttt{if}\ \ p^c \neq ()\ \ \texttt{and}\ \ \tau' = \tau^s\ \ \texttt{recursive}$$
$$f_s\ \ (p^r + +\underline{\text{encode}}(p^c, \tau^r))\ \ x$$

where $f_s = \underline{\text{rmatch}}_s(())$ for all recursive variables s in the μ type. Note that we use an encoding of paths p^c to integers to simplify language requirements. Since we require finite formulations, there are only a finite number of paths in a recursive type to a recursive sub-type, which can be easily specified as integers. Recursive extraction is similarly extended to yield (mutually) recursive functions, but needs to decode the integer describing the inner path.

$$\text{extract}_{r\to s}\ =\ \lambda p^r.\ \lambda x : \tau^r.$$
$$\texttt{match}\ \ p^r\ \ \texttt{with}$$
$$()\ \Rightarrow\ \underline{if\ r == s\ then}\ \ x\ \ \underline{else}\ \ v^s_{def}$$
$$k :: p'\ \Rightarrow\ \underline{\text{mkExt}}(\tau^r, \tau^s)\ \ k\ \ p'\ \ x$$
$$\texttt{end}$$

where

$$\underline{\text{mkExt}}(\tau^r, \tau^s)\ =\ \lambda k.\ \lambda p^r.\ \lambda x : \tau^r.$$
$$\underline{for\ each\ path\ encoding\ k'\ in\ \tau^r}:$$
$$\texttt{if k==k' then}$$
$$\underline{let\ p' = \text{decode}(k', \tau^r)\ and\ the\ denoted\ type\ be\ \tau^q}$$
$$\text{extract}_{q\to s}\ p^r\ (\underline{\text{extract}}(p', \tau^r)\ x)$$
$$\texttt{else}\ \ v^s_{def}$$

We need a final requirement to ensure typability of this construction. Informally, given a set of program fragments $p_\bullet$ computing functions $f_\bullet$ with signature type $\tau_\bullet$, and given that each program fragment is typable with its corresponding type under the assumption that all the other program fragments are typable, then the whole program is typable.

Requirement 46 (Mutually Recursive Typing)**.**

$$\forall p_\bullet, f_\bullet, \tau_\bullet, k.\, (\forall i. [\![p_i]\!] = f_i : \tau_i \wedge p_1 : \tau_1, \cdots \vdash p_i : \tau_i) \implies \vdash p_k[p_1, \ldots, p_n] : \tau_k$$

Theorem 47 (Recursive Datatypes)**.** *Assume a language $\mathcal{L}$ and corresponding ground language that fulfills requirements 23, 24, 25, 26, 27, 28, 32, 33 and additionally 46. Also assume a program p in the security-typed language that computes a function f, that is ground-typable, but not necessarily security-typable. Furthermore, assume that f is noninterferent with respect to security signature $\tau^i \to \tau$, where τ fulfills the requirements of this section. Let τ be $\mu\, t.\tau'$, i.e., recursive with variable t. Then the program corresponding to the collection of $f_s = \underline{\mathrm{rmatch}}_s(())$ for all s appearing in τ, all corresponding extractions $\mathrm{extract}_{r\to s}$, and starting with recomputation for the root type is security-typable and computes f.*

4.4.8 Proof of Recursive Case

We extend the set of predicates and their definitions from the nonrecursive part in the following way. Note that all definitions here are with respect to a μ typing (short μ) that we leave implicit.

Recursive Type: $\mathrm{rec}(\tau)$ if τ corresponds to a recursive type in μ.

Paths: We extend the definitions of the predicate vp and functions pt and pv to allow paths made up of encodings of recursive descents and explicit choices. We omit the obvious definitions here.

We follow the same structure as in the nonrecursive case, starting with a set of lemmas.

Lemma 48 (Typing of $\underline{\mathrm{mkExt}}$)**.** *If all $\mathrm{extract}_{r\to s}$ are typable as $\mathrm{extract}_{r\to s} : \mathcal{P}^\ell \to \tau^r \to \tau^s$ for all r and s in μ, then $\underline{\mathrm{mkExt}}(\tau^r, \tau^s) : \mathrm{int}^\ell \to \mathcal{P}^\ell \to \tau^r \to \tau^s$ for all r and s.*

Proof. Follows directly by composition and rules of matching. □

Lemma 49 (Typing of $\mathrm{extract}_{r\to s}$)**.** $\ell \sqsubseteq tp(\tau^s) \implies \mathrm{extract}_{r\to s} : \mathcal{P}^\ell \to \tau^r \to \tau^s$

Proof. First, $\ell \sqsubseteq tp(\tau^s)$. Thus, from the security viewpoint, the match is well-formed. We now inspect the patterns. For the case of an empty p^r, this is either identity or a default value. Both can be typed as τ^s (by the constraints). If p^r is not empty, then k has a security value of ℓ, and $p' : \mathcal{P}^\ell$. By Lemma 48 and composition, $\underline{\mathrm{mkExt}}(\tau^r, \tau^s)\ k\ p'\ x : \tau^s$. □

Lemma 50 (Valid-Path Extraction (2))**.**

$$\forall \tau^r, \tau^s, p^r, t. \mathrm{vp}(p^r, t) \wedge \mathrm{pt}(p^r, \tau^r) = \tau^s \implies \mathrm{extract}_{\tau^r \to \tau^s}\ p^r\ t = \mathrm{pv}(p^r, t)$$

Proof. By induction on the execution of $\mathrm{extract}_{r \to s}\ p^r\ t$.

- $p^r = ()$*:* Then $\tau^r = \tau^s$ by valid path definition, so $r = s$. Then $\mathrm{extract}_{r \to s}\ p^r\ t = \underline{if\ r == s\ then}\ t\ \underline{else}\ v^s_{def} = t = \mathrm{pv}((), t)$.
- $p = k :: p'$*:* Since p is a valid path, k encodes a valid non-recursive path p^c in τ^r to some τ^q, and p' is valid in τ^q. Then $\mathrm{extract}_{r \to s}\ p\ t = \underline{\mathrm{mkExt}}(\tau^r, \tau^s)\ k\ p'\ t = \mathrm{extract}_{q \to s}\ p'\ (\underline{\mathrm{extract}}(p^c, \tau^r)\ x)$ by composition semantics. By Lemma 40, we have $\underline{\mathrm{extract}}(p^c, \tau^r)\ x = \mathrm{pv}(p^c, x) = \mathrm{pv}(k, x) = x'$. By inductive hypothesis, $\mathrm{extract}_{q \to s}\ p'\ x' = \mathrm{pv}(p', x')$. By construction of pv, this is equivalent to $\mathrm{pv}(p, x)$.

□

Noninterference lemmas carry over directly from the nonrecursive part and are not repeated here.

Last, we prove two auxillary lemmas for $\underline{\mathrm{rmatch}}$. Again, assume $f : \tau^i \to \tau$, and denote with τ' the type at then end of the path given in the lemma.

Lemma 51 (Typing of $\underline{\mathrm{rmatch}}$)**.** $\forall p^c. \mathrm{vp}(p^c, \tau^r) \implies \underline{\mathrm{rmatch}}_r(p^c) : \mathcal{P}^\ell \to \tau^i \to \tau'$

Proof. Under the assumption that all $\underline{\mathrm{rmatch}}$-calls can be typed as above, this follows by inspection of the cases:

- In the first case, by composition and typing of $\underline{\mathrm{extract}}$ and $\mathrm{extract}_{\to}$.

- In the second case, by composition, typing of match, and type of an extended $\underline{\mathrm{rmatch}}$. Note that the security level of the matched expression is below the pattern computations.

- In the third case, by recursive assumption.

Thus, by recursive typing constraints, $\underline{\mathrm{rmatch}}$ is typable. □

Lemma 52 (Result of $\underline{\mathrm{rmatch}}$)**.**

$$\forall f : \tau^i \to \tau^r, x : \tau^i. \ni f \implies$$
$$\forall p^r, p^c.\mathrm{vp}(p^r + +p^c, \tau^r) \wedge \mathrm{pt}(p^r, \tau^r) = \tau^s \wedge \mathrm{vp}(p^r + +p^c, f(x)) \implies$$
$$\underline{\mathrm{rmatch}}_s(p^c)\ p^r\ x = \mathrm{pv}(p^r + +p^c, f(x))$$

Proof. Fix f and x. Let $\mathrm{pt}(p^r + +p^c, \tau^r) = \mathrm{pt}(p^c, \tau^s) = \tau'$. Now induction on the evaluation of $\underline{\mathrm{rmatch}}_s(p)p^c s\ p^r\ x$.

$\neg\mathrm{dt}(\tau')$*:* Since $p^r + +p^c$ is valid in τ^r, p^r is valid in τ^r, and since $p^r + +p^c$ is valid in $f(x)$, p^r is valid in $f(x)$. By Lemma 42, $p^r + +p^c$ is valid in $f^{tp(\tau')}(x)$, and thus p^r is also valid in $f^{tp(\tau')}(x)$. By Lemma 50, $\mathrm{extract}_{r\to s}\ p^r\ f^{tp(\tau')}(x) = \mathrm{pv}(p^r, f^{tp(\tau')}(x)) = x'$. By construction, p^c is valid in x'. By Lemma 40, $\underline{\mathrm{extract}}(p^c, \tau^s)\ x' = \mathrm{pv}(p^c, x')$. By construction, this is equivalent to $\mathrm{pv}(p^r + +p^c, f^{tp(\tau')})$. By Lemma 41, this is indistinguishable to $\mathrm{pv}(p^r + +p^c, f(x))$. Since τ' is not a datatype, this means the values are equivalent.

$\mathrm{dt}(\tau')$*, and not recursive:* Analogously to the previous case, we can establish $\underline{\mathrm{extract}}(p^c, \tau^s)$ $(\mathrm{extract}_{r\to s}\ p^r\ f^{tp(\tau')}(x)) = \mathrm{pv}(p^r{+}{+}p^c, f^{tp(\tau')}(x)) = c_i\ t_1, \ldots, t_n \sim_{\tau',l} \mathrm{pv}(p^r{+}{+}p^c, f(x))$, meaning that they have the same root constructor. It remains to show that the parameters are equivalent to the corresponding values.

Take an arbitrary j. It follows that $p^r + +(p^c + +(i, j))$ is valid in τ^r and $f(x)$. Then by inductive hypothesis we have $\underline{\mathrm{rmatch}}_s(p^c + +(i, j))\ p^r\ x = \mathrm{pv}(p^r + +p^c + +(i, j), f(x))$.

$\mathrm{dt}(\tau')$*, and recursive:* Since $p^r + +p^c$ is valid, and p^c denotes a recursive sub-type τ^q in τ^s, there is an encoding k of p^c such that $p^r + +k$ has the same meaning as $p^r + +p^c$ and is

valid. Now, $\underline{\mathrm{rmatch}}_s(p^c)\ p^r\ x = f_q\ (p^r + +k)\ x = \underline{\mathrm{rmatch}}_q(())\ (p^r + +k)\ x$. By inductive hypothesis, $\underline{\mathrm{rmatch}}_q(())\ (p^r + +k)\ x = \mathrm{pv}(p^r + +k, f(x)) = \mathrm{pv}(p^r + +p^c, f(x))$.

□

Now the proof of Theorem 47 is a simple corollary of Lemma 52, instantiated with $p^c = ()$, $p^r = ()$, and $\tau' = \tau$.

4.5 References & Objects

We will treat an object-oriented language with references here. Object-oriented languages introduce additional constructs that need to be handled. While record-like behaviour can already be handled, in general OO is stateful and requires references. This complicates matters and requires further restrictions on languages that our technique can support.

For one, most languages with references only allow limited interactions with references. Allocation of new locations can usually not be influenced directly on the language level. This makes exact recomputation impossible. Our technique can thus only simulate correctly up to renaming of heap locations. This implies that the identity function cannot be simulated correctly: while input and output would themselves be functionally equivalent, the simulation would return a *new* heap location. We see this as a minor disadvantage. We want to use the simulation to replace an embedded program that is cleanly separated from the host. It seems reasonable to require that any objects returned from the embedded program are independent of the inputs. This is usually the case when the embedded program cannot "call back" into the host program.

Any recomputation in parts will repeatedly invoke the original computation at certain levels. This may create several temporary objects polluting the heap, which, of course, do not appear in the original computation. Our technique is thus only correct up to locations reachable from the result of the function.

Also, stateful computation allows side effects. In this case, a side effect may change the input values. We can contain side effects if we can create temporary "clones" of the relevant inputs and

use those for computations. This means that the language needs to guarantee that two suitably related inputs, e.g., clones, compute suitably related outputs. We formalize this as the shape of the part of the heap reachable from the inputs, which corresponds to the first restriction. This, however, forbids any reflective language constructs.[5]

Last, one main principle of object-oriented programming is encapsulation, or hiding of state. This collides with our approach - to correctly recompute an object, all hidden state needs to be recomputed. This means it needs to be accessible. We do not care about the concrete form of accessing state, both accessor methods and direct field accesses are fine. While this might seem like a big restriction, we argue that in practice all state can be made accessible, and still maintain the same functional program (e.g., through principled access). Thus, a simulation in possible, and will guarantee noninterference even for now accessible state. Overall, the simulation is only a theoretical vehicle, and implies that the original embedding with state restrictions is safe, too.

4.5.1 Objects & Heaps

We formalize state through the concept of heaps. Heaps, denoted by $\mathcal{H}$, are mappings of locations, denoted by ℓ, to values. Values are extended to include locations. nil is a special value that is not mapped by any heap. Typings may contain a heap typing that assigns types to locations. Reference types are composed of the type of values that can be stored at the location, as well as the security level of location value itself.

We formalize objects along the lines of [14]. This is a class-based setting. All objects belong to a class which defines which methods and fields are available in the object. Thus, object values can be interpreted as a record of a class tag and fields. Classes induce class types. We only allow primitive types (e.g., `int`) and class types. Objects are stored as references, i.e., a class-type field in an object record contains a location value. We assume that objects can be initialized with default types. We leave it open whether this happens implicitly (as in [14]), or if constructors need explicit values (which we can provide: simple values for primitive types and nil for objects). We use

[5]Note that we are not aware of any security-type systems for reflection.

$new\ C$ to denote object creation. As mentioned, we assume objects are mutable so that we can incrementally update state. We denote accessing state with the typical field syntax $o.f$, but note that there is no technical difference to accessor methods or other techniques. As in [14], we assume that there is a ground type system that ensures that well-typed programs do not go wrong, that in well-typed programs and heaps references always contain location values pointing to values in the heap of the corresponding type, and no runtime exceptions happen.

We assume a simple typing of classes: all fields, method parameters and method result types are security types. Classes contain an annotation that defines the security level of self-references. Methods contain an annotation that bounds the side effect of that method. We assume methods are typable with respect to their annotated signature: here we focus on the typability of a whole program. Wrapping methods, which needs careful isolation of side effects, is future work.

We assume the language may support inheritance/subtyping. In this case, we expect a matching-like construct that enables us to handle specific objects according to their actual class, similar to variants for datatypes (we will reuse this notation). For example, dynamic casts and `instanceof` expressions allow this functionality. To have a precise and complete recomputation, this requires a closed world with a known class hierarchy.

We also assume that the language allows a "main" program fragment that defines a program's behaviour. This might be a specific method in a specific class, e.g., the `main` method in a Java program, or a simple fragment that does not need to be wrapped in a class.

4.5.2 Reachability, Equivalence & Indistinguishability

As outlined above, we restrict our attention to languages that restrict computation to reachable values. We formalize reachability as a set parameterized over a heap and starting location in said heap. The reachable set $\mathcal{R}$ of $\mathcal{H}$ and ℓ is the smallest set closed under the following rules:

1. $\ell \in \mathcal{R}$

2. if $\ell \in \mathcal{R}$ and $\mathcal{H}(\ell) = o$ an object of class C, then for all class-typed fields $f$of C we have

$o.f \in \mathcal{R}$

For our treatment, we require $\mathcal{R}$ to be finite.

To define equivalence for computations, we inspect inputs and results. These are pairs of a heap and an object, which define a reachable portion of the heap. The actual locations are not important for our equivalence. We only want to require the reachable heap parts to be isomorphic, and primitive-typed fields to be equivalent. We can formalize this with respect to a bijection ρ between locations. Two primitive values of type τ are equivalent if they are identical. Two object values o_1 and o_2 of class type C are equivalent with respect to ρ if they agree on all primitive-typed fields, and if for all class-typed fields f we have $o_1.f \ \rho \ o_2.f$. Two heap-object pairs $(\mathcal{H}_1, o_1)$ and $(\mathcal{H}_2, o_2)$ are equivalent if there exists a bijection ρ over the reachable locations $\mathcal{R}_{\mathcal{H}_1,o_1}$ and $\mathcal{R}_{\mathcal{H}_2,o_2}$ such that o_1 is equivalent to o_2 with respect to ρ, and for all locations $\ell_1 \ \rho \ \ell_2$, $\mathcal{H}_1(\ell_1)$ is equivalent to $\mathcal{H}_2(\ell_2)$ with respect to ρ. We denote equivalence by $\equiv$.

If two locations are indistinguishable with respect to bijection ρ and level ϕ and their level is at most ϕ, then they are in relation with respect to ρ. If two object values are indistinguishable with respect to bijection ρ and level ϕ, then all fields are indistinguishable at ϕ with respect to ρ. These are standard. Now, along the lines of datatypes we require that if two references to objects of class C are indistinguishable with respect to ϕ and ρ, and the security annotation is at most ϕ, then the classes of both objects are identical. We require that indistinguishability for heaps is with respect to its reachable part. That is $(\mathcal{H}_\infty, o_1)$ is indistinguishable to $(\mathcal{H}_\in, o_2)$ at level ϕ if there exists a bijection ρ between the reachable sets and o_1 and o_2 are indistinguishable with respect to ρ.

4.5.3 Computation

We define as a computation a function from a heap $\mathcal{H}$ and object in the heap, to a result heap $\mathcal{H}'$ and result object. A program computes f if reduction of the main fragment with variables bound to the input object and given heap results in the result heap and result object. As mentioned above, we assume correct executions. Thus we will leave typing constraints implicit here. All definitions are predicated on heaps and input objects correct with respect to f, that is $\mathcal{H}$ etc. range only over

valid input states for f.

We formalize our requirements on the treated programs in the following way. A function f is ok, if:

- $\forall \mathcal{H}, \mathcal{H}', o, o'. f(\mathcal{H}, o) = (\mathcal{H}', o') \implies \mathcal{R}_{\mathcal{H},o} \cap \mathcal{R}_{\mathcal{H}',o} = \emptyset$
- $\forall \mathcal{H}_1, o_1, \mathcal{H}_2, o_2. (\mathcal{H}_1, o_1) \equiv (\mathcal{H}_2, o_2) \implies f(\mathcal{H}_1, o_1) \equiv f(\mathcal{H}_2, o_2)$

We define a simulation g to be correct for f, if $\forall \mathcal{H}, o. f(\mathcal{H}, o) \equiv g(\mathcal{H}, o)$.

4.5.4 Security-typed Simulation with Heap Objects

We start from the approach for recursive datatypes outlined in Section 4.4.7. Namely, starting from the result object, we walk the reachable parts of the heap. At each step we recompute the current object in question under the reference's security level. By requirements on indistinguishability, the extracted object will now be of the exact class, such that matching allows us to decide what to recompute. Primitive fields can be computed immediately. References are resolved recursively, extending the current path. Since the reachable part of the heap is required to be finite, this process will terminate. There are several noteworthy details that need to be carefully crafted.

Recomputation at a level requires two things. First, we need a copy of the program with all annotations at that level. This means all classes have to be duplicated. Typing is guaranteed since ground typing can be inferred from the original output, and new annotations are typable by single-level requirement. Second, the input has to be duplicated at the current level. This requires a walk and inspection of security levels. Since we only need to duplicate values up to the level, this is typable. We can construct a bijection ρ while copying, associating each original object with its copy. We end up with an extended heap $\mathcal{H}'$ and input object o' such that $(\mathcal{H}, o)$ is indistinguishable to $(\mathcal{H}', o')$, which implies the required noninterference.

A heap may contain cyclic structures. A walk needs to detect cycles to terminate. The process is complicated by the fact that the original result will be recomputed in every step. Our solution is to store a list of objects seen when extracting the current object from the result along the current

path. This list can be typed with a single level, since the result object and heap are typed at a single level. A "back edge" is detected when the current extracted object is present in the list. To also detect "cross edges", we must also walk all completed paths again. This can be implemented through a deterministic processing order of fields in objects.

To break the cycle, we return the object constructed earlier. This is the main reason why the treatment of datatypes (value is created late) differs from objects (value is created early). However, it remains to be seen that we can *find* the constructed object in a typable manner. Our solution is to extend the list structure for cycle detection. Not only do we store the object, but also the path that reached it. Since the result is at a single level, this path is a single-level list at that level. If we update objects early, that is, a child adds itself to its parent, and have access to the root, we can use the path to retrieve the recomputed object in a typable way.

4.5.5 Formalization

Different from the previous treatment, we only present high-level pseudo-code here, since there are too many details for a detailed exposition. We assume there is an encoding of field descriptors, for example, into integers. Paths can then be defined as lists over this encoding. For clarity, we will use names f.

We start with the actual recomputation.

```
Req: ℓ0 ⊑ ℓA ⊑ ℓ ⊑ ℓ'
omatch_{A.fp→B,ℓ,ℓA}(p: Path^ℓ, x: τ^i, start_copy: Object^ℓ0, parent: A^ℓA): B^ℓ
                                                    with side effects ℓ
  Object^ℓ start := f^ℓ(x)

  DoneList^ℓ done := new DoneList
  Object^ℓ cur := extract_ℓ(p, start, start_copy, done)
  for ⟨ Object^ℓ o_orig, Object^ℓ o_copy ⟩ in done:
    if cur == o_orig then
```

```
      parent.f_p := (B)o_copy
      return (B)o_copy

  if cur instanceof B_1 then
    B_1^ℓ copy := new B_1
    parent.f_p := copy

    for all primitive fields f:C^ℓ' in B_1:
      Path^ℓ' p' := p ++ f
      copy.f := extract'_ℓ(p', f^ℓ'(x), f^ℓ'(x), done)

    for all non-primitive fields f:C^ℓ' in B_1,
        ordered by ≺:
      Path^ℓ' p' := p ++ f
      omatch_{B_1.f→C,ℓ',ℓ}(p', x, start_copy, copy)

    return copy
  else if ...
```

A specialized version needs to start the copy at an empty path. This definition needs several auxillary algorithms we describe in the following.

```
extract_ℓ(p:Path^ℓ, start:Object^ℓ, start_copy:Object^ℓ,
          done:DoneList^ℓ):Object^ℓ with side effects ℓ

  if p is empty then
    return start
  else
```

```
add start/start_copy to done

Field f := head of p
Path^ℓ p' := tail of p

extract all fields of start/start_copy before f and below ℓ
into done

if f == f_1 then // first field of start
  if start instanceof C_1 then
    return extract_ℓ(p', ((C_1)start).f, ((C_1)start_copy).f, done)
  else if start instanceof C_2 then
    return nil
  else if ...
else if ... // second, third, ... field of start
return nil
```

The extraction of all previous fields is a simple inspection of the current object's class to find all fields below the given level, and do an unconditional recursion into those fields. The two branches for type checking represent matching *all* classes in the program and show the two possible cases: in the first case, C_1 has a field named f_1 at or below ℓ, and we recurse; in the second case, C_2 either does not have this field, or it is not below ℓ. Last, finding an object from a start object is basically extraction without the done-list.

```
find_ℓ(p: Path^ℓ, start: Object^ℓ): Object^ℓ with side effects ℓ
  if p is empty then
    return start
  else
```

```
Field f := head of p
Path^ℓ p' := tail of p

if f == f_1 then // first field of start
  if start instanceof C_1 then
    return find_ℓ(p', ((C_1)start).f)
  else if ...
else if ... // second, third, ... field of start
return nil
```

Note that in $\mathrm{extract}_\ell$, omatch and find_ℓ, type matching is ordered such that subclasses are tested before superclasses. Thus, matches will be precise.

Note that omatch depends on a total ordering $\prec$ of fields of a class. The only constraint on this ordering is that fields are ordered such that if an object o is reachable from two fields f_1 and f_2 with paths p_1 and p_2, respectively, then if the level at the end of p_1 is less than the level at p_2, then $f_1 \prec f_2$. This ensures that the depth-first approach is correctly typed.

Theorem 53 (Objects)**.** *Assume a language $\mathcal{L}$ and corresponding ground language that fulfill all the requirements of the previous sections, as well as the requirements in this section. Also assume a program p in the security-typed language that computes a function f, that is ground-typable, but not necessarily security-typable. Furthermore, assume that f is ok and noninterferent. Then the program corresponding to the computation outlined above is security-typable and simulates f correctly.*

4.5.6 Proof

We will start with typability.

Lemma 54 (Typability of extract & find)**.** *The functions $\mathrm{extract}_\ell$ and find_ℓ can be typed for all ℓ.*

Proof. Note that all storage (variables and heap locations) are uniformly typed at ℓ. Also, the recursive calls are made with parameters of the exact types of the signature, which implies ground typability. Thus, by single-level requirement, the functions are typable according to their signatures. □

Lemma 55 (Typability of omatch)**.** *The set of function* omatch *corresponding to all combinations of parent classes, fields and respective children, is typable.*

Proof. Assume that the functions are typable. We will show that under this assumption, all omatch functions are typable. We pick an arbitrary function with A, f_p and B, such that the level requirements are fulfilled.

- $start_{copy}$'s level is at $\ell_0 \sqsubseteq \ell$ and can be raised to ℓ. Now the operations in the first two blocks (recomputation of f at level ℓ, loop check) are typed at a single level, ℓ. This is possible by single-level requirement.
- Each type check, object creation and parent update are done at level ℓ. This is possible by single-level requirement.
- All fields f in B or any of its subclasses have security levels $\ell' \sqsupseteq \ell$ by requirement on treated types. Thus, p' is an upgrade of p and thus typable. Now the call to the recursive omatch is according to its signature, that is typable. Its side effect is at $\ell' \sqsupseteq \ell$.

Thus, omatch is typable, with side effects at ℓ or above. □

Next we will show that the recomputation is *correct*. We start with termination (assuming that f terminates for x).

Lemma 56 (Termination of extract & find)**.** *If* p *is not cyclic and finite, then* $\mathrm{extract}_\ell$ *and* find_ℓ *terminate for all inputs.*

Proof. We begin with find_ℓ. Since p is not cyclic and finite, every recursive call decreases the length of the path to look for by one. Thus, in a finite amount of steps, either the base case will be

reached and end the computation, or an incorrect field for the current object is found and terminates the computation.

The argument for $\mathrm{extract}_\ell$ is similar for the recursive calls. Furthermore, we have to show that previous-field extraction terminates. This is the case because 1) the reachable heap is finite by requirement, and 2) each extracted element is put into the done-list, so that each element is only handled at most once. □

Lemma 57 (Extraction result is finite)**.** *The done-list result of* $\mathrm{extract}_\ell$ *is finite, if it starts finite.*

Proof. By Lemma 56, $\mathrm{extract}_\ell$ terminates in a finite amount of steps. In each step, only a finite amount of object pairs is added to the done-list. Thus, the resulting list remains finite. □

Lemma 58 (Termination of omatch)**.** *If* p *is not cyclic and finite, then* omatch *terminates for all inputs.*

Proof. By assumption, $f(x)$ terminates, thus, by noninterference, $f^\ell(x)$ terminates. Furthermore, by requirements, it leaves the heap reachable from x intact, so that this is an invariant over all calls. The result of $\mathrm{extract}_\ell$ is finite by Lemma 57. Thus, there is a finite amount of object pairs to iterate over and compare. If the current object was handled before, this is detected correctly by Lemma 61. Thus, omatch recreates each object and recurses on it at most once. If no original object matches the current object, omatch compares against a finite number of classes (finite, closed-world assumption). In case there is a match for class B_i, the algorithm iterates over all fields of class B_i, of which there is a finite amount. For each field, it extends the current path. This increases the part of the heap marked done with respect to $\mathrm{extract}$. Since the reachable heap is finite, this means each recursive step decreases the size of the heap not recomputed yet. After a finite amount of steps, all the heap is touched and recomputed. The algorithm will then terminate. □

Last, we will show correctness.

Definition 59 (Valid Path (OO))**.** *A path* p *is valid with respect to object* o*, if* p *is the empty path* $()$*, or* p *is* $f :: p'$*, where* f *is a valid field of* o*, and* p' *is valid in* $o.f$*.*

Lemma 60 (Paths are valid)**.** *In a computation starting with $p = ()$, all intermediate paths are valid, that is, describe a valid path in the heap.*

Proof. Obviously, $p = ()$ is valid for any recomputation. Now do an induction on the execution. By construction, the paths p' constructed before recursion are valid in the heap reachable from $start$, since p is valid and denotes cur, and cur has field f. Now, the recursive step is recomputed at level $\ell' \sqsupseteq \ell$, yielding $start'$. All references along p' are at level ℓ' or below. Thus, by noninterference, the heap starting from $start'$ is equivalent on path p'. Thus, p' is valid from $start'$. □

Lemma 61 (Loop-Detection is correct)**.** *Loop detection is correct in* omatch*, that is, the loop in the second block finds a matching object if and only if an object at the same location has been handled before (or is still being handled).*

Proof. By Lemma 60, in any step p is valid in the heap reachable from $start$. Furthermore, by ordering of the field traversal, we have that each object reachable from $f(x)$ is handled *first* with a path p_o such that the level ℓ is minimal.

A loop means that there are at least two paths from $f(x)$ to the same location, w.l.o.g. we restrict to exactly two. Let ℓ_1 and ℓ_2 be the levels of references at the end of the corresponding paths p_1 and p_2. Then by requirements, we have $\ell_1 \sqsubseteq \ell_2$ or $\ell_2 \sqsubseteq \ell_1$. Then by ordering, ℓ_1 is handled before ℓ_2. Since $\ell_2 \sqsupseteq \ell_1$, and p_1 is at ℓ_1, p_1 is valid with respect to $start_2$, and valid in $f(x)$. Then p_1 is picked up in $\mathrm{extract}_{\ell_2}$, either as a prefix of p_2 or a predecessor. Thus, the loop is detected, and the right element is returned.

On the other hand, if a loop is detected, it means that $\mathrm{extract}_{\ell_2}$ picked up an object reachable along either the prefix of p, or along a previously handled field. Thus, since the object locations match, there are two paths toward the current object. Since the reachable heap up to ℓ_2 is correct with respect to $f(x)$, this means there is a loop in $f(x)$, too. Thus, the detection is correct. □

Lemma 62 (Correctness of find)**.** *If p is a valid path, then* $\mathrm{find}_\ell(p, start) = \mathrm{pv}(p, start)$.

Proof. By induction on p. If p is empty, then the statement is trivially true. If $p = f :: p'$, then because p is valid, $start$ is of a class C that has a field f. Thus, one of the type checks

will be true. Since they are ordered in reverse order, C will match before any superclass of C. Thus, the right case will match. Now, by definition, p' is valid in $start.f$. So, $\mathrm{find}_\ell(p, start) = \mathrm{find}_\ell(p', start.f) = \mathrm{pv}(p', start.f) = \mathrm{pv}(p, start)$. □

Definition 63 (Handled nodes)**.** *Let p be a valid path in o. The the set of objects o' such that there is a path $p' \prec p$ that is valid in o and $\mathrm{pv}(p', o) = o'$ is called the handled nodes of o up to p. We denote this set by $H(o, p)$.*

Definition 64 (Correct up to path)**.** *If p is a valid path in an object o and an object o', then o is correct up to p with respect to o' if the graphs induced by $H(o, p)$ and $H(o', p)$ are isomorphic (preserving the reaching paths) and related objects are of the same class.*

Lemma 65 (Correctness of extract)**.** *If p is a valid path for $start$, and $start_{copy}$ is correct up to p, then*

1. $\mathrm{extract}_\ell(p, start, start_{copy}, done) = \mathrm{pv}(p, start)$*, and*
2. *for all valid paths $p' \prec p$ with $tp(\mathrm{pt}(p', \tau)) \sqsubseteq tp(\mathrm{pt}(p, \tau))$, $\mathrm{pv}(p', start) \in done$ after $\mathrm{extract}_\ell(p, start, stat_{copy}, done)$*
3. *for all valid paths $p' \prec p$ with $tp(\mathrm{pt}(p', \tau)) \sqsubseteq tp(\mathrm{pt}(p, \tau))$, $\mathrm{pv}(p', start_{copy}) \in done$ after $\mathrm{extract}_\ell(p, start, stat_{copy}, done)$*

Proof. All at the same time by induction on p. If p is empty, then the statement is trivially true. If $p = f :: p''$, then because p is valid, $start$ is of a class C that has a field f, and $start_{copy}$ is of class C because it is correct up to p. Thus, p'' is valid in both $start.f$ and $start_{copy}.f$.

1. Thus, some of the type checks will be true, and it will be cases with a recursion. Since they are ordered in reverse order, C will match before any superclass of C. Thus, the right case will match. So, $\mathrm{extract}_\ell(p, start, done) = \mathrm{extract}_\ell(p', start.f, done') = \mathrm{pv}(p', start.f) = \mathrm{pv}(p, start)$.

2. Since p'' is valid in $start.f$, for all $p''' \prec p''$ and valid in $start.f$, $\mathrm{pv}(p''', start.f) \in done$ after the recursive call (see 1.). Now, by definitions of $\prec$ and validity, we can extend each such p''' to $f :: p'''$. Then $f :: p''' \prec f :: p'' = p$, and $\mathrm{pv}(p''', start.f) = \mathrm{pv}(f :: p''', start) \in done$. Now, the only other $\prec$ paths are by construction the ones starting with f' such that $f' \prec f$. These are collected in the extraction-of-fields-before block.

3. Analogously to the previous case, with the argument that $start_{copy}$ is correct up to path p.

□

Lemma 66 (Correctness of omatch up to path)**.** *If p is a valid path for $f(x)$, and $start_{copy}$ is correct up to p, and $p = p' + +f$ and $\mathrm{pt}(p', \tau) = A$ and $\mathrm{pt}(p, \tau) = B$, and let ℓ_A and ℓ_B be the corresponding levels of the references. Then after $\mathrm{omatch}_{A.f \to B, \ell_B, \ell_A}(p, x, start_{copy}, \mathrm{pv}(p', start_{copy}))$, $start_{copy}$ is correct up to all valid cyclic-free extensions of p.*

Proof. By induction on the execution of omatch. If omatch returns with a detected loop, then there are no cycle-free extensions of p. If there is no cycle, then cur is indistinguishable from $\mathrm{pv}(p, f(x)) = cur'$ by noninterference. This means that cur is of the same class as cur'. It follows that the newly created object $copy$ is of the same class as cur'. Furthermore, by correctness up to p and $p' \prec p$, $\mathrm{pv}(p', start_{copy})$ is related to $\mathrm{pv}(p', f(x))$. We connect $copy$ to $\mathrm{pv}(p', start_{copy})$. Now $start_{copy}$ is correct up to $p + +f'$ for f' minimal in B. Now inspect each iteration of the loop over the fields. Before an iteration for f', we have $start_{copy}$ correct up to $p + +f'$. Furthermore, $p + +f'$ is valid. Thus, by inductive hypothesis, after the recursive call, $start_{copy}$ is correct up to all valid cyclic-free extensions of $p + +f'$. This means it is correct up to $p + +f''$ where f'' is the next field. Thus, when the loop concludes, $start_{copy}$ is correct up to all valid cyclic-free extensions of p. □

Lemma 67 (Correctness of omatch)**.** *Execution of omatch is correct for f.*

Proof. Simple corollary of Lemma 66, since the start is instantiated with $p = ()$, and any path is an extension of $()$. □

4.6 Example Languages

This section briefly describes three case studies which demonstrate that our formalization and requirements permit such different paradigms as imperative, functional and object-oriented languages.

4.6.1 Volpano, Smith & Irvine

VSI [89] is based on a simple WHILE language based on integers. It fits the development in Section 4.3. Erasure and lifting functions are straightforward for VSI, since only types are annotated. We will show Requirements 25 and 28 in more detail.

Requirement 25 follows from the polymorphic setup of the type rules and can be formally proved by induction.

Lemma 68 (While Single-level Typability)**.** *For all programs* c *and expressions* e *and security levels* ℓ*, there is* Γ_ℓ *such that* $\Gamma_\ell \vdash c : \ell\,\mathrm{ok}$ *and all variables are mapped to* ℓ *in* Γ_ℓ*.*

Proof. Let $\mathcal{V}$ be the set of variables mentioned in c or e. Then define Γ_ℓ as mapping all variables in $\mathcal{V}$ to ℓ. Now proceed by structural induction on c and e. We show select cases.

Literal: The typing rule is polymorphic in the security level. Thus, typing at ℓ is permissible.

Variable: The variable x is an element of $\mathcal{V}$. Thus, $\Gamma_\ell(x) = \ell$.

Assignment: By inductive hypothesis, the expression can be typed as ℓ. Furthermore, analogously to the previous case, the variable is typed as ℓ. Thus, assignment is permissible at $\ell\,ok$.

Condition: By inductive hypothesis, the condition as well as the branches are typable at ℓ. Thus, if is typable at $\ell\,ok$.

□

Variable assignment represents the projection function of requirement 26, typable by the assignment rule. Assignment of an integer literal represents the constant function of requirement 27, and can be typed at the output variable level.

Lemma 69 (Assignment Typings)**.** *For all variables x and x' and security level ℓ, if $\Gamma(x) \sqsubseteq \Gamma(x') = \ell$, then $\Gamma \vdash x' := x : \ell$ ok.*

Proof. By $T\text{-}Var$, $\Gamma \vdash x : \Gamma(x)$ and $\Gamma \vdash x' : \Gamma(x')$. By assumption, $\Gamma(x) \sqsubseteq \Gamma(x')$. With $\ell = \Gamma(x')$ and $\ell' = \Gamma(x)$, an application of $T\text{-}Ass$ yields the result. □

The next lemma states the existence of a typed program that computes the composition of two given typed programs.

Lemma 70 (While Composition)**.** *Given two programs c_a and c_b that are typed under Γ_a and Γ_b and compute f_a and f_b, respectively, where outputs of c_b agree with inputs of c_a, there exists a program $c_{a \circ b}$ that is typed under $\Gamma_{a \circ b}$ and computes $f_{a \circ b} = f_a \circ f_b$. Furthermore, $\Gamma_{a \circ b}$ agrees with Γ_a and Γ_b under respective renamings of variables.*

Proof. For simplicity we assume that c_a and c_b agree on the variables that are used to pass results, that is, outputs of c_b with respect to f_b are named the same as inputs of c_a with respect to f_a. We denote those variables by $\overrightarrow{y}$. No other variables are shared. Note that this can be accomplished by consistently renaming variables. Now $\Gamma_a(x) = \Gamma_b(x)$ for all variables x in $\overrightarrow{y}$.

Let $\Gamma_{a \circ b}(x)$ be $\Gamma_a(x)$ if x appears in c_a, and let $\Gamma_{a \circ b}(x)$ be $\Gamma_b(x)$ if x appears in c_b. Let $c_{a \circ b} = c_b; c_a$. Inspection of the semantic rule for sequencing shows that this program fulfills the functional requirements, that is, computes $f_{a \circ b}$.

By construction, c_a and c_b can be typed under $\Gamma_{a \circ b}$. Namely, $\Gamma_{a \circ b}$ is a weakening of both Γ_a and Γ_b. Thus, by inspection of the typing rule for sequencing $c_{a \circ b}$ is typable under $\Gamma_{a \circ b}$. □

We thus have as a corollary to Theorem 29 that While is security-complete.

Corollary 71 (While Security Completeness)**.** *If a function f is computable in WHILE, and noninterferent under a signature given by Γ, then there exists a WHILE program c that is typable under a typing environment Γ' that is an extension of Γ.*

4.6.2 FlowML

FlowML [72, 73] is based on a core functional ML fragment including references, pairs, sums and exceptions. For simplicity of the functional interpretation we do not treat exceptions and references here. FlowML fits the development in Section 4.4.4. Lifting, erasure and single-level typing follow from the polymorphic setup of the rules. Projection is provided by a simple variable, while constants can be freely formed. Composition is provided by variable substitution, which may be combined with renaming and weakening to fulfill the requirements. Extraction for pairs is provided by typed projection, and a `case` construct allows to distinguish variants. However, basic noninterference cannot be lifted to abstractions, so that we cannot support arrow types (c.f. [46]).

4.6.3 Banerjee & Naumann

It is easy to extend the work in Section 4.5 to a class-based setting. We study the work in [14]. Additional treatment over pure references is necessary for encapsulation, which we solve by making all fields accessible through accessor methods. This does not change the computation. Single-level requirements can be ensured by complete copies of all classes and setting all annotations at the requested level. Projection, constant functions, and composition can be handled as in VSI. Furthermore, we need a matching construct to match objects to their respective classes. This can be realized with `instanceof` and dynamic casts provided by the language. Note that these constructs have the same security level as their inputs, so that they are typable as required.

Chapter 5: EXTENSIONS

This chapter contains content based on [40].

5.1 Nondeterminism

We study the case of possibilistic noninterference of the embedded language, embedded into a deterministic host language. We adapt our simulation technique by restricting the nondeterminism such that a deterministic simulation can be constructed that is in a sense functionally equivalent.

Definition 72 (Deterministic Simulation)**.** *A deterministic language $\mathcal{L}_d$ simulates a nondeterministic language $\mathcal{L}_n$ iff for all program $p_n \in \mathcal{L}_n$ there exists a program $p_d \in \mathcal{L}_d$ such that for all input states μ and values v_n, $(p_n, \mu) \rightsquigarrow_n v_n$ if and only if there exists an i such that $(p_d, i, \mu) \rightsquigarrow_d v_n$.*

Given a deterministic simulation, we can easily lift noninterference. If a program p_n is possibilistic noninterferent in $\mathcal{L}_n$ with respect to signature $\tau_1 \times \cdots \times \tau_n \to \tau$, then p_n is noninterferent in $\mathcal{L}_d$ with respect to signature $\tau_1 \times \cdots \times \tau_n \times \texttt{int}^L \to \tau$. At this point, security completeness as defined before applies and we can embed a host-level simulation of p_n with the deterministic semantics and add a determinism variable corresponding to i, which will capture all and only those behaviors of p_n of the nondeterministic semantics. We now show a broadly applicable procedure for constructing deterministic simulations of programs in non-deterministic languages.

5.1.1 Determinization

For the construction of a deterministic simulation we use the following notations. Inference rules for semantics are of the shape $P \Rightarrow C_1 \to C_2$, where P is a set of constraints, and C_1 and C_2 are configurations (or configuration templates). All may contain free metavariables.

A rule $R \equiv P \Rightarrow C_1 \to C_2$ applies to a pair of concrete configurations c_1 and c_2 if there is a substitution σ of all free variables of P, C_1 and C_2 such that $\sigma C_1 = c_1$ and $\sigma C_2 = c_2$, and for all constraint templates $p \in P$, σp is true (we use σP to abbreviate).

A semantics $\mathcal{S}$ is given as a finite set of inference rules. We require that for any concrete configuration and rule $P \Rightarrow C_1 \rightarrow C_2$, there is a finite number of substitutions σ such that $c_2 = \sigma C_2$ and the rule applies to c_1 and c_2. We assume a predicate $\mathcal{F}$ that categorizes a configuration as final. We assume that if $\mathcal{F}(c_1)$, then there is no c_2 such that any rule applies. We define $\rightarrow^*$ as the transitive closure of the relation implied by the inference rules. We define $\rightsquigarrow$ as the subset of $\rightarrow^*$ where the range is final.

There are two sources of nondeterminism. The first is applicability of multiple inference rules. Given a concrete configuration c_1, multiple rule constraints P and associated configuration templates C may apply. In the deterministic simulation, this situation has to be resolved such that exactly one of the rules applies. The second source is with respect to a single rule. As rules may include meta-variables, which make them templates, and those meta-variables are implicitly universally quantified, different valuations for those meta-variables are possible. After choosing a rule in the first step, a deterministic language has to resolve the mapping of those metavariables. We tackle both issues by transforming the rules. First, the constraints of original rules are extended to decide whether a rule is enabled, that is, chosen in the first step. Then a function is used in the constraints to map the meta-variables to valuations. One may visualize this as implicitly transforming the original rule into a finite *set* of new rules, one for each valuation of the meta-variables. Both new constraints take advantage of the explicit randomness source to define whether a rule is chosen, and which meta-variable mappings should be used. We will use meta-variables i to denote this source.

We adapt an overline convention to separate original and simulation. Let $\overline{C} ::= (C, i)$ and define $\overline{\mathcal{F}}(\overline{C}) = \mathcal{F}(C)$. We translate each rule $R \equiv P \Rightarrow C_1 \rightarrow C_2$ of $\mathcal{S}$ to a rule $\overline{R} \equiv \overline{P} \Rightarrow \overline{\overline{C}_1 \rightarrow \overline{C}_2}$ in the following way: (1) Give the rule a unique number n. (2) Extend the templates of C_1 and C_2 to be pairs with metavariables i and i' for the randomness source. (3) Add the constraint $\text{rule-selected}(R, i, i_r)$, where i_r fresh. (4) For $V = fv(P) \cup fv(C_2) \setminus fv(C_1)$, add a constraint $(v_1, v_2, \ldots, i_v) = \text{select-v}_V(P, C_1, i_r)$, where i_r is fresh. Now let the current i_c be i_v. (5) For all original constraint templates $p \in P$: If p contains $C \rightarrow C'$, then add i_c to C, and add i' a fresh i to

C'. Now let i_c be i'. All other constraints add without change. This threads the randomness source through the inference rule. (6) Add constraint $i_c = i'$.

As an example, take a language with a choice assignment, that is, including reduction rules like

$$\frac{}{\mu, x := e_1 | e_2 \ \rightarrow \ \mu[x \mapsto \mu(e_1)]} \ R_1 \quad \frac{}{\mu, x := e_1 | e_2 \ \rightarrow \ \mu[x \mapsto \mu(e_2)]} \ R_2$$

In our notation, those rules can be written as $\emptyset \Rightarrow (\mu, x := e_1 | e_2) \rightarrow (\mu[x \mapsto \mu(e_1)], \mathrm{skip})$ and $\emptyset \Rightarrow (\mu, x := e_1 | e_2) \rightarrow (\mu[x \mapsto \mu(e_2)], \mathrm{skip})$. For simplicity assume we number the first rule with 1, and the second rule with 2. Then the transformation leads to the following new rules:

$$\begin{array}{l} \mathrm{rule\text{-}selected}(R_1, i, i_r) \\ \wedge\, i_r = i' \end{array} \Rightarrow ((\mu, x := e_1 | e_2), i) \rightarrow ((\mu[x \mapsto \mu(e_1)], \mathrm{skip}), i')$$

$$\begin{array}{l} \mathrm{rule\text{-}selected}(R_2, i, i_r) \\ \wedge\, i_r = i' \end{array} \Rightarrow ((\mu, x := e_1 | e_2), i) \rightarrow ((\mu[x \mapsto \mu(e_2)], \mathrm{skip}), i')$$

where we elided $\mathrm{select\text{-}v}$ as there are no free variables to be bound.

To define the predicates $\mathrm{rule\text{-}selected}$ and $\mathrm{select\text{-}v}$ we need some auxilliary definitions. Let $\mathcal{R}$ be the set of all rules with an arbitrary order. We use square brackets $[\,]$ to index into the set. Then define $E_Q[C]$ with $Q \subseteq \mathcal{R}$ to be the formula

$$E_Q[C] = (\forall_{R_i \in Q}.\ (C = R_i.C' \wedge R_i.P')) \wedge \left(\forall_{R_j \in \mathcal{R} \setminus Q}.\ \neg(C = R_j.C' \wedge R_j.P')\right)$$

where the prime notation denotes a consistent renaming of meta-variables to fresh ones. Given a concrete configuration c such that $c = \sigma C$ for some substitution σ, $\sigma E_Q[C]$ is true if and only if all rules in Q are applicable to c, and all remaining rules are not. We can use this formula with a

free meta-variable to count the applicable rules:

$$\text{nrules}(C, n) = \left(n = 1 \wedge \bigvee_{Q \subseteq \mathcal{R} \wedge |Q|=1} E_Q[C] \right) \vee \left(n = 2 \wedge \bigvee_{Q \subseteq \mathcal{R} \wedge |Q|=2} E_Q[C] \right) \ldots$$

We will write $n = \text{nrules}(C)$ to denote the binding of n in $\text{nrules}(C, n)$. A similar formula allows to derive the (zero-based) position of R in the set Q that applies. We denote this formula by $\text{irule}(R, ind)$ or the functional binding $ind = \text{irule}(R)$. Now we can define rule-selected as

$$\text{rule-selected}(R, i, i_r) \equiv n = \text{nrules}(R.C) \wedge ind = \text{irule}(R) \wedge ind = \text{next}(i, n) \wedge i_r = \text{rest}(i, n)$$

Here we use next to extract randomness from our randomness source i. We require that $\text{next}(i, n) \in \{0, \ldots, n-1\}$ and onto, that is, forall $j \in \{0, \ldots, n-1\}$ there exists i such that $\text{next}(i, n) = j$. Only then is it guaranteed that all new rules can be selected. Similarly, rest "advances" the randomness source, and again we require that forall j there exists i such that $\text{rest}(i, n) = j$. An example for a randomness source and associated functions satisfying those conditions are natural numbers with $\text{next}(i_i, n_i) = (i_i \% n_i)$ and $\text{rest}(i_i, n_i) = i_i / n_i$, but we leave the exact nature abstract.

For select-v_V, we know that there are only a finite number of instances to consider (by the requirement on substitutions). Let this collection be $\mathcal{V}_C$, and have an arbitrary order. Then define

$$\text{select-v}(C, i) = (\mathcal{V}_C[\text{next}(i, |\mathcal{V}_C|)], \text{rest}(i, |\mathcal{V}_C|))$$

Lemma 73 (Determinism)**.** *For any configuration $\overline{c}$, there is at most one rule that applies, and if it applies, there is only one substitution $\overline{\sigma}$ for it.*

The first part derives from the unique numbering, while the second is by induction on the derivation.

Lemma 74. *Assume a pair of configurations c_a and c_b. Assume that rule $R = P \Rightarrow C_a \rightarrow C_b$ is applicable to c_a and c_b, that is, there is a substitution σ such that $\sigma C_a = c_a$, $\sigma C_b = c_b$ and σP.*

Then

1. *There is $Q \subseteq \mathcal{R}$ with $R \in Q$ such that for an extension σ' of σ, $\sigma' E_Q[C_a]$.*

2. *For each i', there exists i such that* $\text{rule-selected}(R, i, i')$.

Proof. For the first part, we let $Q = \{R | R \in \mathcal{R} \wedge \exists \sigma_R.\ \sigma_R R.C_1 = c_a \wedge \sigma_R R.P\}$. Obviously, $R \in Q$ as σ is evidence. Let σ_R denote the substitution guaranteed to exist by Q for $R \in Q$. Let σ'_R denote the substitution that results under the renaming of variables as in $E_Q[C_a]$. Then the domains of all σ' and σ are pairwise disjunct, and for all R we have $\sigma'_R R.C'_1 = c_a \wedge \sigma'_R R.P'$. Then the join σ' of all substitutions σ'_R with σ satisfies the requirements.

The second part follows by inspection of rule-selected. As R applies, $\text{nrules}(R.C) > 0$ and $\text{irule}(R)$ exists. Then by onto-requirements of rest, there exists an i for all i'. □

Lemma 75. *Given a pair of configurations c_a and c_b, and also that rule $R = P \Rightarrow C_a \to C_b$ is applicable to c_a and c_b, that is, there is a substitution σ such that $\sigma C_a = c_a$, $\sigma C_b = c_b$ and σP, let $\overline{R}$ be the rule in $\overline{\mathcal{S}}$ corresponding to R. Then for each i', there exists i such that* $\text{select-v}(C_a, i) = (w_1, \ldots, i')$ *where $\sigma v_i = w_i$, where v_i the free variables of P and C_b.*

Proof is by properties of next and rest.

Lemma 76. *Given a pair of configurations c_a and c_b of $\mathcal{S}$, and given that $c_a \to c_b$ by rule $R = P \Rightarrow C_a \to C_b$, for any $i' \in \mathbb{N}$ there is a $i \in \mathbb{N}$ such that $\overline{(c_a, i) \to (c_b, i')}$ in $\overline{\mathcal{S}}$ by the rule $\overline{R} = \overline{P} \Rightarrow \overline{(C_a, i_a) \to (C_b, i_b)}$ corresponding to R.*

Proof. Proof follows from determinism and an induction in the derivation of $c_a \to c_b$, using properties of next and rest.

If R applies, then there exists a substitution σ such that $\sigma C_a = c_a$, $\sigma C_b = c_b$ and σP. Proof is by induction on the derivation of application of R.

First, consider the case that there is no constraint p in P that describes a nested reduction, that is, is of the form $C_1 \to C_2$. We will show by contradiction that there exists a $\overline{\sigma}$ such that $\overline{R}$ applies to $\overline{(c_a, i) \to (c_b, i')}$ by $\overline{\sigma}$. Assume there is no $\overline{\sigma}$ such that $\overline{\sigma} i_a = i$ for some i and $\overline{\sigma} i_b = i'$

and $\overline{\sigma}\overline{P}$. Thus, either there is no i such that $\overline{\sigma}i_a = i$ and $\overline{\sigma}\text{rule-selected}(R, i_a, i_r)$, or some other constraint cannot be satisfied at the same time. By Lemma 74 and the existence of σ it follows that there exists an i for any i_r such that $\text{rule-selected}(R, i, i_r)$. Thus, another constraint must be unsatisfiable.

Now, either there is no i_r such that $\overline{\sigma}\left[(v_1, \ldots, i_v) = \text{select-v}(C_a, i_r)\right]$, or some other constraint cannot be satisfied. By Lemma 75 and the existence of σ it follows that there exists an i_r for any i_v such that $\text{select-v}(c_a, i_r) = (\sigma v_1, \ldots, i_v)$.

Note that all unmodified constraints in $\overline{P}$ can be satisfied by σ, and $i_v = i_b$ can be satisfied by setting $\overline{\sigma}i_v = i'$. Thus, there is an extension of σ that satisfies $\overline{R}$ and assigns the value i' to i_b.

Now consider the case that there is some constraint p that contains $C_1 \to C_2$. Up to such a constraint, reasoning carries over from the previous case. Then by inductive hypothesis, there is an i_i for any i_o such that $\overline{(\sigma C_1, i_i) \to (\sigma C_2, i_o)}$. This concludes the proof. □

Lemma 77. *Given a pair of configurations c_a and c_b of $\mathcal{S}$ and integers i_a and i_b such that $\overline{(c_a, i_a) \to (c_b, i_b)}$ in $\overline{\mathcal{S}}$, then $c_a \to c_b$ in $\mathcal{S}$.*

Proof. Proof follows from determinism and an induction in the derivation of $\overline{(c_a, i_a) \to (c_b, i_b)}$, using properties of next and rest.

Since $\overline{(c_a, i_a) \to (c_b, i_b)}$ in $\overline{\mathcal{S}}$, there is (by Lemma 73) a unique rule $\overline{R} = \overline{P} \Rightarrow \overline{(C_a, i) \to (C_b, i')}$ and substitution $\overline{\sigma}$ such that $\overline{\sigma}(C_a, i) = (c_a, i_a)$, $\overline{\sigma}(C_b, i') = (c_b, i_b)$ and $\overline{\sigma}\overline{P}$. Let $\sigma x = \overline{\sigma}x$ for all non-i variables. Now proof by induction on the derivation of $\overline{(c_a, i_a) \to (c_b, i_b)}$.

First, consider the case where there is no constraint $\overline{p}$ in $\overline{P}$ that contains $\overline{(C_1, i_1) \to (C_2, i_2)}$. Since $\overline{\sigma}\overline{P}$, we have that $\overline{\sigma}\text{rule-selected}(R, i_a, i_r)$ and $\overline{\sigma}\left[(v_1, \ldots, i_v) = \text{select-v}(C_a, i_r)\right]$. By definition, that means there is a substitution σ such that σP (rule-selected) and $\sigma v_i = \overline{\sigma}v_i$ (select-v). But then $\sigma C_a = c_a$ and $\sigma C_b = c_b$. Thus σ applies to R, the rule $\overline{R}$ was derived from. So $c_a \to c_b$ in $\mathcal{S}$.

Now consider the case that there is a constraint $\overline{p}$ in $\overline{P}$ that contains $\overline{(C_1, i_1) \to (C_2, i_2)}$. We have the same deduction of σ as before. Now also $\overline{\sigma}\overline{(C_1, i_1) \to (C_2, i_2)}$. By inductive hypothesis,

this means that $\overline{\sigma}C_1 \to \overline{\sigma}C_2$ in $\mathcal{S}$. Since σ agrees on all variables of r with $\overline{\sigma}$, $\sigma C_1 = \overline{\sigma}C_1$ and $\sigma C_2 = \overline{\sigma}C_2$. Thus, the constraint p corresponding to $\overline{p}$ is satisfied. It follows that r is applicable, and thus $c_a \to c_b$ in $\mathcal{S}$. □

Theorem 78 ($\overline{\mathcal{S}}$ simulates $\mathcal{S}$.)**.** $c_1 \rightsquigarrow c_2 \iff \exists i.\overline{(c_1, i) \rightsquigarrow (c_2, 0)}$

Proof. Follows by inductions on the length of $\rightsquigarrow$ and $\overline{\rightsquigarrow}$, respectively.

1. $\Rightarrow$: By induction on the length of the derivation of $c_1 \rightsquigarrow c_2$. Base case is $c_1 \to c_2$. Then by Lemma 76, there is an i for any i' such that $\overline{(c_1, i) \to (c_2, i')}$. Choosing $i' = 0$ concludes the base case.

   For the inductive step, assume $c_1 \to c'$ and $c' \rightsquigarrow c_2$. Then by inductive hypothesis, there is an i' such that $\overline{(c', i') \rightsquigarrow (c_2, 0)}$. Furthermore, by Lemma 76, there is an i for this i' such that $\overline{(c_1, i) \to (c', i')}$. Thus, $\overline{(c_1, i) \rightsquigarrow (c_2, 0)}$.

2. $\Leftarrow$: Analogous to $\Rightarrow$. Substitute Lemma 76 with Lemma 77.

□

5.2 Declassification

In practice noninterference is too strong a property to enforce. A canonical example is a login process, which compares a given string to a stored password and allows access if they are identical. However, this constitutes a leak from the viewpoint of noninterference.

Declassification is necessary for intentional information release, relabeling data so that it becomes accessible. This is required when a system needs to leak information to function, the canonical example being a login process. The main questions are under what circumstances a declassification should be allowed and what security guarantee this entails. For a general overview and classification we refer to [80]. There is no generally agreed-upon best definition. We will demonstrate how two examples can be integrated into our framework.

5.2.1 Delimited Release

In Delimited Release [79], declassification expressions define escape hatches. A program is secure iff for any observer level, the program produces indistinguishable outputs given any pair of indistinguishable inputs for which all expressions declared as hatches declassifying to the observer's level evaluate to the same value. Sabelfeld and Myers [79] show how a type-and-effect system can be used to enforce delimited release. The effects here are variables used in declassifaction (D), and variables modified (U). A program is guaranteed to be secure, if $D \cap U = \emptyset$.

Given two languages with such type-and-effect systems, we can extend the typing of an embedding to also encompass effects. An example is

$$\frac{\Gamma \vdash e : \ell, D_e \quad \Delta, [\dot{x} : \ell] \vdash \dot{c} : \ell_v/\ell_s, D_{\dot{c}}, U_{\dot{c}} \quad \ell_v \sqsubseteq \Gamma(x) \quad D_{\dot{x}} = \begin{cases} Vars(e) & \dot{x} \in D_{\dot{c}} \\ \emptyset & else \end{cases}}{\Gamma, \Delta \vdash x := \text{eval}\, e \,\text{in}\, \dot{c} : \Gamma(x_i) \sqcap \ell_S, D_e \cup D_{\dot{c}} \cup D_{\dot{x}}, U_{\dot{c}} \cup \{x\}}$$

In words, the declassification effect of an eval encompasses the declassification in the parameter expression and the embedded declassification effect. Also, if the embedded effect contains the embedded parameter, then all variables in the parameter expression are involved in declassification. Similarly, the modified variables are made up of the variables modified in the embedded fragment, and the result variable at the host level.

5.2.2 Robust Declassification

Robust declassification [92] informally defines a system potentially including declassification to be *robust* if an attacker cannot deduce additional information when attacks are applied. This is an example of the *who* dimension of declassification, as the attacker does not have influence over what gets declassified.

In [68], a type system is proposed that enforces a variant robust declassification. A secure While language with both confidentiality and integrity is extended with a declassification expres-

sion and holes. Attacks are defined to be noninterfering programs that cannot influence high-integrity variables. A program is complete when holes are filled with attacks. The type system then enforces that for a typable program with holes, for any two derived complete programs and all inputs, if an attacker cannot distinguish runs of the first program, then she cannot distinguish runs of the second.

To show how two languages enforcing robust declassification can be composed, we first abstract the guarantees of the language of [68], transposing them to requirements on traces in a state transition system. We call this property step-wise robustness. We then show that a system satisfying the requirements is robust with respect to the definition in [92]. Finally, given two systems guaranteeing step-wise robustness, we show that the disciplined composition given by typed embedding continues to be step-wise robust.

Let $\mathcal{S} = (\Sigma, \mapsto, L, D, A, E, \Gamma)$ be an (annotated) state transition system, where states are tuples of the form $\sigma = (\sigma_{HH}, \sigma_{HL}, \sigma_{LH}, \sigma_{LL}, l)$ with l denoting an abstract location from L. $\mapsto$ is a binary relation over states, D is a predicate describing which locations may declassify data, that is, over which $\mapsto$ is not required to be noninterference-preserving. A is a predicate describing which locations denote attacks, E maps locations to exit points of single-exit regions of which the location is a part, and Γ is a mapping of locations to security levels, that is, Γ is an abstract representation of knowledge encoded in the code part of a program's configuration, for example, the knowledge of branch outcomes given the current program location, while E encodes knowledge about the structure of a program and is used to allow downgrading of the context label given by Γ whenever it can be guaranteed that code flows reconverge. Traces τ are (possibly empty) sequences of states such that $\tau(0) \mapsto \tau(1) \mapsto \ldots$. We define $\tau.e$ as the last state of τ if τ is finite and $\perp$ otherwise. Traces (and states) are concatenated with $\oplus$.

A system $\mathcal{S}$ is valid iff properties V1 through V5 hold for Γ and E, namely

V1. $\forall \sigma \mapsto \sigma'.\ \exists k \geq 0.\ E(\sigma.l) = E^k(\sigma'.l) \wedge \forall 0 \leq i < k.\ \Gamma(\sigma.l) \sqsubseteq \Gamma(E^i(\sigma'.l))$

V2. $\forall l.\Gamma(E(l)) \sqsubseteq \Gamma(l)$

V3. $\forall \sigma_1 \mapsto \sigma_1', \sigma_2 \mapsto \sigma_2'.\sigma_1 \approx_{I/C} \sigma_2 \wedge \sigma_1.l = \sigma_2.l \wedge \sigma_1'.l \neq \sigma_2'.l \implies \Gamma(\sigma_1'.l) \in H_{I/C} \wedge \Gamma(\sigma_2'.l) \in H_{I/C}$.

V4. $\forall l.D(l) \implies \Gamma(l) \in H_I \wedge \Gamma(l) \in L_C$

V5. $\forall l.A(l) \implies \Gamma(l) \in L_C$

where $E^0(l) = l$ and $E^{k+1}(l) = E^k(E(l))$, and subscripts $_{X/Y}$ are an abbreviation for separate identical rules with subscripts $_X$ and $_Y$. Here $L_{I/C}$ are the low-integrity/low-confidentiality levels, and $H_{I/C}$ correspondingly. We define confidentiality-indistinguishability satisfying $\sigma \approx_C \sigma' \implies \sigma_{LH} = \sigma'_{LH} \wedge \sigma_{LL} = \sigma'_{LL}$, and analogously integrity-indistinguishability satsifying $\sigma \approx_I \sigma' \implies \sigma_{LH} = \sigma'_{LH} \wedge \sigma_{HH} = \sigma'_{HH}$. The reverse direction is required to hold whenever also $\sigma.l = \sigma'.l$; or $\Gamma(\sigma.l) \in H$, $\Gamma(\sigma'.l) \in H$ and for the smallest k and k' such that $E^k(\sigma.l) \in L$ and $E^{k'}(\sigma'.l) \in L$ the elements $E^k(\sigma.l)$ and $E^{k'}(\sigma'.l)$ are the same; or $\Gamma(\sigma.l) \in H$, $\Gamma(\sigma'.l) \in L$ and $\sigma'.l = E^k(\sigma.l)$ is minimal in k such that $E^k(\sigma.l) \in L$; or the symmetric case of the last one. For termination-sensitive noninterference, $\perp$ is only equivalent to itself, while for termination-insensitive noninterference $\perp$ is low-equivalent with every state.

We model attacks as part of $\mathcal{S}$, that is, $\mapsto_A \subseteq \mapsto$, where a single transition captures the whole attack. The requirements on $\mapsto_A$ then are that attacks are noninterferent computations, thus indistinguishable inputs to an attack lead to indistinguishable outputs; attacks do not change the high-integrity parts of a state; and start in a low-confidentiality locations, that is $\forall l.A(l) \implies \Gamma(l) \in L_C$. These are the standard definitions from [68]. Further, attacks are only allowed at specific locations denoted by A, and for all attacks $\sigma_1 \mapsto_A \sigma_2$ and $\sigma_1' \mapsto \sigma_2'$ where $\sigma_1.l = \sigma_1'.l$ we have $\sigma_2.l = \sigma_2'.l$. Last, to capture that attacks stand for (terminating) computations, we require they satisfy a closure property: all states with locations $A(l)$ have an attack transition, and only such relations.

A system $\mathcal{S}$ is step-wise robust with respect to attacks $\mapsto_A$ iff

S1. $\mathcal{S}$ is valid and deterministic in non-attack transitions/locations

S2. $\forall \sigma_1, \sigma_1', \sigma_2.\ \sigma_1 \approx_I \sigma_2 \wedge \sigma_1 \mapsto \sigma_1' \implies \exists \tau_2'.\ \tau_2'(0) = \sigma_2 \wedge \forall i < len(\tau_2').\ \tau_2'(i) \approx_I \sigma_2 \wedge \tau_2'.e \approx_I \sigma_1'$

Given two integrity-indistinguishable states, a step of one can be matched by zero or more steps of the other so that the result remains integrity-indistinguishable.

S3. $\forall \sigma_1, \sigma_1', \sigma_2.\ \sigma_1 \approx_C \sigma_2 \wedge \sigma_1 \mapsto \sigma_1' \wedge \neg D(\sigma_1) \implies \exists \tau_2'.\ \tau_2'(0) = \sigma_2 \wedge \forall i < len(\tau_2').\ \tau_2'(i) \approx_C \sigma_2 \wedge \tau_2'.e \approx_C \sigma_1'$

Given two confidentiality-indistinguishable states and the first is not declassifying, a step of the first can be matched by zero or more steps of the second so that the result remains confidentiality-indistinguishable.

S4. $\forall \sigma_1, \sigma_2.\ \sigma_1 \approx_{I/C} \sigma_2 \wedge D(\sigma_1) \implies \exists \tau_2'.\ \tau_2'(0) = \sigma_2 \wedge \forall i < len(\tau_2').\ \tau_2'(i) \approx_{I/C} \sigma_2 \wedge D(\tau_2'.e)$

If a state is marked as declassifying, then a declassification must be reachable from all integrity-equivalent/confidentiality-equivalent states.

S5. $\forall \sigma, \sigma'.\ \sigma \mapsto \sigma' \wedge D(\sigma) \implies \sigma_{LL} = \sigma'_{LL}$

Declassification does not influence the LL part of a state.

The language in [68] is a structured and well-behaved While language with small-step semantics defined over heaps M and statements c. Given a typed program $\Gamma, pc \vdash c[\bullet]$, we can translate it to a corresponding transition system $\mathcal{S}$. This system satisfies that for any attack a and intermediate state $\langle M, c' \rangle$ such that $\langle M_0, c[a] \rangle \rightarrow^* \langle M, c' \rangle$, $\langle M, c' \rangle \equiv \sigma \mapsto \sigma' \equiv \langle M', c'' \rangle$ can be matched to a (multi-)step $\langle M, c' \rangle \rightarrow^* \langle M', c'' \rangle$. With this we can derive the following theorem.

Theorem 79 (Language-robust implies step-wise robust)**.** *If $\Gamma, pc \vdash c[\bullet]$, then there exists $\mathcal{S}$ that is (a) able to simulate all runs of c under any attack a and (b) $\mathcal{S}$ is step-wise robust.*

We require two auxiliary lemmas.

Lemma 80 (Single Exit)**.** *For all states σ in a valid system $\mathcal{S}$ with $\Gamma(E(\sigma.l)) \sqsubset \Gamma(\sigma.l)$, it holds for all traces $\tau \in \mathcal{T}(\mathcal{S}, \sigma)$ generated from σ that one of the following is satisfied*

- $\forall i \leq len(\tau).\ pc_E \sqsubset \Gamma(\tau(i).l)$
- $\tau = \tau_1 \mapsto \sigma_E \mapsto \tau_2$, $\forall i \leq len(\tau_1).\ pc_E \sqsubset \Gamma(\tau_1(i).l)$, *and* $\sigma_E.l = E(\sigma.l)$

where $pc_E = \Gamma(E(\sigma.l))$.

Proof. Let τ be a trace generated by σ. Let $l = \sigma.l$ and $l_E = E(\sigma.l)$. Let $\ell = \Gamma(l)$. Note that $pc_E \sqsubset \ell$. Let k be the first index in τ such that $\ell \not\sqsubseteq \Gamma(\tau(k).l)$.

If such a k exists, it must be $k > 0$. Now for the prefix of τ made up of the first $k-1$ elements, we have that $\forall i < k.\ell \sqsubseteq \Gamma(\tau(i).l)$. We apply V1 iteratively to the prefix. Let l_1 be the location before a step, and l_2 be the location after the step. Then before a step, we have $l_1 \neq l_E$, $\ell \sqsubseteq \Gamma(l_1)$, and $\exists j.\ E^j(l_1) = l_E \wedge \forall 0 \leq i < j.\ell \sqsubseteq \Gamma(E^i(l_1))$. By V1, $\exists m \geq 0.E(l_1) = E^m(l_2)$ and $\forall 0 \leq i < m.\ \Gamma(l_1) \sqsubseteq \Gamma(E^i(l_2))$. If $m = 0$, then we have $l_2 = E(l_1)$. Now also $\ell \sqsubseteq \Gamma(l_2)$. Thus $l_2 \neq l_E$, so $\exists j.E^j(l_2) = l_E$ and $\forall 0 \leq i < j.\ell \sqsubseteq \Gamma(E^i(l_2)$. If $m = 1$, then the property is preserved immediately. If $m > 1$, then $\Gamma(E(l_1)) \sqsubseteq \Gamma(l_1)$ (V2) and $\Gamma(l_1) \sqsubseteq \Gamma(E(l_2))$ and $\Gamma(l_1) \sqsubseteq \Gamma(l_2)$ (V1), so by transitivity the property is preserved. We thus have $\tau_1 = \tau(0) \mapsto^* \tau(k-1)$ with $\forall i \leq len(\tau_1) = k-1.\ pc_E \sqsubset \ell \sqsubseteq \Gamma(\tau_1(i).l)$.

Now inspect $\tau(k-1) \mapsto \tau(k)$. Again part 1 of validity applies. It cannot be the case that $m \geq 1$, as then k was not minimal. So $m = 0$. Then $\tau(k).l = E(\tau(k-1).l)$. From the above property of τ_1, we have that $\exists j.\ E^j(\tau(k-1).l) = l_E \wedge \forall 0 \leq i < j.\ell_E \sqsubset \Gamma(E^i(\tau(k-1).l)$. It follows that $j = 1$, as all other cases are contradictory: $j = 0$ implies $\tau(k-1).l = l_E$ and so k not minimal; $j > 1$ implies $\ell_E \sqsubset \Gamma(E(\tau(k-1).l)) = \Gamma(\tau(k).l)$. So $\tau(k).l = l_E$.

The case when k does not exist follows the derivation of τ_1 above. □

Lemma 81 (Translation is valid)**.** *The translation of* $\Gamma, pc \vdash c[\bullet]$ *is valid.*

Proof. Validity is defined as:

V1. $\forall \sigma \mapsto \sigma'.\ \exists k \geq 0.\ E(\sigma.l) = E^k(\sigma'.l) \wedge \forall 0 \leq i < k.\ \Gamma(\sigma.l) \sqsubseteq \Gamma(E^i(\sigma'.l))$

Let $l = \sigma.l$ and $l' = \sigma'.l$. $\mapsto$ is generated by the small-step semantics. Induction over (unrolled) semantic rules. We show select cases.

- $[x := a]^l; [c]^{l'} \rightarrow [c]^{l'}$.

  Then $E(l) = E(l')$ and $\Gamma(l) = \Gamma(l')$ and case $k = 1$ applies.

- $[\text{if } e \text{ then } [c_1]^{l_1} \text{ else } [c_2]^{l_2} \text{ end}]^l; [c_3]^{l_3} \rightarrow [c_1]^{l_1}; [c_3]^{l_3}$, where $e \Downarrow \text{true}$ and $\Gamma(e) = H$.

  Then $\Gamma(l_1) = H$ and $E(l_1) = l_3$ and $E(l) = E(l_3)$. Thus case $k = 2$ applies.

- $[c_1]^{l_1}; [c_3]^{l_3} \rightarrow [c^3]^{l_3}$ (continued from above).

  Then, as $l_3 = E(l_1)$, the case $k = 0$ applies.

V2. $\forall l.\Gamma(E(l)) \sqsubseteq \Gamma(l)$

Let c' be the control structure immediately enclosing l. Then the pc of c' is lower or equal to the pc at l and equal to the one at $E(l)$. Thus, $\Gamma(E(l)) \sqsubseteq \Gamma(l)$.

V3. $\forall \sigma_1 \mapsto \sigma_1', \sigma_2 \mapsto \sigma_2'.\sigma_1 \approx_{I/C} \sigma_2 \wedge \sigma_1.l = \sigma_2.l \wedge \sigma_1'.l \neq \sigma_2'.l \implies \Gamma(\sigma_1'.l) \in H_{I/C} \wedge \Gamma(\sigma_2'.l) \in H_{I/C}$.

The only way that successor locations can vary is if $l = \sigma_1.l = \sigma_2.l$ denotes a control structure. If different confidentiality/integrity-equivalent lead to different branches, the condition must depend on the hi-confidentiality/lo-integrity part of the state. Then, since the program is typed, the pc at $\sigma_1'.l$ and $\sigma_2'.l$ must be hi-confidential/lo-integrity.

V4. $\forall l.D(l) \implies \Gamma(l) \in H_I \wedge \Gamma(l) \in L_C$

This follows immediately from the typing rules.

V5. $\forall l.A(l) \implies \Gamma(l) \in L_C$

This follows immediately from the typing rules.

□

Now we can show Theorem 79.

Proof. We first note that we investigate a termination-sensitive version of [68]. We feel this is valid because, in fact, proofs to the soundness of the type system proposed in [68] are only correct if the type system is termination-sensitive.

For the first part, let l_c be the location assigned to $c[\bullet]$. Take an arbitrary M_0 and a, and generate the (deterministic) execution started by $\langle M_o, c[a]\rangle$.

Proceed by induction on the length of an execution prefix. For all steps $\langle M^1, c^1\rangle \to \langle M^2, c^2\rangle$ that are not part of executing a, the result follows immediately since $\mapsto$ is generated by the small-step semantics. So assume c^1 is part of executing a. Then there is a (possibly empty) prefix of the computation such that the prefix ends just before executing a, that is, $\langle M', a; c'\rangle$. This prefix is matched by σ'. Since a is an attack, it is a noninterfering, high-integrity-preserving, non-declassifying computation. Thus, if a terminates, there exists σ'' such that $\sigma' \mapsto_A \sigma''$ and σ'' is updated with the effect of a on σ'. Complete the execution of a in c^1, that is, $\langle M'', c'\rangle$, as we assumed that a terminates on M'. Then it follows that $\langle M'', c'\rangle \equiv \sigma''$, as c' is the successor of the hole in $c[\bullet]$. On the other hand, if a does not terminate, then there won't be a transition from σ', and also $c[a]$ does not terminate. The stuck state simulates this non-termination.

For the second part, we first note that $\mathcal{S}$ is valid by the previous lemma. Further, all attacks ($\mapsto_A$) that were added are noninterfering, high-integrity preserving computations. This satisfies the first two attack properties. Now take two C-indistinguishable states σ_1 and σ_2 so that $A(\sigma_1.l)$. Then its label $l_1 = \sigma_1.l$ denotes a hole in c, and $\Gamma(l_1)$ is low-confidential. To be C-indistinguishable, $l_2 = \sigma_2.l$ must be either $= l_1$ or $\exists k.\ l_1 \in E^k(l_2)$ such that for all indices $i < k$ $\Gamma(l_1) \sqsubset \Gamma(E^i(l_2))$. In the first case, there must be an attack transition. In the second case, if the computation starting with σ_2 terminates, it must reach l_1, at which point there is an attack.

For non-attack transitions, we check each part of step-wise robustness.

S2. $\forall \sigma_1, \sigma_1', \sigma_2.\ \sigma_1 \approx_I \sigma_2 \wedge \sigma_1 \mapsto \sigma_1' \implies \exists \tau_2'.\ \tau_2'(0) = \sigma_2 \wedge \forall i < len(\tau_2').\ \tau_2'(i) \approx_I \sigma_2 \wedge \tau_2'.e \approx_I \sigma_1'$.

If the input states are indistinguishable, then either $l_1 = \sigma_1.l = \sigma_2.l = l_2$, or $\exists k_1, k_2.\ E^{k_1}(l_1) = E^{k_2}(l_2)$ where all smaller indices are not in L_I. In the first case, both states denote the same program location. As the program is typed, we can apply to Theorem 2 of [68]. The second case has three sub-cases. If $l_2 = E(l_1)$, then $\tau = \emptyset$ is the solution. If $l_1 = E^{k_2}(l_2)$, then we can use Lemma 80 to reduce to the first case, and otherwise we apply Lemma 80 twice to

reduce to the first case.

S3. $\forall \sigma_1, \sigma_1', \sigma_2.\ \sigma_1 \approx_C \sigma_2 \wedge \sigma_1 \mapsto \sigma_1' \wedge \neg D(\sigma_1) \implies \exists \tau_2'.\ \tau_2'(0) = \sigma_2 \wedge \forall i < len(\tau_2').\ \tau_2'(i) \approx_C \sigma_2 \wedge \tau_2'.e \approx_C \sigma_1'$

Analogous to the first case. As $\neg D(\sigma_l)$, the location does not denote a declassification, and we can appeal to Theorem 1 of [68].

S4. $\forall \sigma_1, \sigma_2.\ \sigma_1 \approx_{I/C} \sigma_2 \wedge D(\sigma_1) \implies \exists \tau_2'.\ \tau_2'(0) = \sigma_2 \wedge \forall i < len(\tau_2').\ \tau_2'(i) \approx_{I/C} \sigma_2 \wedge D(\tau_2'.e)$

Declassification is required to be in a low-confidentiality, high-integrity context by typing rules. Any equivalent state is either at the same location, or the declassification is the exit. We conclude with Lemma 80.

S5. $\forall \sigma, \sigma'.\ \sigma \mapsto \sigma' \wedge D(\sigma) \implies \sigma_{LL} = \sigma'_{LL}$

We declassify high-integrity-high-confidentiality data to high-integrity-low-confidentiality data.

□

Next we show that step-wise robustness is a meaningful declassification guarantee by showing that it implies robustness as adapted from [92]. Let $\mapsto_{skip} \subseteq \mapsto_A$ such that for all $\sigma \mapsto_{skip} \sigma'$ we have the four non-location components equivalent. A system restricted to those attack transitions can be considered not under attack. Informally, trace-based robustness states that if two starting states are observationally equivalent in the base system, then they are observationally equivalent when under attack, where observational equivalence is the equivalence of the sets of traces generated starting at the start states modulo indistinguishability. Formally, $\mathcal{O}_\sigma(\mathcal{S}, \approx) = \{\tau / \approx \mid \tau \in \mathcal{T}(\mathcal{S}, \sigma)\}$ and $\mathcal{S}$ is robust with respect to attack A iff for all σ and σ', $\mathcal{O}_\sigma(\mathcal{S}, \approx) = \mathcal{O}_{\sigma'}(\mathcal{S}, \approx) \implies \mathcal{O}_\sigma(\mathcal{S} \cup A, \approx) = \mathcal{O}_{\sigma'}(\mathcal{S} \cup A, \approx)$. The requirements for step-wise robustness allow to derive the following theorem.

Theorem 82 (Step-wise robust implies robust)**.** *If $\mathcal{S} = (\Sigma, \mapsto, D)$ is step-wise robust with respect to $\mapsto_A$, then $(\Sigma, \mapsto \setminus \mapsto_A \cup \mapsto_{skip})$ is robust with respect to A and $\approx_C$.*

We require some auxilliary notation and lemmas for this proof. Let $\overline{\mathcal{S}}$ be $\mathcal{S}$ without $\mapsto_A$, and $\overline{\mathcal{S}} \cup A'$ be the system when adding $\mapsto_{A'}$ to $\overline{\mathcal{S}}$.

Lemma 83. *If σ_1 and σ_2 are observationally equivalent under $\approx_C$, $\overline{\sigma}_1 \approx_I \sigma_1$, and $\overline{\sigma}_2 \approx_I \sigma_2$, then $\overline{\sigma}_1$ and $\overline{\sigma}_2$ are observationally equivalent under $\approx_C$.*

Proof. Observational equivalence can be stated as $\forall \overline{\tau}_1 \in \mathcal{T}(\mathcal{S}, \overline{\sigma}_1).\exists \overline{\tau}_2 \in \mathcal{T}(\mathcal{S}, \overline{\sigma}_2).\ \overline{\tau}_1 / \approx_C = \overline{\tau}_2 / \approx_C$. Induction on the length of $\overline{\tau}_1$. Assume $\overline{\tau}_1 = \overline{\sigma}_1 \mapsto \overline{\tau}'_1$ and let $\overline{\sigma}'_1 = \overline{\tau}'_1(0)$. Case decision on $D(\overline{\sigma}_1)$.

If $\overline{\sigma}_1$ cannot declassify, then by step-wise robustness $\exists \overline{\tau}'_2$ with $\overline{\tau}'_2(0) = \overline{\sigma}_2$ such that $\overline{\tau}'_2$ preserves indistinguishability and $\overline{\tau}'_2.e \approx_C \overline{\sigma}'_1$. By step-wise robustness, $\exists \tau_1, \tau_2$ with $\tau_1(0) = \sigma_1$ and $\tau_2(0) = \sigma_2$, τ_1 and τ_2 preserve C-indistinguishability and $\tau_1.e \approx_I \overline{\sigma}'_1$ and $\tau_2.e \approx_I \overline{\tau}'_2.e$. As σ_1 and σ_2 are observationally indistinguishable, all corresponding reachable states must be, too. Thus $\tau_1.e$ is observationally equivalent to $\tau_2.e$. By induction hypothesis, $\overline{\sigma}'_1$ is observationally equivalent to $\overline{\tau}'_2.e$. Thus for all traces generated by $\overline{\sigma}'_1$ there exists a trace generated by $\overline{\tau}'_2.e$ that is equivalent. Thus, there exists $\overline{\tau}''_2$ starting from $\overline{\tau}'_2.e$ such that $\overline{\tau}'_1 / \approx_C = \overline{\tau}'_2 / \approx_C$. Concatenation concludes.

Now assume $\overline{\sigma}_1$ can declassify. Then by step-wise robustness exists τ_1 such that $\tau_1(0) = \sigma_1$, τ_1 preserves indistinguishability and $\tau_1.e$ can declassify. Further, there exists σ'_1 such that $\tau_1.e \mapsto \sigma'_1$, $\overline{\sigma}'_1 \approx_I \sigma'_1$, $\tau_1.e_{LL} = \sigma'_{1LL}$ and $\overline{\sigma}_{1LL} = \overline{\sigma}'_{1LL}$. As $\overline{\sigma}_1 \approx_C \overline{\sigma}_2$, $\overline{\sigma}_2$ can reach a declassification through a trace $\overline{\tau}_2$ preserving indistinguishability on the way, where we let $\overline{\sigma}'_2 = \overline{\tau}_2.e$. As before, this can be matched by τ_2 with $\tau_2(0) = \sigma_2$, where we let $\sigma'_2 = \tau_2.e$.

Now we have $\overline{\sigma}'_1 \approx_I \sigma'_1 \approx_C \sigma'_2 \approx_I \overline{\sigma}'_2$, and furthermore $\overline{\sigma}'_{1LL} = \overline{\sigma}'_{2LL}$ as result of $\approx_C$ followed by declassification. As $\sigma'_1 \approx_C \sigma'_2$ and $\overline{\sigma}'_{1/2LH} = \sigma'_{1/2LH}$, it follows that $\overline{\sigma}'_1 \approx_C \overline{\sigma}'_2$. We conclude as in the non-declassifying case. □

We can now prove the theorem.

Proof of Theorem 5. Given two states σ_1 and σ_2 that are observationally equivalent under attack A_1, we will show that these states are also observationally equivalent under attack A_2. Let $\tau^1 \in \mathcal{T}(\overline{S} \cup A_2, \sigma_1)$. Divide τ^1 into segments (states connected by non-attack transitions) connected by

attack transitions: $\tau^1 = \tau^1_1 \mapsto_{A_2} \tau^1_2 \mapsto_{A_2} \ldots$. We will show that there exists τ^2_i for each τ^1_i such that $\tau^1_i / \approx_C = \tau^2_i / \approx_C$ and τ^2_1 starts with σ_2, by induction on the number of segments.

In the base case, there is just one segment, i.e., $\tau^1 = \tau^1_1$. As there are no A_2 transitions, we have that $\tau^1 \in \mathcal{T}(\overline{S} \cup A_1, \sigma_1)$. Since σ_1 and σ_2 are observationally equivalent, there exists $\tau^2 \in \mathcal{T}(\overline{S} \cup A_1, \sigma_2)$ such that $\tau^1 / \approx_C = \tau^2 / \approx_C$. We can construct τ' a prefix of τ^2 such that $\tau^1 / \approx_C = \tau' / \approx_C$ and τ' does not contain A_1 transitions: Assume there are such transitions. Then let τ' be the prefix up to and excluding the first A_1 transition. As it is a prefix of τ^2, and τ^2 is equivalent to τ^1, there must be a prefix of τ^1 that is equivalent to τ'. By the closure rules on attacks, either $\tau' = \tau^1$, or all states up to the end of τ^1 are equivalent to the end of τ'. Thus, $\tau' / \approx_C = \tau^1 / \approx_C$. The trace τ' does not contain A_1 transitions. It is thus a trace of $\mathcal{T}(\overline{\mathcal{S}} \cup A_2, \sigma_2)$.

Now assume we have segments $\tau^{1/2}_1, \ldots, \tau^{1/2}_i$ such that $\tau^1_j / \approx_C = \tau^2_j / \approx_C$, $\tau^{1/2}_j \mapsto_{A_2} \tau^{1/2}_{j+1}$, $\tau^1_i.e \approx_I \sigma^1_i \approx_C \sigma^2_i \approx_I \tau^2_i.e$, where σ^1_i is observationally equivalent to σ^2_i in $\overline{\mathcal{S}} \cup A_1$, and $\tau^1_i \mapsto_{A_2} \tau^1_{i+1}$. We first note that, as $\tau^1_i / \approx_C = \tau^2_i / \approx_C$, if there is no attack transition from $\tau^2_i.e$ directly, then there exists an extension that preserves indistinguishability and ends in such a state. Thus, w.l.o.g., $\exists \overline{\sigma}^2_{i+1}.\tau^2_i.e \mapsto_{A_2} \overline{\sigma}^2_{i+1}$. Let $\overline{\sigma}^1_{i+1} = \tau^1_{i+1}(0)$. We have that $\tau^1_i.e \approx_I \overline{\sigma}^1_{i+1} \approx_C \overline{\sigma}^2_{i+1} \approx_I \tau^2_i.e$ as an attack transition connects them. Thus $\sigma^{1/2}_i \approx_I \overline{\sigma}^{1/2}_{i+1}$. By Lemma 83, $\overline{\sigma}^1_{i+1}$ and $\overline{\sigma}^2_{i+1}$ are observationally equivalent in $\overline{\mathcal{S}} \cup A_1$, and $\tau^1_{i+1} \in \mathcal{T}(\overline{\mathcal{S}} \cup A_1, \overline{\sigma}^1_{i+1})$. Analogous to the base case, there is a $\mapsto_{A_1}$-free τ^2_{i+1} with $\tau^2_{i+1}(0) = \overline{\sigma}^2_{i+1}$ such that $\tau^1_{i+1} / \approx_C = \tau^2_{i+1} / \approx_C$. Thus $\tau^2_{i+1} \in \mathcal{T}(\overline{\mathcal{S}} \cup A_2, \overline{\sigma}^2_{i+1})$. The concatenation of the segments concludes the proof. □

Finally, we apply step-wise robustness to composition. Given systems $\mathcal{S}_1$ and $\mathcal{S}_2$, we create a composed system by defining $\mathcal{S} = (\Sigma = \Sigma_1 \times \Sigma_2 \times L, \mapsto, L = L_1 \cup L_2, D = D_1 \cup D_2, A = A_1 \cup A_2, E = E_1 \cup E_2, \Gamma = \Gamma_1 \cup \Gamma_2))$ and $(\sigma_1, \sigma_2, s) \mapsto (\sigma'_1, \sigma'_2, s')$ only if (1) $s = l_1$, $s' = l'_1$ and $\sigma_1 \mapsto_1 \sigma'_1$ and $\sigma_2 = \sigma'_2$ where $\sigma_1.l = l_1$ and $\sigma'_1.l = l'_1$, (2) $s = l_2$, $s' = l'_2$ and $\sigma_1 = \sigma'_1$ and $\sigma_2 \mapsto_2 \sigma'_2$ where $\sigma_2.l = l_2$ and $\sigma'_2.l = l'_2$, (3) $s = l_1$, $s' = l_2$, $\sigma_1 = \sigma'_1$, $\sigma_1.l = l_1$ corresponds to $x_1, \ldots, x_n := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$ and σ'_2 is an update of σ_2 corresponding to $\alpha(\sigma_1(x'))$ and $\sigma'_2.l = l_2$ corresponds to $\dot{c}$, or (4) $s = l_2$, $s' = l_1$, $\sigma_2 = \sigma'_2$, there is no successor to σ_2 in $\mathcal{S}_2$, $\sigma_2.l = l_2$, σ_1 corresponds to $x_1, \ldots, x_n := \mathrm{eval}\, x' \,\mathrm{in}\, \dot{c}$, σ'_1 corresponds to the successor of σ_1 and is an update of

σ_1 corresponding to $\gamma(\sigma_2(\dot{c}))$ with $\sigma_1'.l = l_1$. These four options correspond to either pure-host or pure-embedded computation ((1) and (2)), or invoking and returning from an eval.

For security, we straightforwardly lift $\approx_C$ and $\approx_I$ to the composition. If $\mathcal{S}_1$ and $\mathcal{S}_2$, the composite program is typed, and the embedded program is terminating if the eval is typed under high pc, then we can show the following theorem.

Theorem 84 (Composition is step-wise robust)**.** *Given step-wise robust systems $\mathcal{S}_1$ and $\mathcal{S}_2$, and a composition $\mathcal{S}$ constructed as above, then the composition $\mathcal{S}$ is step-wise robust.*

Proof. Step-wise robustness is defined as the set of five properties S1 through S5. We have to show validity of the composed system (S1), and then the other four (core) properties.

We start by noting that `eval` itself is noninterfering, non-declassifying, and by typing, pcs increase. We enforce a single-entry single-exit regime for `eval`. It is thus easy to show that the validity requirements V1 through V4 hold for systems describing simple `eval` executions. Then composed-system validity follows trivially by the single-entry single-exit nature, deterministic and noninterfering behavior of `eval`.

For the other step-wise robustness properties, we note that S4 does not apply to `eval`, and thus immediately carries over from the constituent systems. For all other properties, whenever $\sigma_1.l$ and $\sigma_2.l$ belong to one language mode, the properties either carry over directly (embedded mode) or are follow immediately from the constituent and the validity of simple `eval` executions. We will detail the arguments for $\sigma_1.l$ and $\sigma_2.l$ not denoting the same language mode on the example of S2.

We have to show that $\forall \sigma_1, \sigma_1', \sigma_2.\ \sigma_1 \approx_I \sigma_2 \wedge \sigma_1 \mapsto \sigma_1' \implies \exists \tau_2'.\ \tau_2'(0) = \sigma_2 \wedge \forall i < len(\tau_2').\ \tau_2'(i) \approx_I \sigma_2 \wedge \tau_2'.e \approx_I \sigma_1'$. Note that as σ_1 and σ_2 denote different language states, $l_1 = \sigma_1.l \neq \sigma_2.l = l_2$. Then $\exists k_1, k_2. E^{k_1}(l_1) = E^{k_2}(l_2)$. If $l_1 = E^{k_2}(l_2)$, then by validity l_2 can complete the `eval` and reach l_1, from which a step is possible. If otherwise $E^{k_1>0}(l_1) = E^{k_2}(l_2)$, then the step of l_1 can be matched by a zero-step of l_2. The properties S3 through S5 are analogous. □

Chapter 6: SECURITY-TYPED EMBEDDED LANGUAGES

The content of this chapter is based on [39] and [40].

This chapter describes the work on two type systems for languages that are likely to be embedded.

The first is our contribution towards checking a dynamic object-based calculus. We note that a serious complication with type-checking modern scripting languages is the high dynamicity of objects in those languages. In fact, many common idioms are build around the ability to dynamically change the structure of objects at runtime. If a type system is unable to precisely mirror and abstract those changes, it will not be feasible in practice, as it will fail to type practical programs. As security type systems are simply extended type system, this applies to the security domain, as well.

As a contribution to the effort of improving the applicability of a type system-based approach to security, we investigate the security implications of so-called method-not-found errors, which may occur when a non-existing member of an object is invoked. This can be used as a leaking channel.

```
o := empty_object
if (h>0) {
  o.m := λ x. print x
}
o.m(0)
```

In the case of $h = 0$, the program will terminate with a method-not-found error and thus the attacker gains one bit of knowledge.

There are two ways to handle this: the type system could reject the program, if it cannot show that the call always succeeds or always fails, or it can track the information flow of this termination channel. In Section 6.1 we pursue the latter approach.

In Section 6.2 we investigate an expressive SQL fragment. Our fragment supports arbitrary

tables, standard projection, selection, and table joins. We do not address nested queries, because the literature explains how nested queries can be simulated (e.g., [57]), and handling such queries is tedious but technically insignificant.

As SQL was our canonical motivating example, we show that it is possible to enforce a security type system regime while supporting most of the features of the data manipulation language part. We formalize the fragment with straightforward syntax and semantics and formally prove that our security type system is sound.

6.1 OO

6.1.1 Introduction

There are two main approaches to enforce noninterference: one is static and the other is dynamic. Both control the ways information is allowed to flow in the program and are called static and dynamic information flow control, respectively.

In the dynamic approach, all information is labeled at runtime. Whenever information is used, labels on the results are set accordingly. Only low-labeled results are allowed to be sent to low outputs. Besides a difficulty with certain forms of hidden flows, there can be significant overhead in the label computations (cf. [12]).

In the static context, a type regime is established. Types are annotated with conservative approximations of the labels of runtime values. Type checking will enforce the correct handling of data. A big advantage is the low overhead at runtime. It is also possible to retain the annotation of source programs and use these for a lightweight verification of the programs on a user's computer.

Several languages have been proposed for static information flow control. Where proofs have been given, they are usually established in the standard manner of progress and subject reduction (or preservation). The underlying type system is either proven sound in the process, or assumed so: any program that can be typed in the underlying program cannot produce errors during execution. Thus it is unnecessary to include errors in the information flow control layer on top of the

underlying type system.

In the object-oriented setting, this means there are no message-not-found errors at runtime. This is a strong guarantee, which is hard to prove for many practical, dynamically-typed languages. In fact, recent empirical studies [74] showed that in the case of Javascript, most of the simplifying assumptions made in state-of-the-art work are violated in practice. A general type system that does not reject a significant number of practical programs seems out of reach at the moment. How is it then possible to prove noninterference statically?

We propose to explicitly handle the case of method-not-found errors in the computation. In this model, it is acceptable if a program fails because a method is to be called that the target object does not possess. However, errors cannot be allowed to transport information about private inputs.

We adopt the first-order version $\mathrm{Ob}_{1\leq}$ of the Object Calculus of Abadi and Cardelli [1]. This calculus is centered around objects as the only primitives. The allowed operations are object formation, method invocation, and method override. We extend the semantics to also allow method extension (i.e., adding new methods) and make errors explicit. Differing from previous work however, our target is not type soundness in the usual sense. Our types are overapproximations of the interface of an object. If an object can be typed, it will have a subset of the methods. Thus the meaning of an object type is that the invocation of a method either returns an object of the given method return type, or results in a method-not-found error. This allows us to control the invocation even of methods that do not exist, in a principled manner. If such a method is added through extension, it needs to agree with the previously stored type. Therefore, old judgments are still valid. To make the scheme work, we allow errors to be subtypes of all types.

This section is structured as follows: Section 6.1.2 gives an intuition about our long-term goal, using simple examples. In Section 6.1.3 we define the syntax and semantics of our calculus. Section 6.1.4 describes the type system we employ to enforce noninterference. A proof of noninterference for our calculus is given in Section 6.1.5, while Section 6.1.6 shows how to infer type annotations.

6.1.2 Example

In this section, we give a high-level example of what we are trying to accomplish. For simplicity, we assume a simple, imperative, object-based language that supports method extension. Then we could write a program like

```
o = ...
o.m()
```

With a standard type system, it might not be possible to establish that method `m` is defined when execution reaches the second line. Thus a standard type system approach will reject this program.

However, a missing method does not necessarily mean that there is a malicious flow of information, the usual suspect just being a bug. We propose to ignore the responsibilities of the underlying type system (i.e., ensuring the absence of method-not-found errors) and focus on information flow control even in the presence of such errors. There are two ways the program above might be safe: if we know that there is a method `m`, regardless of what high-level input the program received, or if we know there is no such method. In the first case, the computation will continue (possibly failing inside the call). In the second case, no matter the input, computation will fail at the second line. Thus, an attacker cannot gain more information than already available at that point.

Furthermore, if we cannot prove a method's existence or non-existence, but can show that that decision is only influenced by low conditions, a fail or successful call also does not leak information. Take for example the following program:

```
o = ...
if (b)
  o.n = function()...
o.n()
```

If the condition b is low, i.e., it is not confidential, an attacker will already know if the call will fail or not [1].

We propose a type system similar to a conservative analysis: types overapproximate the set of methods of a value. This allows us to type incomplete objects. In comparison to work in that area (see Related work, Section 8.4.1), we are more lenient with respect to method calls. For safety, previous work only allows a method call if it is statically known the method exists. We however do allow calls to all methods, since we do want to allow method-not-found errors. The reduction will terminate in an error state in the case of a missing method.

Different from the work on incomplete objects, we cannot allow an object's type to "forget" methods by subsumption. For our approximation to be safe, we can only increase the approximation.

In this work, we develop a type system in a simple first-order object calculus. In an imperative setting, information can be leaked through control flow structures and needs to be handled specially. In a functional setting, there are no control structures. Information gained through conditional execution (encoded through method calls) is transferred in the current object and never-decreasing. This eases the exposition of the idea, because if a call may fail in a high setting, the overall result of the program had to be high, too. For an imperative setting, it is necessary to also track side effects (e.g., [89, 11]).

6.1.3 Base Calculus

As mentioned, we adopt the first order calculus of Abadi and Cardelli [1]. This calculus is functional and centered around primitive objects. The calculus consists essentially of four elements: variables, object literals, method invocations and method overrides.

Object literals are records with method names and method bodies. Bodies are functions that take the object itself as their single argument (self-application semantics). Method bodies are only processed when invoked. The syntax uses the ς binder instead of λ to signal this late-binding

[1] If the call is made in a high environment, though, information would be leaked. A program position is in a high environment, if reaching this location depends on some confidential data. This is loosely related to [11].

semantics.

Method invocation selects a method by name from an object, and substitutes the self-parameter with the object in the method body. Method override replaces the function stored in an object. In Abadi's calculus, method override is restricted to replacing an already existing method. We add method extension to allow the extension of an object with new methods, which is a common feature in practical dynamic object-based languages.

This simple calculus does not allow method parameters besides the self-parameter. However, multi-argument functions can be emulated through additional methods that stand for parameters. Method override of those methods will emulate parameter passing. The same technique can also be used to encode the simple lambda calculus in the object calculus. The calculus does not directly support delegation and prototypes. However, a form of delegation can also be encoded. Alternatively, the calculus can be extended, as for example done in [8]. For a more detailed account of the encodings we refer to [1].

For information flow control, we assume the simple security lattice [32] $(L, \sqsubseteq)$ with bottom element $\bot$ and join operation $\sqcup$. Elements of the lattice are used as annotated labels in the calculus and represent confidentiality levels. Labels are ranged over by ϕ and ψ. The partial order $\sqsubseteq$ then orders the labels with respect to the sensitivity of the information: If $\phi \sqsubseteq \psi$, then any value labeled ϕ is assumed less confidential than a value labeled ψ, and any value tagged with ϕ can be used in places where a value with label ψ is needed. As an example, take $L = \{l, h\}$ with the meaning that $l = \bot$ stands for low=public information, and h for high=private. Then $l \sqsubseteq h$, so that 5^l is less confidential than 4^h.

The syntax is given in Figure 6.1. We add security annotations to most of the syntactic elements. The security annotations, denoted by ϕ, describe the confidentiality of the expression they are attached to, and will be preserved (or increased) during execution. The elements can be categorized into four traditional classes: variables, denoted by s; object literals of the form $[m_i = \varsigma(s_i)e_i]^{\phi}_{i \in I}$, which define objects with methods m_i having bodies e_i with a bound self variable s_i; method invocation denoted by "."; and object extension or method override denoted with

$o ::=$	s	Variable
	$[m_i = \varsigma(s_i)e_i]^{\phi}_{i \in I}$	Object
	$o.^{\phi}m$	Method invocation
	$o \leftarrow^{\phi} m = \varsigma(s)o'$	Method override or extension
	err	Error constant

Figure 6.1: Extended Syntax

(With $o = [m_i = \varsigma(s_i)o_i]^{\phi}_{i \in I}$)

$(Red\text{-}Inv)$

$$o.^{\phi}m_j \quad \rightarrow \quad o_j\{s_j := o\} \qquad j \in I$$

$(Red\text{-}Over)$

$$o \leftarrow^{\psi} m_j = \varsigma(s_j)o' \quad \rightarrow \qquad j \in I$$
$$[m_i = \varsigma(s_i)o_i, m_j = \varsigma(s_j)o']^{\phi}_{i \in I \setminus \{j\}}$$

$(Red\text{-}Ext)$

$$o \leftarrow^{\psi} m_j = \varsigma(s_j)o' \quad \rightarrow \qquad j \notin I$$
$$[m_i = \varsigma(s_i)o_i, m_j = \varsigma(s_j)o']^{\phi}_{i \in I}$$

Figure 6.2: Reduction (without errors)

$\leftarrow$. We include an explicit element for error. This element is not intended to be used by the programmer.

We extend the purely reduction-based semantics of [1] to retain annotations. Furthermore, our calculus allows the addition of methods. The reduction rules $(Red\text{-}Over)$ and $(Red\text{-}Ext)$ are applied depending on the shape of the object being extended. In the inherited case, $(Red\text{-}Over)$ applies when the object already has a method of the given name. Otherwise, $(Red\text{-}Ext)$ will add the method to the object. Since the calculus is functional, a new object will be returned in either case. It is noteworthy that both cases will be handled by a single typing rule. There is no reduction rule for variables (they are discharged through substitution), and no rule that reduces an object literal. All other terms are erroneous and will reduce to err. The non-error reduction rules are listed in Figure 6.2. The error reduction rules are listed in Figure 6.3. Evaluation is derived from reduction by contracting the leftmost, outermost redex. This simplifies the exposition and corresponds to the regular evaluation strategy of [1].

(With $o = [m_i = \varsigma(s_i)o_i]^{\phi}_{i \in I}$)

$$
\begin{array}{lrcll}
(Red\text{-}NotFound) & o.^{\phi}m_j & \rightarrow & \mathrm{err} & j \notin I \\
(Red\text{-}ErrInv) & \mathrm{err}.^{\phi}m_j & \rightarrow & \mathrm{err} & \\
(Red\text{-}ErrOver) & \mathrm{err} \leftarrow^{\psi} m_j = \varsigma(s_j)o' & \rightarrow & \mathrm{err} &
\end{array}
$$

Figure 6.3: Reduction (errors)

6.1.4 Type System

The goal of our type system is to enforce noninterference, not the absence of runtime errors (e.g., method-not-found errors). Thus it works in an irregular fashion more reminiscent of a conservative analysis. Types are ranged over by τ and σ, and are labeled. An object type $[m_i : \tau_i]^{\phi}_{i \in I}$ describes objects with an annotation at most ϕ and a subset of the methods m_i. Thus, all possible callable methods are contained in the type. We made errors explicit with the error constant err. The type of errors is E. Since an object type is an over-approximation of an object's interface, we include a subtyping relationship of E with every type. Thus, any method may potentially return err [2]. This makes the approximation safe with respect to the reduction. The type E and the error element err are implicitly labeled with $\bot$ and not a valid type for components of an object type. We use the notation $\tau^{\sqcup\phi}$ for the type that results from τ by replacing the label with the join of the old label and ϕ, e.g., $(\mathtt{int}^{L})^{\sqcup H} = \mathtt{int}^{L \sqcup H} = \mathtt{int}^{H}$.

Type environments Γ store assignments of types to variables in the usual way. The error type is not allowed in a type environment.

We have the usual judgments:

$$
\begin{array}{rl}
\Gamma \vdash * & \text{Well-formed environment} \\
\vdash \tau & \text{Well-formed type} \\
\vdash \tau_1 \leq \tau_2 & \text{Subtypes} \\
\Gamma \vdash o : \tau & \text{Term has type}
\end{array}
$$

[2] In a sense, this is similar to null and the null-type in practical languages.

$$\frac{\vdash \tau}{\vdash \tau \leq \tau}\ (Sub\text{-}Refl) \qquad \frac{\vdash \tau \leq \sigma \quad \vdash \sigma \leq \tau'}{\vdash \tau \leq \tau'}\ (Sub\text{-}Trans)$$

$$\frac{\vdash \tau}{\vdash \mathrm{E} \leq \tau}\ (Sub\text{-}Err) \qquad \frac{\vdash [m_i : \tau_i]^{\phi}_{i \in I} \quad \phi \sqsubseteq \psi}{\vdash [m_i : \tau_i]^{\phi}_{i \in I} \leq [m_i : \tau_i]^{\psi}_{i \in I}}\ (Sub\text{-}Partial)$$

$$\frac{\vdash [m_i : \tau_i, m_j : \tau_j]^{\phi}_{i \in I, j \in J}}{\vdash [m_i : \tau_i]^{\phi}_{i \in I} \leq [m_i : \tau_i, m_j : \tau_j]^{\phi}_{i \in I, j \in J}}\ (Sub\text{-}Ext)$$

Figure 6.4: Subtyping

Well-formedness is defined in the usual way. Note that we do not need the type environment for type-related judgments, since we do not support type variables.

We add subtyping rules that correspond to our semantics of types. If an object is of a type with a set of methods, it is also of a type with the same list of methods, extended by a new method. This unusual subtyping is safe because of the object-has-subset semantics. Furthermore, we lift the partial order of the security lattice to subtyping. We finalize subtyping with the usual reflexivity and transitivity. The subtyping rules are shown in Figure 6.4.

The typing rules are an adapted version of the Object Calculus'. Projection types variables according to the type environment. A subsumption rule is defined in the standard way. The rule $(T\text{-}Obj)$ types an object literal. We can type an object with type τ, if τ contains at least all the methods in the body, and we can type all method bodies under the assumption that the self parameter is of type τ. The $(T\text{-}Inv)$ rule types an invocation if we can type the receiver with a type that contains the method. The $(T\text{-}Over)$ rule is used both for extension and override. In the type system, they are identical. To extend a method means to replace an ("abstract") method already contained in the type. The typing rules are listed in Figure 6.5.

$$\frac{\Gamma, s : \tau, \Gamma' \vdash *}{\Gamma, s : \tau, \Gamma' \vdash s : \tau}\ (T\text{-}Prof) \qquad \frac{\Gamma \vdash o : \sigma \quad \vdash \sigma \leq \tau}{\Gamma \vdash o : \tau}\ (T\text{-}Sub)$$

$$\frac{\tau = [m_j : \sigma_j]^{\phi}_{j \in J} \quad I \subseteq J \quad \forall i \in I.\, \Gamma, s_i : \tau \vdash o_i : \sigma_i}{\Gamma \vdash [m_i = \varsigma(s_i) o_i]^{\phi}_{i \in I} : \tau}\ (T\text{-}Obj)$$

$$\frac{\Gamma \vdash o : [m_i : \tau_i]^{\phi}_{i \in I} \quad j \in I}{\Gamma \vdash o.^{\psi} m_j : \tau_j^{\sqcup \psi \sqcup \phi}}\ (T\text{-}Inv)$$

$$\frac{\tau = [m_i : \tau_i]^{\phi}_{i \in I} \quad \Gamma \vdash o : \tau \quad \Gamma, s_j : \tau \vdash o_j : \tau_j \quad j \in I}{\Gamma \vdash o \leftarrow^{\psi} m_j = \varsigma(s_j) o_j : \tau^{\sqcup \psi}}\ (T\text{-}Over)$$

Figure 6.5: Typing Rules

6.1.5 Noninterference

We proceed to establish noninterference along the following lines: First, we show subject reduction for our type system. Next, we show that reduction and substitution are orthogonal. Then we show that low-typed terms in high environments are either values or can be reduced. Finally, we can establish a version of noninterference.

For this treatment, we need a helper function for labels of types and type environments. $\mathrm{toplabel}(\tau)$ is the outermost annotation of type τ. The judgment $\mathrm{toplabel}(\Gamma) \not\sqsubseteq \phi$ is defined as $\forall (s : \tau) \in \Gamma.\mathrm{toplabel}(\tau) \not\sqsubseteq \phi$. Informally that means that all variables bound in the type environment are more confidential than the given label.

The proof of subject reduction structurally follows [1]. However, our semantics is small-step. We start with three standard auxiliary lemmas that are straightforward by induction and definition of the calculus.

Lemma 85 (Generation)**.** *If* $\Gamma \vdash [m_i = \varsigma(s_i) o_i]^{\phi}_{i \in I} : \tau$*, then there exists a type* $\sigma = [m_j : \tau_j]^{\phi}_{j \in J}$ *with* $I \subseteq J$*, such that* $\forall j \in I.\, \Gamma, s_j : \sigma \vdash o_j : \tau_j$ *and* $\Gamma \vdash [m_i = \varsigma(s_i) o_i]^{\phi}_{i \in I} : \sigma$ *and* $\vdash \sigma \leq \tau$*.*

Lemma 86 (Weakening)**.** *If* $\Gamma, s : \tau, \Gamma' \vdash o : \sigma$ *and* $\vdash \tau' \leq \tau$ *are derivable, then also* $\Gamma, s : \tau', \Gamma' \vdash o : \sigma$.

Lemma 87 (Substitution)**.** *If* $\Gamma, s : \tau, \Gamma' \vdash o : \sigma$ *and* $\Gamma \vdash p : \tau$ *are derivable, then also* $\Gamma, \Gamma' \vdash o\{s := p\} : \sigma$.

We are now able to prove the subject reduction property.

Theorem 88 (Subject Reduction)**.** *If* $\Gamma \vdash o : \tau$ *and* $o \to o'$*, then* $\Gamma \vdash o' : \tau$.

Proof. By induction on the derivation of $\Gamma \vdash o : \tau$.

Case $(T\text{-}Prof)$

In that case, o is a variable and there is no reduction.

Case $(T\text{-}Sub)$

The inductive hypothesis applies to the first premise. Another application of $(T\text{-}Sub)$ yields the result.

Case $(T\text{-}Obj)$

Object literals are values and cannot be reduced.

Case $(T\text{-}Inv)$

Then $o = p.^{\phi}m_j$. There are three sub-cases.

Error

If p is an error or a constant, then the term is reduced to err. The result is obtained by application of $(T\text{-}Sub)$.

Not a value

If p is not a value, the inductive hypothesis applies to the first premise and $p \to p'$. Thus, $o' = p'.^{\phi}m_j$ can be typed with τ.

Value

In case p is a value, i.e., an object literal, there are two cases. If j is not a valid index, the term is reduced to err and subtyping will give the right result. If j is a valid index, then by the first premise we have a typing of the object. This allows us to apply the generation lemma, which yields $\Gamma \vdash p : \sigma$ and $\Gamma, s_j : \sigma \vdash o_j : \tau_j$. From the substitution lemma it follows that $\Gamma \vdash o_j\{s_j := p\} : \tau_j$. The result follows by subtyping.

Case $(T\text{-}Over)$

Then $o = p \leftarrow^{\phi} m_j = \varsigma(s_j)o_j$. Again, we have the three cases for p, where the error/constant and non-value cases are analogous to $(T\text{-}Inv)$. In the case that p is a value, we have a typing for the literal by the first premise. Applying the generation lemma gives us typings of the already established methods with a type $\sigma \leq \tau$. We can use the weakening lemma with the second premise of $(T\text{-}Over)$ and receive a typing of the new method with respect to σ. We can combine all those results to type the object literal that is the result of either $(Red\text{-}Over)$ or $(Red\text{-}Ext)$, according to whether a method was already there. Subtyping completes the case.

□

We now show that reduction and substitution are orthogonal.

Lemma 89 (Subst Eval)**.** *If* $\Gamma, s : \sigma \vdash o : \tau$*,* $o \to o'$*, and* $\Gamma \vdash p : \sigma$*, then* $o\{s := p\} \to o'\{s := p\}$.

Proof. By induction on the derivation of $\Gamma, s : \sigma \vdash o : \tau$. Most cases are either vacuous or straightforward. We show the case of $(T\text{-}Inv)$, where $o = q.^{\phi}m_k$ and q is an object literal with method m_k. Then $o' = o_k\{s_k := q\}$ and $o\{s := q\} = (q\{s := q\}.^{\phi}m_k$. The substitution does not change the set of methods, so $(Red\text{-}Inv)$ can be applied. We need to distinguish the cases $s = s_k$ and $s \neq s_k$ when we apply the reduction. Finally, rewriting the term according to the standard substitution rules results in $o_k\{s_k := q\}\{s := p\}$. □

The next to last step is progress of lowly-typed terms in high environments.

Lemma 90 (Low Progress)**.** *If* $\Gamma \vdash o : \tau$ *with* $\text{toplabel}(\Gamma) \not\sqsubseteq \text{toplabel}(\tau)$*, then either* o *is a value (object literal, constant or* err*), or there is an* o' *so that* $o \to o'$*.*

Proof. By induction on the derivation of $\Gamma \vdash o : \tau$.

Case $(T\text{-}Prof)$

In that case, o is a variable and $(o = s : \tau)$ is part of the type environment. Thus, $\neg(\text{toplabel}(\Gamma) \not\sqsubseteq \text{toplabel}(\tau))$, so the second premise of the lemma is not fulfilled.

Case $(T\text{-}Sub)$

Subtyping maintains the partial order of security labels. Thus the inductive hypothesis applies to the first premise. So there exists an o' such that $o \to o'$.

Case $(T\text{-}Obj)$

Object literals are values, which fulfills the lemma.

Case $(T\text{-}Inv)$

Then $o = p.^{\phi}m_j$. There are three sub-cases.

Error

If p is an error or a constant, then the reduction $p.^{\phi}m_j \to \text{err}$ applies.

Not a value

If p is not a value, the inductive hypothesis applies to the first premise (same type, same environment) and $p \to p'$. Thus, $p.^{\phi}m_j \to p'.^{\phi}m_j$.

Value

In case p is a value, i.e., an object literal, there are two cases. If j is not a valid index, the reduction that applies is $o \to \text{err}$. If j is a valid index, then we can apply $(Red\text{-}Inv)$. Thus we get $o \to o_j\{s_j := p\}$.

Case $(T\text{-}Over)$

Then $o = p \leftarrow^{\phi} m_j = \varsigma(s_j)o_j$. Again, we have the three cases for p, where the error and non-value cases are analogous. In the case that p is a value, we can either apply $(Red\text{-}Over)$ or $(Red\text{-}Ext)$, depending on the existence of m_j in p.

□

Finally, we approach noninterference. In the original calculus there are only primitive objects. To state noninterference, we need to define an equivalence relation on objects with respect to a given labeling. A simple definition is observational equivalence, where an attacker is allowed to invoke only low-typed methods, which corresponds to the ability of an attacker to inspect low-typed heap elements in an imperative model. The definition is slightly complicated by the fact that methods in literals are not annotated with types.

Let $\Downarrow$ denote the "big-step" closure of the reduction, i.e., $o \Downarrow v$ iff v is a value and $o \rightarrow^* v$. Note that the reduction rules are deterministic, so there is at most one value v. Furthermore, let $o \Downarrow \bot$ stand for an infinite, non-terminating computation, i.e., there is no value v such that $o \rightarrow^* v$.

Definition 91 (Object Equivalence)**.** *Two objects o_1 and o_2 are equivalent with respect to security level ϕ, written as $o_1 \sim_\phi o_2$, iff for all types $\tau = [m_i : \tau_i]_{i \in I}^{\psi}$ with $\psi \sqsubseteq \phi$, $\emptyset \vdash o_1 : \tau$ and $\emptyset \vdash o_2 : \tau$: For all methods m_i with $\mathrm{toplabel}(\tau_i) \sqsubseteq \phi$:*

1. *If $o_1.^{\bot}m_i \Downarrow v_1$, then for some v_2, $o_2.^{\bot}m_i \Downarrow v_2$ and $v_1 \sim_\phi v_2$.*
2. *If $o_2.^{\bot}m_i \Downarrow v_2$, then for some v_1, $o_1.^{\bot}m_i \Downarrow v_1$ and $v_1 \sim_\phi v_2$.*

Now we can state our non-interference theorem.

Theorem 92 (Noninterference)**.** *If $s : \sigma \vdash o : \tau$ (where $\mathrm{toplabel}(\tau) = \phi$) with $\mathrm{toplabel}(\sigma) \not\sqsubseteq \phi$, $\emptyset \vdash p : \sigma$ and $\emptyset \vdash p' : \sigma$, then either both computations $o\{s := p\}$ and $o\{s := p'\}$ diverge, or both computations return results equivalent with respect to ϕ.*

Proof. If $\mathrm{toplabel}(\sigma) \not\sqsubseteq \mathrm{toplabel}(\tau)$, we can apply the low-progress lemma to o. With subject reduction, we can repeat the process. This results in a chain of reductions $o \to o' \to o'' \to \ldots$, that is either finite and ends with a value (by the low-progress lemma), or that is infinite, i.e., the computation diverges.

We can now apply the subst-eval lemma to this chain for both p and p'. The result is, that the computations for $o\{s := p\}$ and $o\{s := p'\}$ diverge exactly if the computation of o diverges.

Now we can handle the case of converging computations. By subst-eval, the computations are $o \Downarrow v$, $o\{s := p\} \Downarrow v\{s := p\}$ and $o\{s := p'\} \Downarrow v\{s := p'\}$ for some v. To show equivalence, we use bisimulations, or more precisely the technique of strong bisimulations up to $\sim$ [65]. We choose the following binary relation:

$$S_\phi = \left\{ \begin{array}{l|r} (q\{s := p\}, & \exists \tau, \sigma.\ s : \sigma \vdash q : \tau \wedge \\ \quad q\{s := p'\}) & \vdash p : \sigma \wedge \vdash p' : \sigma \\ & \mathrm{toplabel}(\sigma) \not\sqsubseteq \phi \wedge \\ & \mathrm{toplabel}(\sigma) \not\sqsubseteq \mathrm{toplabel}(\tau) \end{array} \right\}$$

We need to show that S_ϕ is closed under invocation of low-typed methods, that is, if $(q\{s := p\}, q\{s := p'\}) \in S_\phi$, and $s : \sigma \vdash q : [m_i : \tau_i]^{\psi}_{i \in I}$, then for all m_i with $\mathrm{toplabel}(\tau_i) \sqsubseteq \phi$ with $q\{s := p\} \Downarrow v$ and $q\{s := p'\} \Downarrow v'$ we have $(v, v') \in S_\phi$.

The approach is similar to the one above. First, since $\mathrm{toplabel}(\tau_i) \sqsubseteq \phi$, $\mathrm{toplabel}(\sigma) \not\sqsubseteq \mathrm{toplabel}(\tau_i)$. Thus we can apply low-progress, subject reduction and subst-eval. As above, either both computations return results or diverge. If they converge, the computations are $q.^{\perp} m_i \Downarrow v_0$, $q\{s := p\}.^{\perp} m_i \Downarrow v_0\{s := p\}$ and $q\{s := p'\}.^{\perp} m_i \Downarrow v_0\{s := p'\}$ for some v_0. Now $s : \sigma \vdash v_0 : \tau_i$, $\vdash p : \sigma$, $\vdash p' : \sigma$, $\mathrm{toplabel}(\sigma) \not\sqsubseteq \phi$ and $\mathrm{toplabel}(\sigma) \not\sqsubseteq \mathrm{toplabel}(\tau_i)$. Thus, $(v, v') \in S_\phi$. This concludes the proof. □

Note that err is different from all other elements. Thus, if one computation fails with an error, so must the other. It is not possible to use the message-not-understood error to gain information

x is a variable:
$$x \leq [\![x]\!]$$

$[m_i = \varsigma(x_i)o_i]$ is a literal:
$$[m_i : [\![o_i]\!]] \leq [\![[m_i = \varsigma(x_i)o_i]]\!]$$
$$x_i = \underline{[\![[m_i = \varsigma(x_i)o_i]]\!]}$$

$o.m_i$ is invocation:
$$[m_i : \langle o.m_i \rangle] \underline{\leq} [\![o]\!]$$
$$\langle o.m_i \rangle \leq [\![o.m_i]\!]$$

$o \leftarrow m_i = \varsigma(x_i)o_i$ is method extension/override:
$$[\![o]\!] \leq [\![o \leftarrow m_i = \varsigma(x_i)o_i]\!]$$
$$[m_i : [\![o_i]\!]] \underline{\underline{\leq}} [\![o]\!]$$
$$[\![o]\!] = x$$

Figure 6.6: Constraint Rules

about the input.

6.1.6 Inference

It is possible to adopt the work of Palsberg [71] for type inference [3]. The inference algorithm is constraint-based, and works by reducing constraints first to a graph and then an automaton. The language accepted by the automaton for different start states describes the types of the corresponding expressions.

Because of our changed typing rules, and most significantly the change in the subtyping relationship, compared to the original object calculus, a modification of the constraint rules is necessary. The adapted rules are given in Figure 6.6, where changes to Palsberg's work have been underlined. With those changes, it is easy to prove that a system of constraints generated by an expression has a solution if and only if the expression is typeable, and the solution for the expression is a valid type (cf. Lemma 4.2 in [71]).

The system of constraints is then translated into an equivalent graph representation. Types and expressions become nodes. Constraints establish directed $\leq$-labeled edges, while types also

[3] In our context, typing, since we do not use explicit method parameter annotations.

introduce method-labeled edges. The graph is closed for $\leq$ edges, that is, edges for reflexivity and transitivity of $\leq$ are added. Furthermore, since the type system only supports width-subtyping, edges are included to enforce this property[4]. A solution of the graph is a mapping of variable nodes to types, such that the subtyping on edges is satisfied. It is obvious that a solution to a constraint system exists if and only if a solution to the corresponding graph exists (cf. Theorem 5.2 in [71]).

Finally, the graph is used to derive an automaton with the method names as alphabet and states corresponding to the nodes of the graph. All states are accepting. The automaton has transitions between states with ϵ if there's a $\leq$-labelled edge, and with the name of the method on a method-labelled edge. However, since our subtyping relation is inverse to that of Abadi and Cardelli, and the typing rules usually establish lower bounds, we inverse the direction of the $\leq$-edges, e.g., a constraint $a \leq b$ induces an $a \overset{\leq}{\leftarrow} b$, which defines a transition $b \xrightarrow{\epsilon} a$.

The language $\mathcal{L}(s)$ is defined as all the words accepted when starting from state s. It can be shown that $\mathcal{L}$ is a solution for the graph (cf. Theorem 6.5 in [71]). The proofs in [71] have to be adapted for the reversed subtyping relationship, but are structurally the same. If the language $\mathcal{L}$ is finite for the top-level expression, the types are finite and can be expressed in our system. Otherwise, recursive types are necessary.

From this basic inference of the structure of types, we can iteratively propagate labels from free variables, which need to be defined in a type environment, to obtain a complete typing. All types are initially assumed to be low. If a type mapping contains a high label, it is joined, into the enclosing expression according to the type rules. Then the process is repeated, until either the propagation finishes or a contradiction is found.

[4]Note that the closure is changed in direction to account for our subtyping.

6.2 SQL

We only sketch our security-typed fragment of SQL here. States are sets of named tables of integer data. The syntax of the fragment is given by

$$\begin{array}{rcl} \hat{t} & ::= & \dot{t} \mid \dot{t} \,\mathrm{join}\, \dot{t}' \,\mathrm{on}\, i = i' \qquad\qquad \dot{e} ::= \dot{n} \mid i \mid \dot{x} \mid \dot{e} \oplus \dot{e}' \\ \dot{c} & ::= & \mathrm{select}\, \dot{e}_1, \ldots, \dot{e}_n \,\mathrm{from}\, \hat{t} \,\mathrm{where}\, \dot{e}' \mid \mathrm{insert}\, i = \dot{e} \,\mathrm{into}\, \dot{t} \mid \\ & & \mathrm{update}\, i = \dot{e} \,\mathrm{in}\, \dot{t} \,\mathrm{where}\, \dot{e}' \mid \mathrm{delete}\, \mathrm{from}\, \dot{t} \,\mathrm{where}\, \dot{e} \end{array}$$

The semantics is straightforward. We add a security-type system with judgments relative to environments Δ that map table columns to security labels. The type judgment has the form $\Delta, \dot{\Gamma} \vdash \dot{c} : (\ell_1, \ldots, \ell_n)^{\ell'}/\ell_s \,\mathrm{ok}$, where ℓ_i are the levels of the result columns, ℓ' is the level of the whole result and ℓ_s is a lower bound on the side effect of $\dot{c}$. States are indistinguishable with respect to Δ and ℓ if the erasure of columns not typed at or below ℓ is equivalent. Typable queries can be shown to be noninterfering.

Theorem 93 (SQL Noninterference)**.**

$$\begin{array}{l} \forall \Delta, \ell_\bullet, \ell', \ell'', \ell_x, c, \mu_1, \mu_2, \mu_1', \mu_2', \dot{n}_1, \dot{n}_2, s_1, s_2. \\ \quad \Delta, x : \ell_x \vdash c : \ell_\bullet^{\ell'}/\ell'' \,\mathrm{ok} \wedge \mu_1 \dot{\sim}_{\Delta,\ell_o} \mu_2 \wedge \dot{n}_1 \dot{\sim}_{\ell_x,\ell_o} \dot{n}_2 \wedge \mu_1, c[x := \dot{n}_1] \Rightarrow \mu_1', s_1 \wedge \\ \quad \mu_2, c[x := \dot{n}_2] \Rightarrow \mu_2', s_2 \\ \qquad\qquad\qquad\qquad \Longrightarrow\ \mu_1' \dot{\sim}_{\Delta,\ell_o} \mu_2' \wedge s_1 \dot{\sim}_{\ell_\bullet^{\ell'},\ell_o} s_2 \end{array}$$

6.2.1 Language

We formalize a simplified version of the data retrieval and manipulation fragment of SQL. We allow (finitely many) named tables of integer data. Names $\dot{t}$ are drawn from a set $\mathcal{T}$. The set of tables is static and finite for a particular program: we do not allow table creation or deletion. Tables have a finite number of columns $\mathcal{I}_{\dot{t}}$, where we assume for simplicity that column names are distinct between tables. We use i to range over *all* column names. We have the following syntax, where

we use a dot to distinguish from the WHILE syntax.

$$\begin{array}{rcll} \hat{t} & ::= & \dot{t} \mid \dot{t} \operatorname{join} \dot{t}' \operatorname{on} i = i' & \qquad \dot{e} ::= \dot{n} \mid i \mid \dot{x} \mid \dot{e} \oplus \dot{e}' \\ \dot{c} & ::= & \operatorname{select} \dot{e}_1, \ldots, \dot{e}_n \operatorname{from} \hat{t} \operatorname{where} \dot{e}' \mid \operatorname{insert} i = \dot{e} \operatorname{into} \dot{t} \mid & \\ & & \operatorname{update} i = \dot{e} \operatorname{in} \dot{t} \operatorname{where} \dot{e}' \mid \operatorname{delete} \operatorname{from} \dot{t} \operatorname{where} \dot{e} & \end{array}$$

A table state T is represented as a finite map of naturals to records r, which are a total map of column names to integers. Note that we identify states up to an order-preserving remapping. A database state ν is a finite map of table names to table states. For simplicity of handling table joins we extend the set of table names by table expressions, e.g., if A and B are table names and i and i' column names, then $A \operatorname{join} B \operatorname{on} i = i'$ is a table name.

Expressions can be evaluated for a record, denoted by $r(\dot{e})$, in the obvious way. Expressions do not change the database. Table expressions project and manipulate table states from a database state in the obvious way. Table projection and manipulation is formally defined in the following way:

$$r \oplus r' = \lambda i. \begin{cases} r.i & i \in dom(r) \\ r'.i & i \in dom(r') \wedge i \notin dom(r) \\ \bot & \text{Otherwise} \end{cases}$$

$$\nu(\hat{t}) = \nu(\dot{t}) \quad \text{if } \hat{t} = \dot{t}$$

$$\nu(\dot{t} \operatorname{join} \dot{t}' \operatorname{on} i = i') = \lambda n. \begin{cases} r \oplus r' & r = \nu(\dot{t})(n/|\nu(\dot{t})|).i \wedge r' = \nu(\dot{t}')(n\%|\nu(\dot{t})|) \wedge r.i = r'.i' \\ \bot & \text{Otherwise} \end{cases}$$

A natural semantics reduces a statement and database state to a result and a state. We use the

following auxiliary definitions.

$$T^{\wedge}_{\dot{e}} = \lambda n. \begin{cases} r & T(n) = r \wedge r(\dot{e}) \neq \dot{0} \\ \bot & otherwise \end{cases} \qquad T^{\vee}_{\dot{e}} = \lambda n. \begin{cases} r & T(n) = r \wedge r(\dot{e}) = \dot{0} \\ \bot & otherwise \end{cases}$$

$$(i_1 : \dot{n}_1, \dots).i_j \leftarrow \dot{n}' = (i_1 : \dot{n}_1, \dots, i_j : \dot{n}', \dots)$$

$$T^{\leftarrow}_{i,\dot{e},\dot{e}'} = \lambda n. \begin{cases} r.i \leftarrow r(\dot{e}) & T^{\wedge}_{\dot{e}'}(n) = r \\ T(n) & otherwise \end{cases}$$

Here $T^{\wedge}_{\dot{e}}$ restricts the domain of T to the records satisfying $\dot{e}$, while $T^{\vee}_{\dot{e}}$ restricts to the opposite. Finally $T^{\leftarrow}_{i,\dot{e},\dot{e}'}$ computes a new map where columns i of records r that satisfy $\dot{e}'$ are updated to $r(\dot{e})$. Now with the intent that $\bot(\dot{e}) = \bot$ the semantic rules are defined as follows.

$$\nu, \text{select } \dot{e}_1, \dots, \dot{e}_k \text{ from } \hat{t} \text{ where } \dot{e}' \;\Rightarrow\; \nu, \lambda n. \left(\nu(\hat{t})^{\wedge}_{\dot{e}'}(n)(\dot{e}_1), \dots, \nu(\hat{t})^{\wedge}_{\dot{e}'}(n)(\dot{e}_k)\right)$$

$$\nu, \text{insert } i_j = \dot{n} \text{ into } \dot{t} \;\Rightarrow\; \nu[\dot{t} := \nu(\dot{t}) + (i_1 : \dot{0}, \dots, i_{j-1} : \dot{0}, i_j : \dot{n}, i_{j+1} : \dot{0}, \dots)], \emptyset$$
$$(where \ + \ extends \ \nu(\dot{t})'s \ domain)$$

$$\nu, \text{update } i = \dot{e} \text{ in } \dot{t} \text{ where } \dot{e}' \;\Rightarrow\; \nu[\dot{t} := \nu(\dot{t})^{\leftarrow}_{i,\dot{e},\dot{e}'}], \emptyset$$

$$\nu, \text{delete from } \dot{t} \text{ where } \dot{e} \;\Rightarrow\; \nu[\dot{t} := \nu(\dot{t})^{\vee}_{\dot{e}}], \emptyset$$

We assign security levels to tables for two functions. First, all tables have a simple level that describes the confidentiality of the table itself. Observers not at or above this level cannot access a table at all. Second, we assign security levels to columns, that is, all records have the same security level for the corresponding columns. We denote the mapping of column names of a table to security levels by δ, and $\ell_\delta = \bigsqcap \delta(i)$. For simplicity, we map the table security level to the synthetic column name $table$.

For typing purposes, we collect all δ in Δ which maps from table names. For table expression names, we define the following evaluation.

$$\Delta(\dot{t}_1 \operatorname{join} \dot{t}_2 \operatorname{on} i_1 = i_2) = \lambda i. \begin{cases} \bigsqcup_{i \in \{1,2\}} \left(\Delta(\dot{t}_i)(table) \sqcup \Delta(\dot{t}_i)(i_i)\right) & i = table \\ \Delta(\dot{t}_1)(i) & i \in \Delta(\dot{t}_1) \\ \Delta(\dot{t}_2)(i) & i \in \Delta(\dot{t}_2) \wedge i \notin \Delta(\dot{t}_1) \\ \bot & \text{else} \end{cases}$$

The type system is defined by the following sets of rules, where $\dot{\Gamma}$ is a typing environment mapping variables to security levels. Note that typing of expressions refers to a table typing δ, whereas statement typing uses Δ. Well-formedness requirements for δ and Δ are standard.

$$\delta, \dot{\Gamma} \vdash \dot{n} : \ell \quad \delta, \dot{\Gamma} \vdash i : \delta(i) \quad \delta, \dot{\Gamma} \vdash \dot{x} : \dot{\Gamma}(\dot{x})$$

$$\frac{\delta, \dot{\Gamma} \vdash \dot{e} : \ell_1 \quad \delta, \dot{\Gamma} \vdash \dot{e}' : \ell_2}{\delta, \dot{\Gamma} \vdash \dot{e} \oplus \dot{e}' : \ell_1 \sqcup \ell_2} \qquad \frac{\delta, \dot{\Gamma} \vdash \dot{e} : \ell \quad \ell \sqsubseteq \ell'}{\delta, \dot{\Gamma} \vdash \dot{e} : \ell'}$$

$$\frac{\delta = \Delta(\hat{t}) \quad \forall i. \delta, \dot{\Gamma} \vdash \dot{e}_i : \ell_i \quad \delta, \dot{\Gamma} \vdash \dot{e}' : \ell \quad \ell \sqcup \delta(table) \sqsubseteq \ell'}{\Delta, \dot{\Gamma} \vdash \text{select } \dot{e}_1, \ldots, \dot{e}_n \text{ from } \hat{t} \text{ where } \dot{e}' : (\ell_1, \ldots, \ell_n)^{\ell'} / \top \text{ ok}}$$

$$\frac{\delta = \Delta(\dot{t}) \quad \delta, \dot{\Gamma} \vdash \dot{e} : \ell \quad \ell \sqsubseteq \delta(i)}{\Delta, \dot{\Gamma} \vdash \text{insert } i = \dot{e} \text{ into } \dot{t} : \bot^{\bot} / \ell_\delta \text{ ok}} \qquad \frac{\delta = \Delta(\dot{t}) \quad \delta, \dot{\Gamma} \vdash \dot{e} : \delta(table)}{\Delta, \dot{\Gamma} \vdash \text{delete from } \dot{t} \text{ where } \dot{e} : \bot^{\bot} / \ell_\delta \text{ ok}}$$

$$\frac{\delta = \Delta(\dot{t}) \quad \delta, \dot{\Gamma} \vdash \dot{e} : \ell_1 \quad \delta, \dot{\Gamma} \vdash \dot{e}' : \ell_2 \quad \ell_1 \sqcup \ell_2 \sqsubseteq \delta(i)}{\Delta, \dot{\Gamma} \vdash \text{update } i = \dot{e} \text{ in } \dot{t} \text{ where } \dot{e}' : \bot^{\bot} / \delta(i) \text{ ok}}$$

For SQL we have to formalize to notions of indistinguishability. We use a projection to erase all

information in a record that is not typed at or below ℓ, and lift it to sequences.

$$\downarrow^{\delta}_{\ell}(r).i = \begin{cases} r.i & \delta(i) \sqsubseteq \ell \\ 0 & \text{otherwise} \end{cases} \qquad \downarrow^{\delta}_{\ell}(f) = \lambda n.\downarrow^{\delta}_{\ell}(f(n))$$

We define two result sets as indistinguishable with respect to a type $\ell^{\ell'}_{\bullet}$ (where $\ell_{\bullet}$ defines a sequence of labels), abusing notation of projection, as $s_1 \dot{\sim}_{\ell^{\ell'}_{\bullet},\ell_o} s_1 \iff (\ell' \sqsubseteq \ell_o \implies (|s_1| = |s_2| \wedge \downarrow^{\ell_{\bullet}}_{\ell_o}(s_1) = \downarrow^{\ell_{\bullet}}_{\ell_o}(s_2)))$. Projection is also used to define indistinguishability of two table states, which are indistinguishable with respect to observer level ℓ if they agree on all columns at most ℓ: $T_1 \dot{\sim}_{\delta,\ell} T_2 \iff \delta(table) \sqsubseteq \ell \implies \downarrow^{\delta}_{\ell}(T_1) = \downarrow^{\delta}_{\ell}(T_2)$. Finally, indistinguishability of table states is lifted component-wise to database states: $\nu_1 \dot{\sim}_{\Delta,\ell} \nu_2 \iff \forall t \in \mathcal{T}.\nu_1(t) \dot{\sim}_{\Delta(t),\ell} \nu_2(t)$

6.2.2 Proofs

The proof of SQL noninterference relies on two auxiliary lemmas about table projections. The first one formulates a statement for restrictions in indistinguishable inputs.

Lemma 94 (Restriction Indistinguishability)**.** *If δ, $\ell_e \sqsubseteq \ell_o$, ℓ_x, T_1, T_2, $\dot{n}_1$, $\dot{n}_2$, and $\dot{e}$ such that $\delta, [\dot{x} : \ell_x] \vdash \dot{e} : \ell_e$, $T_1 \dot{\sim}_{\delta,\ell_o} T_2$, and $\dot{n}_1 \dot{\sim}_{\ell_x,\ell_o} \dot{n}_2$, then $T_1{}^{\wedge}_{\dot{e}[\dot{x}:=\dot{n}_1]} \dot{\sim}_{\delta,\ell_o} T_1{}^{\wedge}_{\dot{e}[\dot{x}:=\dot{n}_2]}$ and $T_1{}^{\vee}_{\dot{e}[\dot{x}:=\dot{n}_1]} \dot{\sim}_{\delta,\ell_o} T_1{}^{\vee}_{\dot{e}[\dot{x}:=\dot{n}_2]}$.*

Proof. First consider the case that $\delta(table) \sqsubseteq \ell_o$, the other case is trivial. Then $\downarrow^{\delta}_{\ell_o}(T_1) = \downarrow^{\delta}_{\ell_o}(T_2)$. Next, either $\dot{e}$ does not contain $\dot{x}$, or $\ell_x \sqsubseteq \ell_e$. In the latter case, $\dot{n}_1 = \dot{n}_2$. So always $\dot{e}[\dot{x} := \dot{n}_1] = \dot{e}[\dot{x} := \dot{n}_2] = \dot{e}'$. Now for each $n \in dom(T_1) = dom(T_2)$, we have $T_1(n)(\dot{e}') = T_2(n)(\dot{e}')$, because the records must agree on the columns referenced in $\dot{e}'$ by $\downarrow$. Thus, a record gets selected for $T_1{}^{\wedge}_{\dot{e}'}$ if and only if it gets selected for $T_2{}^{\wedge}_{\dot{e}'}$, and selected for $T_1{}^{\vee}_{\dot{e}'}$ if and only if it gets selected for $T_2{}^{\vee}_{\dot{e}'}$. Since the records are not modified otherwise, it follows that $T_1{}^{\wedge}_{\dot{e}'} \dot{\sim}_{\delta,\ell_o} T_1{}^{\wedge}_{\dot{e}'}$ and $T_1{}^{\vee}_{\dot{e}'} \dot{\sim}_{\delta,\ell_o} T_1{}^{\vee}_{\dot{e}'}$. □

This lemma states that typed table joins from indistinguishable inputs create indistinguishable outputs.

Lemma 95 (Join Noninterference)**.** *If Δ, ℓ_o, ν_1 and ν_2 such that $\nu_1 \dot{\sim}_{\Delta,\ell_o} \nu_2$, then $\nu_1(\hat{t}) \dot{\sim}_{\Delta(\hat{t}),\ell_o} \nu_2(\hat{t})$.*

Proof. By analysis of $\hat{t}$.

$\hat{t} = \dot{t}$. Follows by the definition of $\dot{\sim}_{\Delta,\ell_o}$.

$\hat{t} = \dot{t}_1 \operatorname{join} \dot{t}_2 \operatorname{on} i_1 = i_2$. Let $\delta_1 = \Delta(\dot{t}_1)$, $\delta_2 = \Delta(\dot{t}_2)$, $T_1^1 = \nu_1(\dot{t}_1)$, $T_2^1 = \nu_1(\dot{t}_2)$, $T_1^2 = \nu_2(\dot{t}_1)$, and $T_2^2 = \nu_2(\dot{t}_2)$. Furthermore, let $\delta_j = \Delta(\hat{t})$ and $T_j^1 = \nu_1(\hat{t})$ and $T_j^2 = \nu_2(\hat{t})$. Let $\delta_j(table) \sqsubseteq \ell_o$, the other case is trivial. Then by construction $\delta_1(table) \sqsubseteq \ell_o$ and $\delta_2(table) \sqsubseteq \ell_o$. Thus $\downarrow_{\ell_o}^{\delta_1}(T_1^1) = \downarrow_{\ell_o}^{\delta_1}(T_1^2)$ and $\downarrow_{\ell_o}^{\delta_2}(T_2^1) = \downarrow_{\ell_o}^{\delta_2}(T_2^2)$. We need to show that $T_j^1 \dot{\sim}_{\delta_j,\ell_o} T_j^2$. First note that δ_j agrees on the levels with δ_1 and δ_2 for the respective columns. Next, selecting records to join in $\oplus$ happens at $\delta_1(i_1)$ and $\delta_2(i_2)$. By this case, these are $\delta_1(i_1) \sqsubseteq \ell_o$ and $\delta_2(i_2) \sqsubseteq \ell_o$. Thus the original values are retained in $\downarrow_{\ell_o}^{\delta_j}(\ldots)$, which by indistinguishability means that the respective tables have equivalent values in those columns. Thus corresponding records are merged for both inputs. Pick a generated record. For any column typed at or below ℓ_o, we have that the corresponding column from T^1 or T^2 is also typed at or below ℓ_o by construction of join. Thus, the corresponding values in T_1 and T_2 are equivalent, which means that the column in the joined records are equivalent. Thus, $\downarrow_{\ell_o}^{\delta_j}(T_j^1) = \downarrow_{\ell_o}^{\delta_j}(T_j^2)$.

□

Finally, the proof of SQL noninterference is a case analysis, invoking the previous two lemmas in the case of select queries.

Theorem 96 (SQL Noninterference)**.**

$$
\begin{aligned}
&\forall \Delta, \ell_\bullet, \ell', \ell'', \ell_x, c, \mu_1, \mu_2, \mu_1', \mu_2', \dot{n}_1, \dot{n}_2, s_1, s_2. \\
&\Delta, x : \ell_x \vdash c : \ell_\bullet^{\ell'} / \ell'' \operatorname{ok} \wedge \mu_1 \dot{\sim}_{\Delta,\ell_o} \mu_2 \wedge \dot{n}_1 \dot{\sim}_{\ell_x,\ell_o} \dot{n}_2 \wedge \\
&\mu_1, c[x := \dot{n}_1] \Rightarrow \mu_1', s_1 \wedge \mu_2, c[x := \dot{n}_2] \Rightarrow \mu_2', s_2 \wedge \\
&\implies \mu_1' \dot{\sim}_{\Delta,\ell_o} \mu_2' \wedge s_1 \dot{\sim}_{\ell_\bullet^{\ell'},\ell_o} s_2
\end{aligned}
$$

Proof. By case analyis for c. We use the following definitions where applicable: $\delta = \Delta(\dot{t})$, $T_1 = \nu_1(\dot{t})$, $T_1' = \nu_1'(\dot{t})$, $T_2 = \nu_2(\dot{t})$ and $T_2' = \nu_2(\dot{t})$.

$\mathrm{select}\ \dot{e}_\bullet\ \mathrm{from}\ \hat{t}\ \mathrm{where}\ \dot{e}'$. The semantics of select shows that database states remain unchanged, i.e., $\nu_1 = \nu_1'$ and $\nu_2 = \nu_2'$. Thus, by premises, the output states are indistinguishable at ℓ_o. Now a case decision on the typing of the result sets. If $\ell' \not\sqsubseteq \ell_o$, then the results are automatically indistinguishable by construction. So assume that $\ell' \sqsubseteq \ell_o$. Let $T_1 = \nu_1(\hat{t})$, $T_2 = \nu_2(\hat{t})$, and $\delta = \Delta(\hat{t})$. By Lemma 95, it follows that $T_1 \dot{\sim}_{\delta,\ell_o} T_2$. Next, by Lemma 94, we have that $T_1 \overset{\wedge}{\dot{e}'[\dot{x}:=\dot{n}_1]} \dot{\sim}_{\delta,\ell_o} T_2 \overset{\wedge}{\dot{e}'[\dot{x}:=\dot{n}_2]}$. This means that the same number of records gets selected from each table, which means that the outputs have the same number of elements, that is, $|s_1| = |s_2|$.

Now assume also that $\ell_i \sqsubseteq \ell_o$. Then either $\dot{e}_i$ does not include $\dot{x}$, or $\ell_x \sqsubseteq \ell_o$. So, $\dot{e}_i[\dot{x} := \dot{n}_1] = \dot{e}_i[\dot{x} := \dot{n}_2]$. Furthermore, it follows that $\dot{e}_i$ only uses columns typed at or below ℓ_o. As derived above, $T_1 \overset{\wedge}{\dot{e}'[\dot{x}:=\dot{n}_1]} \dot{\sim}_{\delta,\ell_o} T_2 \overset{\wedge}{\dot{e}'[\dot{x}:=\dot{n}_2]}$, or $\downarrow^{\delta}_{\ell_o}(T_1 \overset{\wedge}{\dot{e}'[\dot{x}:=\dot{n}_1]}) = \downarrow^{\delta}_{\ell_o}(T_2 \overset{\wedge}{\dot{e}'[\dot{x}:=\dot{n}_2]})$. Thus, the selected records agree on columns at or below ℓ_o. Then the evaluation of $\dot{e}_i[\dot{x} := \dot{n}_{1/2}]$ will create the same result for corresponding records. Thus, the projected records have the same values for the result of $\dot{e}_i$. It follows that $\downarrow^{\ell_\bullet}_{\ell_o}(s_1) = \downarrow^{\ell_\bullet}_{\ell_o}(s_2)$.

$\mathrm{update}\, i = \dot{e}\, \mathrm{in}\, \dot{t}\, \mathrm{where}\, \dot{e}'$ For an update, the result sequences are obviously equivalent and thus indistinguishable. Assume that $\delta(table) \sqsubseteq \ell_o$, the opposite case is trivial. First, only the table $\dot{t}$ is potentially modified. All other tables remain the same. Second, the size of $\dot{t}$ doesn't change, only record data is modified. Now several case decisions on whether (1) $\ell_1 \sqsubseteq \ell_o$ and (2) $\ell_2 \sqsubseteq \ell_o$.

FF,FT,TF. Then $\delta(i) \not\sqsubseteq \ell_o$. It follows that since $\downarrow^{\delta}_{\ell_o}(T_1) = \downarrow^{\delta}_{\ell_o}(T_2)$ we have $\downarrow^{\delta}_{\ell_o}(T_1') = \downarrow^{\delta}_{\ell_o}(T_2')$.

TT. Then neither $\dot{e}$ nor $\dot{e}'$ do contain x, or $\ell_x \sqsubseteq \ell_o$, so that $\dot{n}_1 = \dot{n}_2$. It follows that $\dot{e}[x := \dot{n}_1] = \dot{e}[x := \dot{n}_2]$ and $\dot{e}'[x := \dot{n}_1] = \dot{e}'[x := \dot{n}_2]$ and does not use columns or variables not at or below ℓ_o. Since $T_1 \dot{\sim}_{\delta,\ell_o} T_2$ it follows that corresponding records

are selected for modification, and that the same updates are computed for each. Thus, $\downarrow^{\delta}_{\ell_o}(T'_1) = \downarrow^{\delta}_{\ell_o}(T'_2)$.

$\mathrm{insert}\, i = \dot{e}\, \mathrm{into}\, \dot{t}$ Analogous to update, the result set indistinguishability is obvious. Assume that $\delta(table) \sqsubseteq \ell_o$, the opposite case is trivial. In general, by indistinguishability of the input states we have that the inputs have the same number of records, which means that the outputs have the same number of records, namely one more. If $\ell_\delta \not\sqsubseteq \ell_o$, then all columns are not at or below ℓ_o. In that case, $\downarrow^{\delta}_{\ell_o}(T'_1) = \downarrow^{\delta}_{\ell_o}(T'_2)$. Now assume the opposite. We have that the type of $\dot{e}$ is at or below $\delta(i)$ by premise. If $\delta(i) \not\sqsubseteq \ell_o$, then by the argument above the value is not visible in the projection, so that again $\downarrow^{\delta}_{\ell_o}(T'_1) = \downarrow^{\delta}_{\ell_o}(T'_2)$. If however $\delta(i) \sqsubseteq \ell_o$, then if $\dot{e}$ contains x, then $\ell_x \sqsubseteq \ell_o$, which means that $\dot{n}_1 = \dot{n}_2$. So always $\dot{e}[x := \dot{n}_1] = \dot{e}[x := \dot{n}_2]$. Thus, the value computed is the same for both runs. Thus the additional records in the projection are equivalent.

$\mathrm{delete\, from}\, \dot{t}\, \mathrm{where}\, \dot{e}$ Analogous to update and insert, the result set indistinguishability is obvious. Assume that $\delta(table) \sqsubseteq \ell_o$, the opposite case is trivial. Then if $\dot{x}$ appears in $\dot{e}$ it holds that $\dot{n}_1 = \dot{n}_2$. Thus, always $\dot{e}[x := \dot{n}_1] = \dot{e}[x := \dot{n}_2]$. Furthermore, since the inputs are indistinguishable at ℓ_o, it follows that corresponding records will be deleted. Thus, the outputs will be indistinguishable, too, since they will be restricted in the same way.

□

Chapter 7: INCREMENTAL LOADING

The content of this chapter is based on [38].

Language-based security traditionally enforces information flow control at compile-time. That means that a developer can be sure (as much as she trusts the compiler) that the source program as written is correct and handles confidential data according to a given policy. However, without further support, a user (different from the developer) is unable to establish if a program is secure:

- The user might not trust the developer. Even though the user trusts the correctness of the type system and the guarantees it brings, she can not establish that the developer actually ever used a secure language and compiler.
- If the user trusts the developer and the type system, it is not clear that the compiler used is free of bugs or malicious parts.
- If the user trusts the developer, the type system, and the compiler, the program might still be modified when stored or transmitted to the user, e.g., by a man-in-the-middle attack.

This problem can be mitigated by load-time checks performed by the user. The program loader and linker ensures that the program still fulfills all requirements for confidentiality. However, a full analysis requires a static analysis, since no security information is available anymore (security annotations are dropped by the compiler since CPUs do not support security operations). This is obviously too heavy-weight an approach, since users expect a timely application start.

Necula's proof-carrying code [69] annotates a (binary) program with a proof of correctness. In Necula's implementation, first-order predicate logic is used to formalize the program semantics and proofs of behavior. This technique can be adapted to annotate programs with enough security information to perform a light-weight verification. An example of this technique is [15], which describes a system for the analysis and verification of Java ME, the mobile edition of Java.

This technique has two drawbacks. First, it is based on Java Bytecode. It is well-known that Java Bytecode is not a good intermediate code representation: its stack-based nature makes it hard

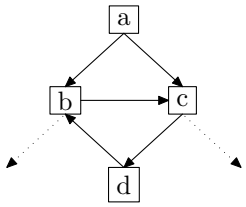


Figure 7.1: Example Irreducible Control Flow Graph

to verify, analyze and optimize.

Second, Java Bytecode is a quite general code format that allows to represent irreducible control flows, that is, it allows general jumps and branches to almost any point in the code. For example, the control flow graph in Figure 7.1 is encodable as Java Bytecode: The generality of irreducible control flow graphs leads to complex flows of information. In the above case, There is a flow from both `b` and `c` to `d`, where `b` and `c` are in branches separated by `a`. Without knowledge about both branches, the information flows will be incomplete and the program may leak data. Thus, for irreducible CFGs, all the code needs to be available for analysis, which means that information flow control is inherently a *whole-program analysis*. In the case of [15], this establishes itself through SOAP: *S*afe *O*ver*A*pproximation *P*roperty. All annotated information has to have the SOA property, which implicitly requires the full control flow graph.

Over the last decade, the Internet has become a premier distribution channel for both media content and software applications. While the available bandwidth has multiplied, content sizes also increased. For media content, streaming techniques have been developed that allow, for example, a video to be played before it is fully downloaded. For applications, however, such techniques are not commonly in use. Thus, there is still a significant time lag between the start of the download/installation and the start of the application.

For Java, [90] and [52] investigated a client-server architecture that streams code blocks. The server pre-links a program and partitions it into blocks for transmission, which are requested on demand by the client. [90] investigated different granularities for the blocks on a range of microbenchmarks. Method-level partitioning, which transfers whole methods on the first invocation, already leads to significant space savings. However, for maximum savings, the method body it-

self needs to be split up: [52] shows that on average only about half of the instructions of loaded classes is ever executed. For that, the server breaks up the method code into basic blocks (a straight sequence of instructions with a single entry and a single exit) and only transfers the entry block of a method on program start. Referenced (e.g., through jumps), but not loaded basic blocks are requested from the server when execution reaches that code location.

Note that [15] mentioned above cannot be applied to this architecture. The incremental code loading necessarily prohibits a whole-program analysis establishing safe over-approximative properties. Thus, incremental code loading for application streaming and information flow control seem to be mutually exclusive. We perceive this situation as a problem, since historical information suggests that it will be a while (if ever) until bandwidth will catch up with rising application sizes to achieve imperceptible download times.

Our solution is a novel intermediate code representation that allows for incremental loading *and* program verification. As mentioned before, irreducible information flow implies the necessity of a whole-program analysis. Thus we restrict our code format to only allow reducible control flows. Note that for our case, this actually does not represent a significant restriction. The Java ecosystem has an interesting semantic gap: while Java Bytecode allows a general `goto` and thus irreducible control flows, the source language Java is structured and only has reducible control structures. Thus, if we start from Java source code, our intermediate code representation is able to handle all legal Java code. Furthermore, the standard Java compiler creates a straightforward translation of Java to Java Bytecode that does not include irreducibility. Thus, all class files created solely by the standard compiler can also be re-translated to our format.

Our intermediate representation is centered around the concept of regions (e.g., [3]). All reducible control flow graphs can be structured through regions having a single entry, and the CFG can be collapsed through certain rules to a single region. Our code format structures the code more than basic regions, to transfer more information about the code. Similar to regions, our structures allow nesting for composition to create all reducible CFGs. Thus, the name of our code format is Nested Control Regions (NCR).

The difference between Java control structures and nested control regions is the level of abstraction. This is based on the idea that a small set of constructs can be used to create a whole CFG through nesting and combination, emulating higher-level control structures. We identified seven patterns of control flow that are necessary for Java programs. These seven patterns establish our NCR types, and the number and types of possible NCRs that can be placed in *compartments* (i.e., nested inside), as well as how the control flow connects them.

The advantage of this representation is the imposed structuring on low-level code. Whereas high-level languages impose a similar structure on source code, they usually highly restrict the control flow. Low-level approaches like Java Bytecode, on the other hand, impose neither structuring nor restrictions, making analysis a more complex and expensive task.

For example, a main part in load-time verification is the check of well-formed references. A reference (here to mean variable or register) can only be used after it is defined. This is formalized through the dominance relation: a program point dominates another, if the first precedes the latter on all paths through the program. Dominance thus is an important program information needed at load-time. Several algorithms with varying runtime complexity exists, with optimal algorithms, while being linear time, needing multiple passes and very complex data structures.

Our structuring allows us to use a single-pass, linear-time algorithm, instead. This is possible because we can approximate the dominance relation on a purely syntactical basis, and also allow fast retrieval of dominating parents. Note that the single pass is also important for incremental loading.

For information flow control, we layer the JIF type system over the basic Java-inspired type system. All standard label types of JFlow (the confidentiality part of JIF), e.g., policy labels, parameter labels and so on, are supported. Our loader only needs to verify the compliance with the security policies - all variables with non-default labels (i.e., labels that cannot be derived from input operands) have to be annotated in the serialized form. A program is safe then, if a traversal of the program yields no errors in computing the labels, that is, all annotated labels are at least as restrictive as the computed ones. This corresponds to just one iteration on an already existing

$$
\begin{array}{rcl}
e & ::= & x \mid n \mid e + e \mid e - e \mid e = e \mid e < e \\
C, D & ::= & \mathbf{skip} \mid \mathbf{load}^I_{pc} D \mid x := e \mid C_1; C_2 \mid \\
 & & \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 \mid \mathbf{while}\ e\ \mathbf{do}\ D
\end{array}
$$

Figure 7.2: Syntax

fixpoint of the dataflow algorithm that can infer security labels. The NCR structure helps in keeping this a single-pass linear time algorithm, since the encoding enforces that all relevant information for verification of a certain program location is already computed when verification arrives at that point.

The following section will formalize our incremental loading and verification approach. Section 7.2 describes Nested Control Regions in detail. Finally, Section 7.3 describes information flow control over NCR.

7.1 Formalization

This section gives a formal justification of our approach. We present a simple, imperative language with delayed loading. The language is intentionally kept simple to help the presentation. The abstraction is a valid foundation, since code loading is confined to the currently executing method. Thus prior work on extending imperative languages with procedures (e.g., [89]) and objects (e.g., [66, 13, 15]) are applicable to enforce noninterference for incrementally loaded Java.

7.1.1 Definitions

We base our work on the simple imperative language presented in [89]. The language has variables (ranged over by x), literals n out of $\mathbb{N}$ and expressions over those. The expressions and commands of the language are given by the syntax shown in Figure 7.2.

The sole new statement is $\mathbf{load}^I_{pc} D$. Here I defines a label for the code to be loaded and is assumed to be unique for each $\mathbf{load}$ statement in a program. The security label pc is an upper bound for the security that the statement is executed under. This corresponds to the program counter label, e.g., in [66]. It is a static annotation to simplify the semantics. Finally, the statement

D describes the fallback code for the case that verification of the loaded code fails.

We use the simple security lattice given by $(\{L, H\}, \sqsubseteq)$ with the usual semantics: L is a public value, while H is confidential; $L \sqsubseteq L$, $L \sqsubseteq H$ and $H \sqsubseteq H$. A typing environment Γ maps variables to security values.

A natural semantics gives meaning to our language. The advantage of a natural semantics lies in the easier exposition of the workings of the language: in a smallstep semantics, multiple **load** commands with the same loading label might be present in the current program: this would happen in the case a loop is expanded and the body contains a **load**. Correctly replacing all such commands requires extra work, e.g., non-standard handling of sequence commands and some rewriting semantics. This is a technicality we would like to avoid here.

A state (ranged over by μ, ν) is a finite function from variables to values. We define an equivalence relation $\sim_\Gamma$ on states such that two states are equivalent if their low variables (according to Γ) are equivalent: $\mu \sim_\Gamma \nu \iff dom(\Gamma) \subseteq dom(\mu) \wedge dom(\Gamma) \subseteq dom(\nu) \wedge \forall (x = L) \in \Gamma.\mu(x) = \nu(x)$. This formalizes the observable differences for low attackers. $\mu(e)$ describes the computation of e with the given variable values from state μ. The semantics for expression evaluation is standard and elided.We use the syntax $\mu[x := n]$ for the update of a state.

The semantics is formally defined as $(\mu, C) \rightsquigarrow_\Gamma (\mu', C')$ with the following meaning: given state μ, the execution of C changes the state to μ'. Because of placeholders, the program code might be changed. This is captured by C'. Note that this is different from a smallstep semantics, where C' is the continuation to be executed next. The relation is parameterized by the variable typing, which is necessary for **load**. The $\rightsquigarrow_\Gamma$ relation is given in Figure 7.3. Note that the semantics refers to the typing to ensure the correctness of loaded code. In practical systems, this could be implemented as type checking over type annotations.

The type system is an extension of [89]. Judgements are of the form $\Gamma \vdash exp : \tau$ and $\Gamma \vdash C : \tau\ \mathrm{cmd}$, where τ is a security level. The former judgement describes up to which confidentiality the expression reads variables. Variables are typed according to the variable typing. Literals are public. The result of an expression is the least upper bound of the sub-expressions.

$$[R_{SKIP}] \quad (\mu, \mathbf{skip}) \rightsquigarrow_\Gamma (\mu, \mathbf{skip})$$

$$[R_{ASS}] \quad (\mu, x := e) \rightsquigarrow_\Gamma (\mu[x := \mu(e)], x := e)$$

$$[R_{SEQ}] \quad \frac{(\mu, C_1) \rightsquigarrow_\Gamma (\mu', C_1') \quad (\mu', C_2) \rightsquigarrow_\Gamma (\mu'', C_2')}{(\mu, C_1; C_2) \rightsquigarrow_\Gamma (\mu'', C_1'; C_2')}$$

$$[R_{IFT}] \quad \frac{\mu(e) = 1 \qquad (\mu, C_1) \rightsquigarrow_\Gamma (\mu', C_1')}{(\mu, \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2) \rightsquigarrow_\Gamma (\mu', \mathbf{if}\ e\ \mathbf{then}\ C_1'\ \mathbf{else}\ C_2)}$$

$$[R_{IFF}] \quad \frac{\mu(e) \neq 1 \qquad (\mu, C_2) \rightsquigarrow_\Gamma (\mu', C_2')}{(\mu, \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2) \rightsquigarrow_\Gamma (\mu', \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2')}$$

$$[R_{WIT}] \quad \frac{\mu(e) = 1 \qquad (\mu, D) \rightsquigarrow_\Gamma (\mu', D') \quad (\mu', \mathbf{while}\ e\ \mathbf{do}\ D') \rightsquigarrow_\Gamma (\mu'', C')}{(\mu, \mathbf{while}\ e\ \mathbf{do}\ D) \rightsquigarrow_\Gamma (\mu'', C')}$$

$$[R_{WIF}] \quad \frac{\mu(e) \neq 1}{(\mu, \mathbf{while}\ e\ \mathbf{do}\ D) \rightsquigarrow_\Gamma (\mu, \mathbf{while}\ e\ \mathbf{do}\ D)}$$

$$[R_{LOT}] \quad \frac{(C = load(I)) \quad \Gamma \vdash C : pc\ \mathrm{cmd} \quad (\mu, C) \rightsquigarrow_\Gamma (\mu', C')}{(\mu, \mathbf{load}^I_{pc} D) \rightsquigarrow_\Gamma (\mu', C')}$$

$$[R_{LOF}] \quad \frac{(C = load(I)) \quad \Gamma \nvdash C : pc\ \mathrm{cmd} \quad (\mu, D) \rightsquigarrow_\Gamma (\mu', D')}{(\mu, \mathbf{load}^I_{pc} D) \rightsquigarrow_\Gamma (\mu', D')}$$

Figure 7.3: Semantics

$$[T_{VAR,INT}] \quad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \Gamma \vdash n : L$$

$$[T_{EXP}] \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \otimes e_2 : \tau_1 \sqcup \tau_2}$$

$$[T_{ASS,SKIP}] \quad \frac{\Gamma(x) = \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash x := e : \tau \text{ cmd}} \quad \Gamma \vdash \mathbf{skip} : H \text{ cmd}$$

$$[T_{IF}] \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash C_1 : \tau \text{ cmd} \quad \Gamma \vdash C_2 : \tau \text{ cmd}}{\Gamma \vdash \mathbf{if}\ e\ \mathbf{then}\ C_1\ \mathbf{else}\ C_2 : \tau \text{ cmd}}$$

$$[T_{WHILE}] \quad \frac{\Gamma \vdash e : \tau \quad \Gamma \vdash C : \tau \text{ cmd}}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ C : \tau \text{ cmd}}$$

$$[T_{SEQ}] \quad \frac{\Gamma \vdash C_1 : \tau \text{ cmd} \quad \Gamma \vdash C_2 : \tau \text{ cmd}}{\Gamma \vdash C_1; C_2 : \tau \text{ cmd}}$$

$$[T_{LOAD}] \quad \frac{\Gamma \vdash C : \tau \text{ cmd} \quad \tau \sqsubseteq pc}{\Gamma \vdash \mathbf{load}^I_{pc} C : \tau \text{ cmd}}$$

$$[T_{SUB1,2}] \quad \frac{\Gamma \vdash e : \tau_1 \;\; \tau_1 \sqsubseteq \tau_2}{\Gamma \vdash e : \tau_2} \quad \frac{\Gamma \vdash C : \tau_1 \text{ cmd} \;\; \tau_2 \sqsubseteq \tau_1}{\Gamma \vdash C : \tau_2 \text{ cmd}}$$

Figure 7.4: Type System

The latter judgement describes the lower bound of variables which a command may modify. Most rules are standard. We show the type system in Figure 7.4.

The important rule is T_{LOAD}. Statically, the type system ensures that the fallback statement C is correctly typed. Furthermore, for later loading, the semantics of the load command ensures that the pc annotation is at least as restrictive as the typing.

7.1.2 Noninterference proof

Our proofs follow [89] and [31] and are an instance of progress and reduction lemmas.

Lemma 97 (Subject Reduction)**.** *If* $\Gamma \vdash C : \tau \text{ cmd}$ *and* $(\mu, C) \rightsquigarrow_\Gamma (\mu', C')$*, then* $\Gamma \vdash C' : \tau \text{ cmd}$.

Proof. By induction on $(\mu, C) \rightsquigarrow_\Gamma (\mu', C')$. We show only the cases where C is a load statement,

which follows from the typing on such statements. The other cases are standard.

Case R_{LOT}**:** Then $C = \mathbf{load}^I_{pc} D$, and the typing derivation ends with T_{LOAD} followed by an application of subsumption. This is a standard simplification, as it can be easily shown that whenever a load statement is typable, there exists a typing derivation of said form, so this is without loss of generality.

$$\dfrac{\dfrac{\Gamma \vdash D : \tau' \text{ cmd} \quad \tau' \sqsubseteq pc}{\Gamma \vdash \mathbf{load}^I_{pc} D : \tau' \text{ cmd}} \; T_{LOAD} \quad \tau \sqsubseteq \tau'}{\Gamma \vdash \mathbf{load}^I_{pc} D : \tau \text{ cmd}} \; T_{SUB2}$$

By premise of R_{LOT}, we know that $\Gamma \vdash load(I) : pc \text{ cmd}$. By premise of T_{LOAD}, $\tau' \sqsubseteq pc$, and by T_{SUB2} $\tau \sqsubseteq \tau'$. With subsumption by T_{SUB2}, we get $\Gamma \vdash load(I) : \tau \text{ cmd}$. Finally by the inductive hypothesis of the last premise of R_{LOT}, we find $\Gamma \vdash C' : \tau \text{ cmd}$.

Case R_{LOF}**:** Then $C = \mathbf{load}^I_{pc} D$, and the typing derivation ends with T_{LOAD} followed by an application of subsumption, as above. Thus by the premise of T_{LOAD}, $\Gamma \vdash D : \tau' \text{ cmd}$. By an application of subsumption, we get $\Gamma \vdash D : \tau \text{ cmd}$. By the inductive hypothesis of the last premise of R_{LOF}, we have $\Gamma \vdash D' : \tau \text{ cmd}$.

□

Lemma 98 (Simple Security)**.** *If* $\Gamma \vdash e : L$ *and* $\mu \sim_\Gamma \nu$*, then* $\mu(e) = \nu(e)$.

Proof. The proof proceeds by induction on the derivation of $\Gamma \vdash e : L$.

Case x**:** Then the derivation ends with T_{VAR} followed by an application of subsumption for the reflexive case. Thus, by the premise of T_{VAR}, $\Gamma(x) = L$, and thus by definition of $\sim_\Gamma$ $\mu(x) = \nu(x)$.

Case n**:** Then trivially $\mu(n) = \nu(n) = n$.

Case $e_1 \otimes e_2$. Then the derivation ends with T_{EXP} followed by an application of subsumption for the reflexive case. Thus, $\Gamma \vdash e_1 : L$ and $\Gamma \vdash e_2 : L$, and by induction $\mu(e_1) = \nu(e_1)$ and

$\mu(e_2) = \nu(e_2)$. For $\otimes \in \{+, -\}$, $\mu(e_1 \otimes e_2) = \mu(e_1) \otimes \mu(e_2) = \nu(e_1) \otimes \nu(e_2) = \nu(e_1 \otimes e_2)$. For $\otimes =$"=", we have either $\mu(e_1) = \mu(e_2)$ and thus $\nu(e_1) = \nu(e_2)$, so $\mu(e_1 = e_2) = 1 = \nu(e_1 = e_2)$, or $\mu(e_1) \neq \mu(e_2)$ and thus $\nu(e_1) \neq \nu(e_2)$, so $\mu(e_1 = e_2) = 0 = \nu(e_1 = e_2)$. The case for "$<$" is similar.

□

Lemma 99 (Confinement)**.** *If* $\vdash C : H \text{ cmd}$ *and* $(\mu, C) \rightsquigarrow_\Gamma (\mu', C')$*, then* $\mu \sim_\Gamma \mu'$.

Proof. By induction on the derivation of $(\mu, C) \rightsquigarrow_\Gamma (\mu', C')$, with the help of Lemma 97. Most cases are standard, we show the cases for R_{WIT} and $\mathbf{load}$.

Case R_{WIT}**:** Then $C = \mathbf{while}\ e\ \mathbf{do}\ D$, and the typing derivation ends with T_{WHILE} followed by an application of subsumption:

$$\dfrac{\dfrac{\Gamma \vdash e : \tau' \quad \Gamma \vdash D : \tau' \text{ cmd}}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ D : \tau' \text{ cmd} \quad H \sqsubseteq \tau'}\ T_{WHILE}}{\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ D : H \text{ cmd}}\ T_{SUB2}$$

Then $\tau' = H$, which means $\Gamma \vdash D : H \text{ cmd}$. Now $(\mu, D) \rightsquigarrow_\Gamma (\nu, D')$ by R_{WIT}, with $\mu \sim_\Gamma \nu$ by inductive hypothesis. Subject reduction gives $\Gamma \vdash D' : H \text{ cmd}$. Now by T_{WHILE} it follows that $\Gamma \vdash \mathbf{while}\ e\ \mathbf{do}\ D' : H \text{ cmd}$, and thus by inductive hypothesis $\nu \sim_\Gamma \mu'$, and thus $\mu \sim_\Gamma \mu'$.

Case R_{LOT}**:** Then $C = \mathbf{load}^I_{pc} D$, and the typing derivation ends with T_{LOAD} followed by an application of subsumption:

$$\dfrac{\dfrac{\Gamma \vdash D : \tau' \text{ cmd} \quad \tau' \sqsubseteq pc}{\Gamma \vdash \mathbf{load}^I_{pc} D : \tau' \text{ cmd} \quad H \sqsubseteq \tau'}\ T_{LOAD}}{\Gamma \vdash \mathbf{load}^I_{pc} D : H \text{ cmd}}\ T_{SUB2}$$

Then $\tau' = pc = H$, thus $\Gamma \vdash load(I) : H \text{ cmd}$ by premise of R_{LOT}. Now by inductive hypothesis $\mu \sim_\Gamma \mu'$.

Case R_{LOF}**:** Then $C = \mathbf{load}^I_{pc} D$, and the typing derivation ends with T_{LOAD} followed by an application of subsumption, as above. From the premise we get $\Gamma \vdash D : H\ \mathrm{cmd}$, thus by inductive hypothesis $\mu \sim_\Gamma \mu'$.

□

Finally, the noninterference theorem states that if two computations start in equivalent states and terminate, they end in equivalent states.

Theorem 100 (Noninterference)**.** *If* $\vdash C : \tau\ \mathrm{cmd}$*,* $\mu \sim_\Gamma \nu$*,* $(\mu, C) \rightsquigarrow_\Gamma (\mu', C')$*,* $(\nu, C) \rightsquigarrow_\Gamma (\nu', C'')$*, then* $\mu' \sim_\Gamma \nu'$*.*

Proof. By induction on $(\mu, C) \rightsquigarrow_\Gamma (\mu', C')$, using the previous lemmas. We show the cases for R_{WIT} and $\mathbf{load}$.

Case R_{WIT}**:** Then $C = \mathbf{while}\ e\ \mathbf{do}\ D$, and the typing derivation ends with T_{WHILE} followed by an application of subsumption. By R_{WIT}, $\mu(e) = 1$.

If $\Gamma \vdash exp : L$, then by $\mu \sim_\Gamma \nu$ and Simple Security $\mu(e) = \nu(e) = 1$. Thus, R_{WIT} is also the last rule in $(\nu, C) \rightsquigarrow_\Gamma (\nu', C'')$. Now by induction over the premises of R_{WIT} and transitivity of $\sim_\Gamma$, $\mu' \sim_\Gamma \nu'$.

If $\Gamma \nvdash exp : L$, then by $T_{WHILE}\ \tau \not\equiv L$, thus by confinement $\mu \sim_\Gamma \mu'$ and $\nu \sim_\Gamma \nu'$, thus $\mu' \sim_\Gamma \nu'$.

Case R_{LOT}**:** Then $C = \mathbf{load}^I_{pc} D$, and the typing derivation ends with T_{LOAD} followed by an application of subsumption. By the premise of R_{LOT} the loaded code can be typed, thus by the inductive premise we have $\mu \sim_\Gamma \nu$.

Case R_{LOF}**:** Then $C = \mathbf{load}^I_{pc} D$, and the typing derivation ends with T_{LOAD} followed by an application of subsumption. Since D is well-typed by the premise of T_{LOAD}, we can apply the inductive hypothesis and get $\mu \sim_\Gamma \nu$.

□

7.1.3 Evaluation instead of Loading

The above system uses placeholders in the form of **load** in a one-shot manner: when a placeholder needs to be executed, the corresponding code is loaded and type-checked. If the check succeeds, the placeholder is replaced with the loaded code. Otherwise, the statically checked fallback code replaces the placeholder.

Instead of this replacement, the semantics could be changed to execute the loaded code, but retain the placeholder. This would correspond to `eval` statements in dynamic languages like Javascript. While typing of arbitrary code is complicated and would need type inference, the common case of using `eval` to load libraries from third-party providers can be covered. The provider needs to type its code just once, and can provide that general typing to the client as type annotations. The typing judgement in R_{LOX} will then degenerate to a typing verification.

7.2 Nested Control Regions

The previous section gave a formal proof for a simplified imperative language and thus forms the theoretical foundation for our work. In the following sections we describe the implementation of our practical system. This section gives an overview over the basic set of nested control regions, which form the basis of programs in our architecture, while Section 7.3 shows how to add information flow control.

As stated earlier, nested control regions were designed with two goals in mind. First, the IR should be simple and highly structured to increase verification efficiency and optimizability. In particular, NCR supports a single-pass linear-time verification algorithm so that the format can be used efficiently even on limited devices. Second, the semantics of the regions needs to ensure that nested regions can only effect other regions outside the nesting in a principled manner. We accomplish this by using low-level structured code.

The code structuring will retain the control information from the source languages, albeit in a reduced, lower-level form. Using a structured intermediate language has several advantages over

using the highly-structured source code directly. Source languages usually expose very complex control structures that do not correspond well to the features of current hardware architecture. It's the compiler's task to reduce this complexity. An IR can be a middle step in between, transfer code with already reduced complexity, thus decreasing the overhead at the client. Also, source languages usually expose a lot of syntactic sugar and corner cases, which complicate parsing and analysis, which is undesirable, especially in restricted environments.

7.2.1 Register-based IR

Prior experience with the benefits of a non-stack IR led us to adopt a register based IR. While traditional stack-based models like Java are known for their compactness and easy-to-write interpreters, prior research shows that register-based models can be competitive in those respects [4].Advanced techniques like verifiable annotations (or proof-carrying code in general) are harder to implement on stack-based code, and many runtime environments use Just-in-Time compilation, which itself usually uses multiple variants of register-based intermediate representations.

In NCR, registers are in SSA form. Each register is (statically) assigned to exactly once. When control flows join, φ functions are used to join the corresponding data flows. This construction can significantly reduce the complexity of many analysis and optimization algorithms. Furthermore, SSA-form and code structure can reduce the amount of annotations necessary in a verification systems [5].

7.2.2 Control Regions

Well-formed SSA-code uses registers as instruction parameters only when they are defined on all incoming paths that reach the instruction. The register is then said to "dominate" the instruction. A verifier for NCR code has to check this well-formedness property to ensure the integrity of a program. To ensure fast verification, dominator information should be easily accessible. There are complicated multi-pass algorithms that are able to extract the dominator tree from a whole control flow graph in linear time. We instead rely on structuring the code and its serialization in a way that

allows us to retrieve most dominator information easily in a single pass.

Our structuring is based on the idea that a small set of control constructs can be used to create a control flow graph (CFG) through nesting and combination. That is the reason our approach is called Nested Control Regions. There are several NCR types, each of which represents a certain pattern of control flow with a single entry and dictates the number and types of possible NCRs that can be placed in *compartments* (i.e., nested inside), as well as how the control flow connects them. The advantage of this representation is the imposed structuring on low-level code. Whereas high-level languages impose similar structure on source code, they usually highly restrict the control flow. Low-level approaches like Java bytecode, on the other hand, impose neither structuring nor restrictions, making analysis a more complex and expensive task. Our control structures are more general than those given by high-level languages and allow more complex constructs, while still retaining a structure in the code, allowing for example the easy extraction of dominator information and loop nesting. The control patterns we identified as necessary and translated to control regions are:

Block regions. A straight segment of instructions with a single entry. There is one normal successor, which is nested in the region's single compartment.

Exit regions. A region defining two compartments: a *body* and an *exit*. The code within the body is allowed to jump to the exit. This corresponds to forward jumps in reducible control flow graphs.

Restricted goto. A restricted goto implements the jump allowed by an enclosing region, e.g., an enclosing exit region.

Condition regions. A condition region has two compartments and a value slot. Depending on the (boolean) value, control flow is diverted to one of the two compartments. Note that for a full traditional *if* construct, an exit region is necessary to join the control flow afterwards.

Loop regions. A loop region defines the header of a simple loop and a *body* compartment. Re-

stricted gotos inside the body are allowed to jump to the header (i.e., the loop region itself), forming a looping structure. Note that conditional loops have to be constructed by nesting a condition region inside the loop body.

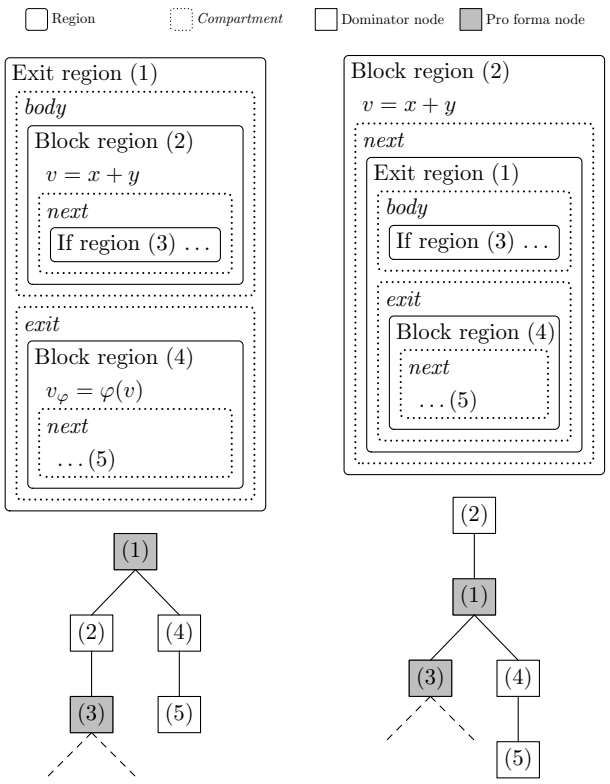
Exception and Finally regions. These region types provide structure related to exceptional control flow. Both types define two compartments: a *body* compartment, as well as a *catch* compartment for exception regions and a *finally* compartment for finally regions, respectively. The regions have the expected semantics.

The nesting of those regions allows us to easily compute a conservative approximation of the dominator relationship. Each region has a corresponding node in our dominator tree. Instructionless regions are associated with pro-forma nodes used to structure the tree. These nodes can be freely arranged, since no actual code is involved. Except for pro-forma nodes, a parent is known to always be a dominator of a nested child. But the reverse does not universally hold in our system: not all dominating nodes must be ancestors of the nodes they dominate. This can lead to an imprecise approximation (e.g., as in Figure 7.5), but for any reducible control flow graph, it is possible to create a control region nesting for which the dominator tree constructed in this manner will be precise.

Proof of Precise Approximation of Dominator Relation

This section sketches a proof that all reducible control flow graphs (CFGs) can be encoded with NCR regions such that the NCR approximation of dominance is precise. It relies on the definition of reducibility based on two transformations T_1 and T_2, which modify a control flow graph. T_1 removes self-edges, while T_2 folds a node which has only a single incoming edge. A graph is reducible if there is a sequence of transformations that change the CFG into a single node.

The proof is constructive: for a given reducible CFG we construct a nesting of NCRs that reflects the CFG. For simplicity, we assume that the out-degree in the CFG is at max two. The development can easily be extended to higher out-degrees. Furthermore, the resulting encoding is



(a) Imprecise (b) Precise

Figure 7.5: Precise and imprecise approximation of dominator relationship

not necessarily compact. In general, multiple regions represent a node in the CFG, such that the encoding may contain several regions that are empty and only necessary for structuring.

The construction works by reversing the transformation sequence that leads to reduction of the CFG. We begin with a single node encoding, and successively extend the current encoding. Each node is represented by a sequence of regions making up the body of the node, and a set of gotos for the out-going edges. The body is made up of loop regions for backward jumps, block regions for actual instructions, and exitable regions for forward jumps (the gotos of which will be in later nodes). The control flow is purely sequential and uses the straightforward nesting of the region types. The set of gotos models the out-going edges. This is not a valid NCR structure, since the regions are not nested correctly. However, depending on the number of edges, a post-processing step can synthesize the right control structure and embed the gotos: a single out-edge can be either represented by nesting the goto (if the target is a join node), or by embedding the target directly;

two out-edges mean a binary decision, which can be encoded with a conditional region, where the gotos are handled like the single-edge case, but embedded in the branches; no out-edges means the end of control flow, so that a return should be synthesized.

The construction moves by case analysis of the transformation. A T_1 transformation removed a self-edge. Thus, a loop is necessary in the NCR structure, which means another loop region is pushed into the body. A T_2 transformation collapsed a node with a single incoming edge. This means a new node has to be established. It is important to distinguish between two situations: the transformation changes a successor into a join node, or not (the join node may be created because a node edge and a parent edge were joined during collapsing). In case a join node is created, the NCR representation is marked to be a join. If it was not marked, the parent is the dominator and will receive an exitable region, with the join node being the exit compartment. No matter the creation of a join node, a new node representation is created and edges/gotos adjusted accordingly.

Semantic correctness of the construction is straightforward. Loops and exitable regions only serve as "landing pads" for incoming edges, while the block regions encode the actual code. The out-going gotos are created directly according to out-going edges of the CFG.

The proof that this construction retains the precise dominator relation is straightforward. It follows by induction over the reversed transformation sequence, and the definition of the dominator approximation as the transitive closure of the parent relation as given by the region nesting. Each step of the construction preserves the original dominator relation in the approximation, and thus after iterating over the transformation sequence, the original dominator relation, interspersed with pro-forma regions, is encoded correctly.

Irreducible Control Flows

While reducible control flow is acceptable when the source language enforces reducible programs, some cases may require the ability to encode irreducible control flow directly. For example, Java programs are compiled to Java Bytecode, which allows programs with irreducible control flow.

We can extend our set of region types with an irreducible region (IRR). This region type has an

arbitrary number of compartments. Control flow inside the IRR is free, i.e., gotos are allowed to transfer control to any compartment. However, the whole region enforces a single-entry discipline by using a distinguished head compartment.

This construction avoids the exponential complexity of transforming irreducible CFGs to reducible CFGs, while localizing the impact of the irreducibility. Of course, linear-time single-pass verification cannot be guaranteed anymore for irreducible parts of a program. Furthermore, incremental loading and verification requires IRRs to be loaded completely before being executed.

Structuring code through control regions not only helps with dominator tree retrieval. Nesting regions allow us to verify code in a modular fashion: no information from a compartment is necessary to verify basic information. This can be exploited to easily partition and break up code. Static analyses can take advantage of structural information since regions restrict the set of valid sub-control flows.

7.2.3 Placeholders

To include incremental loading, we add placeholder regions to the set of region types. Adopting a region instead of an instruction has the benefit of simpler replacement of the loaded code in the existing program. In the current implementation, simple numbers are used as handles in the communication with the server. When control would flow to a placeholder, execution is stopped and the server contacted. The placeholder will be replaced with the region received from the server, and finally the execution is resumed.

7.2.4 Instruction Set

The instruction set is a register-oriented version of Java Bytecode. All stack manipulation operations have been removed. We support both instructions on primitive types as well as object operations. As a simplification, we only include a unified allocation-and-initialization operation for objects, while Java bytecode separates the parts. This is however not significant in our setting: we start with an extended version of Java. Most basic Java compilers create object allocation and

initialization code in a principled manner that can be emulated by our single operation.

7.2.5 Prototype Implementation

We adapted the standard JIF compiler based on Polyglot [70] to emit NCR code. Translation of Java source code is done in a straightforward mapping of Java structures to nestings of NC regions. Most Java control structures correspond to a simple template. For a proper treatment of computing regions we refer to the literature (e.g., [3]).

To demonstrate the feasability of NCR, we created an interpreter able to load and run programs in NCR form. The interpreter is a Java program itself, and uses the facilities of the host JVM. For example, class stubs are created at runtime to easily represent objects with program code in NCR form. With this implementation, classes generated from NCR code can easily interact with (i.e., be called from) the Java environment. Furthermore, with the stubs, management of the objects and the Java call behaviour are moved to the virtual machine.

7.3 Information Flow Control with NCR

On top of the basic NCR representation we layered annotations for security typing. This was done as a mostly straightforward port of the JIF system: security information is encoded as a label from the Distributed Label Model [67]. Our current implementation is based on the older JFlow model, which only protects confidentiality. A future treatment of integrity should be straightforward, since those two are dual. Out of the complex set of label types in JFlow, we implemented the most important ones:

Class/Substitution/Reference labels. These labels are used for genericity with respect to class parameters. This topic is strongly related to generic data structures with parametricity over some type. In this context, a data structure should be generic over some confidentiality label. For example, a list data structure should be written only once for all levels of confidentiality the elements might have, and instantiated with the right labels when used.

Class labels capture the label parametricity of a class by defining named parameters that can be used as placeholders in the class' body. Reference labels stand for a reference to a class parameter in the class' body. Substitution labels are used to assign values to parameters in class labels.

In the original JIF, class labels are layered on top of Java types for classes. Thus, class labels implicitly contain the structure of the Java class they are based on. Since our JIF implementation is decoupled from Java, class labels have to explicitly define this information. Class labels are extended to be simple records of typed fields and functions. Instead of Java types, JIF labels are used.

As an example, take the following JIF code:

```
class [label P] A {
  B{Alice:Bob} b;
  C[P]{P} foo() {...}
}
```

This gets translated to the following class label:

```
classlabel A[P]:
  (classlabel B){Alice:Bob} b;
  (subst L=ref P in classlabel C[L])
    {ref P} foo();
```

To not interfere with loading and linking policies of possible other annotated type systems (e.g., late resolution in Java), our JIF uses named references instead of fully loaded class labels in a class label definition. Those references are only resolved on a by-need basis, e.g., when the field of a class is actually accessed.

Policy labels. Labels made up of a simple policy $p = o : r_1, \ldots$. A policy p restricts the access to a value to the readers r given in the policy. The policy itself is owned by o (i.e., o is the

restricting principal).

Join labels. These labels join together multiple labels. The resulting label is at least as restrictive as each component. Join labels are necessary for a precise analysis, since joining two labels does not usually result in a label of the same type. An example is a value under the influence of two simple policies of two different owners.

Parameter labels. Parameter labels are similar to a reference to another label. A parameter label $\{a\}$ has the some value as the label the variable a is annotated with.

Bottom and top label. The bottom label is the least restrictive label in the label lattice, whereas the top label is the most restrictive one.

In the JIF framework, general label constraints are derived from program statements and collected in a constraint systems. An iterative, data flow-like algorithm computes a fixpoint solution of the system, which is a conservative labeling of all occurring variables, in case the program is secure. This approach is also used to include label inference, which makes programming in JIF an easier task (label inference automatically computes the least restrictive label for an unannotated variable).

Our security-typed mobile code only needs to verify the compliance with the security policies. All variables with non-default labels (i.e., labels that cannot be derived from input operands) have to be annotated in the serialized form. A program is safe then, if a traversal of the program yields no errors in computing the labels, that is, all annotated labels are at least as restrictive as the computed ones. In that regard, our algorithm is one iteration of the data flow algorithm on an already existing fixed point. It is thus similar to other annotation schemes based on data flow properties, e.g., [44, 5, 25].

Since the NCR structure is rebuilt in a pre-order traversal of the dominator tree, all information needed for verifying the security labels is available at the time a node is processed. This allows us to keep the single pass structure of the general loading and verification. On the assumption that

operations on labels take a constant time[1], we can also keep the linear time bound.

Important points in the control flow that need to be annotated are loops. This is necessary to avoid the looping in the iterative data flow algorithm until a fixed point is reached. In the context of security-typed code, the changing entity over loop iterations is the PC. The original JIF implementation introduces a new, blank label to compute the information that is transferred over loop iterations. This label is then filled by the constraint solver. Our fixed-point algorithm expects this information annotated at the loop head. The original PC entering the loop will be compared with and replaced by the annotated label, which has to be at least as restrictive. If a jump back to the loop header is encountered in the loop body, the current PC and the annotated loop entry PC are compared. The program is valid if the current PC is at most as restrictive as the loop entry PC (see JFlow).

Other join points, like joins after branches, do not need to be annotated. In conformance with JIF, the current PC will be computed as the label join of all PCs at the end of the predecessors.

A special building block used with NCR are single-exit (SE) regions. These regions contain a sub-flow that can only be entered through the entry of the SE region as well as exited through a special compartment, the exit compartment. Note that this is slightly different from exit regions: in a nested exit region, it is permissible to jump to the outer exit with a restricted goto, whereas single-exit regions forbid such jumps. This simplifies information flow insofar as any PC information gathered in the nested sub-flow is lost at the exit of the region, since all paths through the control flow graph will go through that node. Thus, the PC at the end of the region is the same as the PC at the entry. Note that values, however, might have labels dependent on the internal control flow.

Most instructions produce values, which now have a labeled type. In most instances, annotating the value is not necessary. The label produced by joining all the labels of the input operands with the current PC, which captures all the restraints of the input data, will be as restrictive as needed. Any valid annotation must be at least as restrictive as that label.

[1]This assumption is intuitively valid because only a bounded number of "atoms" can be introduced through method parameters etc. Thus operating on a finite set of elements is possible in constant time.

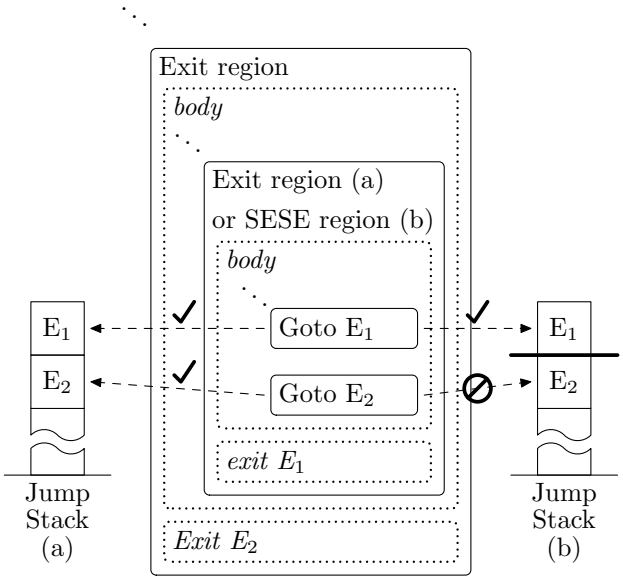


Figure 7.6: SE region implementation

φ functions do not need to be handled in any special way. The different parameters for different control flows are treated just as normal parameters. The resulting label will be the join of all input labels and the PC of the exit region, being at least as restrictive as each input label and including the control flow information.

Annotating the values of SSA instructions has advantages over the original JIF approach. In JIF, labels annotate source variables. A variable thus has the same label for its lifetime (since the annotated component is the type of the variable). SSA mandates a fresh variable every time a value is produced, splitting up the scope of a source variable. This has the potential to increase the granularity of label annotations, since a reused variable in the source code (like helper variables) is split up into multiple scopes and can have different labels, possibly decreasing the number of rejected programs. In a sense, the type system annotations become flow-sensitive. Of course, since our prototype translates code produced by the standard JIF compiler, this advantage is purely theoretical until we implement our own independent JIF-to-NCR compiler.

Another benefit arises from NCR's use of low level of instructions. In most cases, complex source code statements are broken up into several simpler SSA instructions. This greatly simplifies the constraints placed on instructions.

Exceptions are handled with path labels as introduced in JIF. To ensure linear-time verification, we currently need to restrict methods to declare escaping exceptions precisely (i.e., it is not allowed to throw an exception subtype). This is common; e.g., [15] requires a mapping of exception types to exception handlers to maintain lightweight verification.

We integrate incremental loading along the lines of Section 4.3. Placeholder regions contain both a reference handle and fallback code. The fallback code is statically checked and valid at the point of the placeholder. During execution, when the placeholder is reached, the referenced code is requested from the server and type-checked. If the check succeeds, the downloaded code replaces the placeholder. Otherwise, the fallback code is used. This way, no matter the outcome of the type-check, execution can continue.

Further issues arise in an actual system. First and foremost, the verification environment has to be extended so that it includes the server. The client will request code blocks depending on the current execution. This will, of course, transfer information to the server, as in the following example:

```
if (condition)
  { /* Block 1 */ }
else
  { /* Block 2 */ }
```

The server will be able to deduce the condition value based on which block is requested. For the verification to succeed, the server has to be trusted with the information. Thus, incremental loading has to be rejected in environments where the server is not trusted enough.

For proper treatment of integrity, it is important that the integrity of the code be considered. Low-integrity code should not be able to handle high-integrity data or take actions that require high integrity such as declassification. The integrity of the code can be established through trust of the server or through code signing.

Chapter 8: RELATED WORK

This chapter surveys related work on the topics covered in this dissertation. Each of the following sections discusses work related to one chapter of this thesis.

8.1 Framework

8.1.1 Language Composition

Basic type-safety of composition has been investigated, e.g., [20] studied an SQL-like extension to $C^{\#}$. However, the query language was fully incorporated into the host, which seems infeasible for the sheer amount and expressivity of languages considered for embedding in practice. Compositionality of noninterference has been studied in different contexts (see [64, 77, 62] for overviews). The goal of these works was to find circumstances under which composing secure program fragments of *one* formal system yields a secure result. In contrast, our fragments are derived from *different* security-typed languages (here While and SQL), and we base our compositionality result around the notion of security completeness.

8.1.2 System Composition

There are several related approaches to secure a complex system. Li and Zdancewic [54] describe a web programming language extended with an interface to a relational database (note only simple select statements), where the language and interface are security typed. But this means that the storage side needs to be fully trusted. Arrows and monads (e.g.,[55, 86, 76] and references therein) can be used to isolate and control information flows in a library fashion. We contend that this approach has problems similar to those of Li and Zdancewic's approach. We note that a library approach to embedded languages restricts expressivity and conciseness to that of the host language, as library exposes functionality to the host language through an interface describable in the language. Thus, concepts of the embedded language need to be mapped to the host language,

potentially losing all benefits of domain specificity and conciseness.

Fabric [61] extends JIF to a secure distributed system. The set of nodes in this system is partitioned into worker and storage nodes, so that managing storage is an integrated part of the language/system. We believe that separation of concerns is important and our approach allows a modular proof of the safety of the whole system, composed from smaller fragments. One may take note that JIF, the foundation of Fabric, has still no formal guarantees of safety. Also note that, persistent storage is but one use of an embedded language, and it seems unrealistic to expect one language to excel in all domains.

8.2 Security Completeness

To the best of our knowledge, this is the first work to make use of completeness in the context of security-typed languages. Kahrs [49, 50] studied completeness for basic type systems, where the question is if all computable functions that are "well-going" can be typed. Kahrs uses transitions systems, whereas our goal is to permit easy adoption of existing languages.

Traditional work in security-typed languages attempts to broaden the permissibility of the type system, that is, accept more programs as typed and thus secure. For an older survey we refer to [78]. Some more recent work is, for example, [31, 14, 48]. Our work is orthogonal to such efforts. We show that, under certain constraints, there are always programs that compute a given noninterferent function.

Our approach to prove a language security-complete is related to Secure Multi-Execution (SME) [33]. SME can be seen as the dynamic counter part to our static program transformation. Instances of the program are run in parallel. As the noninterference notion is channel-based (vs our input-output-based one), SME also needs additional runtime infrastructure to clone and dispatch inputs from input channels and merge or discard outputs on output channels.

Formally, SME extends a standard language with specific concurrency support and concrete runtime semantics that enforce the separation of the runs. The result is that in case the original program was noninterfering, the output of secure multi-execution will be equivalent to the orig-

inal program. If the original was not noninterfering, the program's semantics are *not* preserved. However, even in that case the result is some noninterfering program.

For a comparison, first note that secure multi-execution has a slightly different goal than security completeness. Security completeness ensures that an *equivalent* typable program exists for noninterfering computations. Secure multi-execution ensures that a program run *is* noninterfering, no matter if the input program is noninterfering itself. While this means that some technical limitations are shared, e.g., the need for an ordered lattice in case of termination-sensitive noninterference, other properties and problems are unrelated: Security completeness must ensure equivalence. In the case of termination-insensitive noninterference this implied additional work, as we must also ensure equivalent termination behavior. SME only ensures (termination-insensitive) noninterference and thus does not need this extra step. Security completeness is based on input-output noninterference, and input-output equivalence. The order of computations is thus a minor issue - only termination-sensitive noninterference may restrict the order of computation in some type systems. SME is based on channel noninterference. The semantics of the SME extended language must thus include a scheduler that is level-aware. The seminal paper [33] actually contains leaks in case that the security lattice is not totally ordered. While this issue was remedied [51], the solution implies even more runtime overhead than the original secure multi-execution.

One important difference is that SME is envisioned as an implementation technique whereas we use level-separated simulation as a theoretical device to prove security completeness. Secure multi-execution, as a practical method, has prototype implementations (e.g., [30]). Security completeness is a theoretical property of security-typed languages. While our proof approach can be used to derive typed programs, those programs will in general not be practical.

8.3 Extensions

8.3.1 Nondeterminism

To the best of our knowledge, we are the first to argue about security of nondeterministic programs by deterministic simulation.

Our approach to nondeterministic languages is similar to prophecy variables [2, 63]. That work proposes to prove the correctness of an implementation with respect to a specification by showing that the implementation is a refinement. While an implementation may only expose a subset of the behaviors of the specification, it likely includes more detailed/concrete data structures and so on. To match specification and implementation runs, this mismatch has to be mapped. History variables add information about past decisions and values of data that are not in the specification. Prophecy variables are the dual for the future.

In our case, the determinism variable i in Definition 72 is a prophecy variable for the specific choices of nondeterministic cases in the semantics.

8.3.2 Declassification

Declassification is an active research topic. For a recent survey and categorization we refer to [80, 81]. We are not aware of any work in composing programs with declassification.

We adapted two declassification proposals that are enforcable by a security type system. Delimited release [79] specifies *what* can be declassified. It defines so-called escape hatches that are allowed to be declassified. The proposed security type system ensures that only escape hatches can be declassified, and only in a consistent way, by tracking changes as effects. The model has been extended to also cover the *where* axis of declassification [9].

Robust declassification [92] specifies *who* is allowed to declassify. So-called attacks, that is, modifications of the system under consideration, cannot change the information release in a robust system. Attacks here are modifications that, by themselves, form a secure system. An attacker is assumed to be unable to circumvent basic security properties. This view is seen as practical insofar

it applies to the domain of dynamically loaded code, which is not executed until a basic check of the loaded code succeeds.

In [68] a security type system is proposed that can enforce a language-based notion of robustness. The type system tracks both confidentiality and integrity constraints. Attacks are confined to low-integrity data, while declassification can only be performed in a high-integrity context.

We straightforwardly adapt the effect tracking of [79] to support delimited release. In contrast, prior formulations of robust declassification [92] and [68] do not lend themselves to language composition. The former is too weakly constrained, such that a significant number of further requirements are necessary to show a general compositionality result. The latter is too confined to the specific language upon which robust declassification is enforced. We distill a *relaxed* version from [68] we call step-wise robustness. Step-wise robustness lies between [92] and [68], and forms an implication chain, that is, we show that a typed program of [68] is step-wise robust, and a step-wise robust system is robust with respect to the definition of [92]. Finally, our step-wise robustness is constrained enough to immediately reason about composition.

8.4 Languages

8.4.1 Object-oriented calculus

Several treatments of information flow in object-oriented languages have been presented, e.g., [66, 13, 75, 85]. These are all class-based, a quite different setting from ours, with its own challenges and simplifications.

Our proofs were inspired by work of Barthe and Serpette [16]. To the best of our knowledge, this is the only other information flow work in foundational object calculi. The work presents an annotated version of Abadi and Cardelli's object calculus and its first-order type system. Our work differs in that it allows object extension and does not rely on Abadi & Cardelli as an underlying type system, and thus execution might produce method-not-found errors.

Our work is related to Askarov and Sabelfeld [10], who describe an extension of JIF that allows

to designate exceptions that cannot be caught and lead to termination. While on the surface similar, the goals and features are quite different. Our work considers object-oriented programming based around objects and allows adding methods to objects, a feature widely used in dynamic languages, while Askarov and Sabelfeld's extension of JIF is class-based and thus objects are static.

Most work on information flow control for dynamic languages use dynamic, rather than static, enforcement mechanisms. One noteable exception that is related to our work is [27] for Javascript. A flow analysis is performed to conservatively compute influences. The process is staged, since the presence of `eval` in Javascript makes a whole-program analysis infeasible. To allow a fast syntactic check when loading Javascript, the enforceable policies are very simplistic: writing to or reading from particular variables can be prohibited. In comparison, we support any security lattice, for example, the very expressive DLM [67], and an annotation checker could be used to verify annotations. The staging process could be integrated in our work to support `eval`, which we explore for Nested Control Regions in Chapter 7.

Our work is partially inspired by previous work on extensible object calculi. Two main calculi have been proposed [1, 36], and both have been extended to include subtyping and object extension [21, 23, 22, 58, 59, 60].

Incomplete objects have been considered, for example, in [23, 58, 18]. Types are split up into an interface and a completion component. Only methods in the interface component might be called, while the completion component defines dependencies (e.g., method that need to be added to complete an object). Our type system does not separate between interface and completion components, since we want to allow calling potentially non-existing methods.

8.4.2 SQL

Noninterference for database interactions has been treated before. Li and Zdancewic [54] describe a web programming language extended with an interface to a relational database. Concrete queries a program wants to make are wrapped in a typed interface, so that SQL queries do *not* appear in the actual program. The exact syntax and semantics of the wrapped queries are not defined, and

not integrated into the language. The paper assumes them given and to preserve noninterference. Furthermore, only a very inexpressive fragment of queries is described: just simple selects of data in a single table. Issues of merging or updating tables are not handled.

Li and Zdancewic do not use a type system for the database itself. Instead, each query wrapper defines the security levels of inputs and outputs. The paper does not suggest any method to ensure consistency among wrappers. Our approach, on the other hand, labels the database schema. As such, all queries over the schema automatically enforce the same policy, as column labels are shared among all queries.

The immediate expressive power of the query wrapper type annotations is similar to our type system: all rows share the confidentiality level for respective columns. However, we use an additional level for each table to handle insertions, deletions and table joins.

The Haskell Automatic Information Labeling System (HAILS) [83, 42] is an application of the Labeled IO approach (using monads and arrows) to building web frameworks. Here the Labeled IO monad tracks all information flows of encapsulated computations. If a computation would lead to a potential leak in the current context of the monad, the execution is prevented. This is, in essence, type-checked dynamic information flow control: the correctness of the monad is established by its typability, but the actual control of flows is done at runtime.

HAILS as a web framework covers database storage backends. Policies govern how rows of a table should be labeled. HAILS, in effect, allows a simple form of dependent typing: a column value may be used to label other columns. This allows, for example, user-password tables where each password is protected by the principal of its owner. To support working with values with labels that cannot be known ahead of time, HAILS supports label comparison operators.

The key difference to our approach is that HAILS is, as mentioned, enforcing information flow dynamically. This leads to significant overhead: the authors implemented a simple posting board, which was benchmarked to be slightly slower than a non-secure implementation in *PHP*. That means that the compiled and optimized Haskell program is slower than the interpreted PHP program. Our approach is a pure type system approach: only typable queries are allowed, and if

queries are typable, they are guaranteed to be secure. No runtime overhead is incurred.

Another security property enforced over databases is differential privacy [34]. Differential privacy ensures that a secure algorithm will behave approximately the same on closely related data sets, that is, sets varying on a single element. This implies that the presence or absence of an element will not affect the algorithm's outcome significantly. This notion of security is different from noninterference: it is a statistical property that enforces a notion of anonymity over range queries. As such, it is not directly related to our work.

8.5 Implementation

8.5.1 Intermediate Representations

Several intermediate representations have been developed over the last decades. Java bytecode [56] and .NET CIL [35] are established well in practice. Both define operations over a stack-based virtual machine. While this format allows for a compact code, it is not well-adapted for register-based target machines. In contrast, NCR is easy to analyze and optimize, and annotated information can be efficiently verified.

SafeTSA [4] is a closely related IR. It is SSA-based, structures code in its Unified Abstract Syntax Tree and is strongly typed. All three properties are used to encode SafeTSA code in a way that is type- and reference-safe by construction. However, SafeTSA is strongly bound to Java only, both in respect to the type system and the control structures, whereas NCR accomodates different input languages by lower-level control structures that can be nested to emulate complex structures, and can be extended with annotated type systems for different input languages.

The Dalvik VM is the runtime environment of the Android platform. It uses a register-based model and code can be translated from Java bytecode. The code is not structured and strongly adapted to small devices, e.g., the register set is restricted to a relatively small number of registers (in fact, most instructions can only reference sixteen registers). Thus, the Dalvik Executable format is more restricted and less structured than our format, and as such less adapted to the efficient and

incremental verification of information flow policies.

The Parrot VM[1] was developed for the Perl programming language, but designed to be more general to accept multiple input languages. It accepts multiple register-based input formats (e.g., Parrot Intermediate Representation), which all get compiled to an internal bytecode format on-the-fly. The code is not structured and as such harder to analyze and verify than NCR.

YARV [82] is a virtual machine for Ruby. The IR is stack-based and tailored towards Ruby only, whereas NCR is language-agnostic. RIL [37] is an intermediate language for Ruby. Compared to NCR, it is very high-level and retains many of Ruby's control structures while removing duplicates. RIL and NCR have different goals: RIL is designed for ease of source code analysis and code transformation, whereas NCR is an easy-to-verify transport format.

8.5.2 Secure Information Flow

Wagner et al. [90] designed a Java client-server architecture with an extended interpreter at the client side. Java bytecode is analyzed and modified for partial loading. The default Java bytecode and classfile format is changed for space conservation. For example, symbolic names are replaced with integers and code is pre-linked, both reducing the size of the symbol table. Actual bytecode is partitioned according to a static analysis that determines the probability of its use. Only the likely parts are initially distributed, and the client requests missing parts when necessary.

Our adaptation of NCR with placeholder regions achieves the same goal. However, static analysis for usage prediction, which is necessary to optimize the communication overhead, is significantly simpler on structured, register-based code - structured NCR code can be easily analyzed and split up. Our implementation allows code to be partitioned along compartment boundaries. The structuring of the control regions simplifies where to break-up the code, as well as analyze which parts are likely to be executed and should be transferred in a chunk. A package distribution format can be used to reduce the constant pool similar to the classfile compression. Furthermore, we show how to correctly verify programs that are incrementally loaded. on the example of information

[1] `http://www.parrot.org/`

flow control, which is possible through our baseline single-pass verification algorithm.

Information flow control has been a topic for many years. JFlow/JIF [66] and FlowCaml [72] extend the basic type system of a practical language (Java and ML, respectively). JIF is based on the decentralized label model [67]. Both systems check the program at compile time only. The product is then distributed to the client, who has no way to check the program again, whereas NCR is designed to be easily and incrementally verifyable. The basic NCR is technique-agnostic, i.e., the distributed label model and JIF rules are only one way to enforce information flow control over NCR code.

Several other languages and systems have been designed for static information flow control, e.g., [84, 61, 93]. Most are research languages based on the typed lambda-calculus and some have been proposed as intermediate representations. We believe that a traditional, register-based IR is better suited for current practical purposes: all practical runtime environments use register-based IRs internally, and physical target machines use the same model.

The Mobius project applies proof-carrying code techniques to a subset of Java known as MIDP, used on mobile devices. As part of the project, Barthe and Rezk developed a non-interference bytecode verifier [15]. The algorithm needs three passes to establish non-interference and uses annotation techniques to ensure efficient verification in the last pass. Compared to our solution, the results are restricted: the MIDP specification basically makes a whole-program analysis possible, since class loading is not allowed. This significantly simplifies the flow analysis. In our opinion, it is a better design choice to create an intermediate representation that is easy to fully verify by construction, instead of retro-fitting an existing solution.

Swift [26] is a framework for secure web application partitioning. Its intermediate representation called WebIL is used during compilation, but is not intended to be a mobile code format. It only contains partitioning constraints instead of full security labels. This makes load-time checking of code impossible.

An alternative to static information flow control checks flows dynamically at runtime. However, this incurs a significant overhead due to security label computations [12]. Laminar [75]

allows security to be traded for lower overhead: only selected values need to be labelled, and such values can only be accessed in a special region, which reduces overhead. This "secure region" is comparable to our single-exit region, in that it suppresses any knowledge of control flow to leak out. However, the PC is raised at the beginning of the PC to its final value. This means that a programmer is more restricted in using the secure region, and secure regions have to be tightly nested.

Several hardware extensions have been proposed to support dynamic information flow control. RIFLE [87], for example, adds security registers to dynamically track information flows. Implicit information flows can be tracked through the placement and inclusion of security registers in label computations. However, in comparison to our approach, the RIFLE hardware does not verify the (correct) use of the security extensions. A static whole-program analysis at compile time is proposed to translate traditional machine code to the new ISA, which precludes the use in dynamic environments like mobile application platforms. As a last difference, it is not obvious how to extend RIFLE to include declassification.

Our reference region alternatives address the problem of information flow leaking through runtime verification failures. A similar issue arises in dynamic information flow systems, specifically, information can be leaked because labels will only be updated on values that are written on the path taken. RIFLE [87] uses a hybrid system with dynamic label tracking supported by hardware and compiler static analysis to insert label computations that capture all implicit flows.

To the best of our knowledge, only [27] achieves a similar goal to our work. Javascript programs are verified incrementally, with the borders being `eval` instructions. A conservative flow analysis computes influencing definitions. The computation is split between server and client, because the flow analysis is heavy-weight. To ensure fast checking (purely syntactical) and feasible flow analysis, only very simple policies (a variable cannot be read or written) are possible, whereas our technique can use the significantly more powerful policies JIF provides. It is also worth noting that, in [27], the client has to trust the policies sent by the server.

Chapter 9: CONCLUSION

9.1 Summary

It is feasible to control information flow in composed languages with a modular, type-theoretic framework.

Led by the observation that, in practice, host languages are usually at least as *computationally* powerful as the language being embedded, we investigate a *simulation* approach to the problem: If one can show that there exists a simulation for a composed program, properties of the simulation can be used to derive properties of the composed program, side-stepping reasoning over the complex, likely complicated, composed language.

We first formalize what we consider a language composition. We describe a simple `eval` construct and give it reasonable semantics, semantics that such a construct could be assumed to have in general. We leave the concrete implementation abstract and instead use transfer functions and type transfer functions to generically argue about specific compositions. Transfer functions abstract the necessary translation of values between languages, connecting the separate semantics of host and embedded language. Similarly, type transfer functions abstract the translation between types (or just security levels) between languages, connecting the separate type judgment of host and embedded type system. Constraints placed onto these functions guarantee that trivial leaks do not occur, for example, that unobservable information becomes observable. The derived properties appear similar to those required in the framework of Abstract Interpretation [28, 29], letting transfer functions and more specifically type transfer functions appear like Galois connections.

From this starting point, we derive a number of requirements or assumptions. The first set is local and describes a simulation and its properties just for `eval` constructs. The Correct Simulation requirement defines what it means to simulate an `eval` statement, that is, to work on an encoding of the embedded state such that an encoding predicate ($\mathrm{encodes}$) is preserved over the execution. The Typable Simulation requirement ensures that the encoding used by the simulation is fine-grained

enough to retain the security typing of the embedded language. Only then is arguing over the simulation meaningful, as otherwise security information is lost in the translation.

The second set of requirements lifts the local properties from `eval` constructs the complete composed programs. The first of three requirements ensures that replacing `evals` with the observationally equivalent simulations will result in an observationally equivalent program without `evals`, that is, a pure host-language program. As noninterference, our security property, is semantical, this requirement allows us to connect the simulation to the original composed program and guarantee the soundness of the approach. However, to establish noninterference, we also need a complete typing of the transformed program. The remaining two requirements establish this. Similarly to the semantics requirement, the type replacement requirement establishes that a corresponding typing to the composed program, potentially extended by typing constraints for the simulated embedded state, exists. The last requirement then allows to reduce a composed-language typing to a host-language typing if no `eval` constructs are encountered in a program. This, finally, allows us to appeal to the soundness of the host-language security type system to show that all well-typed programs in a language thus composed are noninterferent.

This framework is both general and modular. The second set of requirements can be shown purely by inspecting the host language and the requirements on transfer functions. All effort spent on one composition can be fully reused for any other composition with the same host language. While the first set is in general dependent on the embedded language, in many cases general computational arguments can be used to significantly reduce the proof burden. Thus, the type and form of noninterference proofs of host and embedded language is often not important to our framework: the component proofs can be treated as black boxes, significantly lowering the required domain knowledge for proof efforts.

To mitigate the final remaining technical requirement, that of typable simulations, we introduce a property called *Security Completeness*. This novel property of a security-typed language states that there exists a typable program for any computable, noninterfering computation, which immediately satisfies the requirement for the existence of a typable simulation of any noninterferent

program. We derive sufficient, and in the case of Turing-complete languages necessary, conditions for a security-typed language to be security-complete. As it turns out, these conditions are commonly satisfied over integer computations for languages in the literature, as a case study of languages from three different paradigms (imperative, functional, object-oriented) suggests. We then show how our approach can be extended to include complex datastructures and references. There are practical limitations, as type systems from the literature syntactically allow some typings that are not useful in practice: parts of a datastructure are marked as public, while the complete entity is confidential. We show that FlowML [72] is *not* security-complete for computations over such types, and derive restrictions to categorize computations we can show typable.

We note that is possible to extend both the framework and security completeness. In the case of the framework, the security property of interest, noninterference, is often too strong in practice. We show that our approach can accommodate two notions of *declassification* (see [80]), that is, intentional information release. We study Delimited Release [79], a model for restricting *what* can be released. Enforcement of Delimited Release is based on an effect system, which is easily integrated into our framework. We then study Robust Declassification [92, 68], which models the *who* axis of declassification. Here, the type-system approach in [68] does not allow for composition, while the trace-based approach in [92] has neither enforcement nor strong enough guarantees. We derive a trace-based property between those two that is enforced by the type system in [68] and show that we can apply our approach to preserve this property over compositions.

To demonstrate the practicality of our framework approach, we first propose two novel security type system for languages that might be considered for embedding. We first describe a type system for an object-oriented calculus that handles $\mathrm{method\text{-}not\text{-}found}$ errors. This significantly eases the burden for static enforcement of noninterference in dynamic languages like Javascript, where the dynamism is a huge hurdle for traditional type systems. We trade basic type system safety guarantees (that is, $\mathrm{method\text{-}not\text{-}found}$ errors are guaranteed not to appear in typed programs) for an increased number of accepted programs (as potential errors are accepted, but guaranteed not to infer with the security properties). A second language we investigate is the Structured Query

Language (SQL) for relational database queries. We design a security type system for an expressive fragment of the data manipulation sub-language, which includes selection, insertion, modification and deletes, with table joins and nested queries. We formalize the semantics of our fragment and show that our security type system is sound.

With these languages for embedding in hand, we select the seminal while language from [89] as host language and show the composition of that language and our SQL dialect to be secure. We formally prove the requirements established by our framework and security completeness, which formally guarantees the soundness of the security type system of the composed language.

We thus believe that our framework is a successful step to mitigate the mismatch we observed between software engineering practice and security theory when developing complex software. Namely, it is best practice to break up complex designs into many smaller, easier to handle chunks (called components) and develop them separately, often employing separate tools. A very basic tool is the programming language used for the implementation, which can vary between components. On the security side, enforcing security properties like confidentiality and integrity can be handled by security-typed languages, which extend a basic language's type system with security annotations and flow-enforcing typing rules. Security-typed languages are attractive for several reasons, the most important ones being static enforcement, that is, no or minimal runtime overhead, and formal guarantees, that is, proven properties of typable programs. The literature has a wide array of proposed security-typed language since the seminal development in [89], but has neglected investigations of language *combinations*, as required by modern software engineering practice.

Naive, straightforward approaches to composing security-typed languages have similar drawbacks to early software development. Direct embedding requires deep knowledge of the languages being composed, both on the syntactical and the semantical level. To also give formal security guarantees for the composition, a standard approach would require intimate knowledge of the design and working of the formal proofs. While this is certainly a feasible approach for a domain expert, one more problem arises: the effort for one combination of languages cannot be reused.

Even if one of the component languages is used in a composition again, the formal guarantees must be derived again independently. Our framework eases this burden.

As a last use case, we investigate the use of our framework as a formal tool justifying the correctness of incremental verification in an extended Java virtual machine. We propose Nested Control Regions (NCR), a register-based, structured intermediate representation for program code that is targetted specifically for optimizing compilers. A layered security type system based on the practical JIF [66] language enforces noninterference. We extend NCR with a construct for incremental loading of code: "holes" in the intermediate representations denote places where additional code should be requested from the code producer, for example, a server connected over some network infrastructure. When execution reaches a hole, the code for said hole is downloaded and *verified*, and only after success placed into the hole.

We adapt our framework approach to argue that said technique is actually secure, that is, incremental verification preserves noninterference. The adaptation shows how the holes can be treated like a place of language embedding, even if the embedding is the same language. The requirements on the interface layer between surrounding program and code in the hole are similar to those required by our framework, such that the same approach formally justifies the soundness.

9.2 Future Directions

There are three major directions that promise interesting opportunities.

First, there are some open points in the framework and security completeness. While we have shown how finite, erratic nondeterminism can be handled, it is open how full-scale concurrency, and potentially concurrent embeddings, interact with the approach. For example, if the embedded language does not define or provide support for concurrency, but the host language does, concurrent embeddings will force concurrent semantics onto the embedded language. It has been shown, though, that secure type systems that are sound for sequential programs are not automatically sound when the language is extended to support concurrency.

Related or similar open problems are other leaking channels. In a concurrent setting, timing

of programs can leak information, as the order of execution may be dependent on confidential information. This is known as a timing channel. While there are proposals for type-based control of timing channels, it is not immediately clear whether this fits well into our computational framework.

The second direction is a generalization of the security properties the framework can enforce. We have shown in this thesis how two specific notions of declassification can be accommodated. The techniques developed to enforce a form of robust declassification indicate that other generalized properties may be able to be enforced. The outlook of such a generalization is that the framework could be applied to the general area of component-based software and (arbitrary) properties. A practical application of such a generalized framework would be the formalization of component frameworks like OSGi that allow to dynamically load and replace components.

The third direction is the investigation of more powerful security type systems for embedded language. Currently, our SQL dialect's type system establishes a homogenous typing of the data. Each row's data is typed the same, that is, all data in a column is classified the same. An example of data that could not be typed under such a regime is that of (precise) login data. The most precise classification is that each principal's password is only accessible to the principal (and the login process). But then the levels of data in a column of a table varies by the specific row.

This could be handled if our approach is generalized to dependent types. In that case, the type of an element of a row can depend on the *value* of another element in that row. While switching to a dependently typed embedded language may not seem too complicated, we also would need to investigate the interactions in the host language. If the host language is not dependently typed itself, only a very conservative type transfer function is secure, and the whole advantage of dependent embedded typing is lost. If the host language is dependently typed itself, one now has to be able to match dependent types, that is, types involving values, and show that operations preserve security properties assumed in either language.

Together with stronger embedded security type systems, a case study of the difficulties in implementing complex software in a real-world setting would be highly interesting.

BIBLIOGRAPHY

[1] Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1996. ISBN 0387947752.

[2] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, May 1991. ISSN 0304-3975. doi: 10.1016/0304-3975(91)90224-P. URL `http://dx.doi.org/10.1016/0304-3975(91)90224-P`.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986. ISBN 0-201-10088-6.

[4] Wolfram Amme, Niall Dalton, Jeffery von Ronne, and Michael Franz. SafeTSA: A type safe and referentially secure mobile-code representation based on static single assignment form. In *Proceedings of the Conference on Programming Language Design and Implementation (PLDI'2001)*, volume 36 of *ACM SIGPLAN Notices*, pages 137–147, Snowbird, Utah, USA, June 2001. ACM Press.

[5] Wolfram Amme, Marc-André Möller, and Philipp Adler. Data flow analysis as a general concept for the transport of verifiable program annotations. *Electron. Notes Theor. Comput. Sci.*, 176(3):97–108, 2007.

[6] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In *SAS, LNCS 3148*, pages 100–115, 2004.

[7] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. A logic for information flow in object-oriented programs. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 91–102, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2.

[8] Christopher Anderson and Sophia Drossopoulou. δ: an imperative object based calculus. In *USE 2002*, 2002.

[9] Aslan Askarov and Andrei Sabelfeld. Localized delimited release: combining the what and where dimensions of information release. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, PLAS '07, pages 53–60, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-711-7. doi: 10.1145/1255329.1255339. URL `http://doi.acm.org/10.1145/1255329.1255339`.

[10] Aslan Askarov and Andrei Sabelfeld. Catch me if you can: permissive yet secure error handling. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 45–57, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-645-8.

[11] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In Sushil Jajodia and Javier Lopez, editors, *Computer Security ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, pages 333–348. Springer Berlin / Heidelberg, 2008.

[12] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, PLAS '09, pages 113–124, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-645-8.

[13] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Proceedings of the 15th IEEE workshop on Computer Security Foundations*, CSFW '02, pages 253–267, Washington, DC, USA, 2002. IEEE Computer Society.

[14] Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *J. Funct. Program.*, 15(2):131–177, March 2005. ISSN 0956-7968. doi: 10.1017/S0956796804005453.

[15] Gilles Barthe and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In *European Symposium on Programming, Lecture Notes in Computer Science*, pages 125–140. Springer, 2007.

[16] Gilles Barthe and Bernard P. Serpette. Partial evaluation and non-interference for object calculi. In *Proceedings of the 4th Fuji International Symposium on Functional and Logic Programming*, pages 53–67, London, UK, 1999. Springer-Verlag. ISBN 3-540-66677-X.

[17] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report MTR-2547, Vol. 1, MITRE Corp., Bedford, MA, 1973.

[18] Lorenzo Bettini, Viviana Bono, and Silvia Likavec. A core calculus of mixins and incomplete objects. In *Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '04, pages 208–209, New York, NY, USA, 2004. ACM. ISBN 1-58113-833-4.

[19] Kenneth J. Biba. Integrity considerations for secure computer systems. *MITRE Co., technical report ESD-TR 76-372*, 1977.

[20] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in cω: The power is in the dot. In *In ECOOP'02*, 2002.

[21] Viviana Bono and Michele Bugliesi. Matching constraints for the lambda calculus of objects. In *Proceedings of the Third International Conference on Typed Lambda Calculi and Applications*, pages 46–62, London, UK, 1997. Springer-Verlag. ISBN 3-540-62688-3.

[22] Viviana Bono and Kathleen Fisher. An imperative, first-order calculus with object extension. In *Proceedings of the 12th European Conference on Object-Oriented Program-*

ming, pages 462–497, London, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6. URL http://dl.acm.org/citation.cfm?id=646155.679695.

[23] Viviana Bono, Michele Bugliesi, Mariangiola Dezani-Ciancaglini, and Luigi Liquori. Subtyping constraints for incomplete objects (extended abstract). In *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '97, pages 465–477, London, UK, 1997. Springer-Verlag. ISBN 3-540-62781-2.

[24] Gilad Bracha. Pluggable type systems. OOPSLA workshop on revival of dynamic languages, 2004.

[25] Guangyu Chen and Mahmut Kandemir. Verifiable annotations for embedded java environments. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 105–114, New York, NY, USA, 2005. ACM Press.

[26] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.

[27] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for javascript. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 50–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1.

[28] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. doi: 10.1145/512950.512973. URL http://doi.acm.org/10.1145/512950.512973.

[29] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '79, pages 269–282, New York, NY, USA, 1979. ACM. doi: 10.1145/567752.567778. URL `http://doi.acm.org/10.1145/567752.567778`.

[30] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. Flowfox: A web browser with flexible and precise information flow control. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 748–759, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382275. URL `http://doi.acm.org/10.1145/2382196.2382275`.

[31] Zhenyue Deng and Geoffrey Smith. Lenient array operations for practical secure information flow. In *Proceedings of the 17th IEEE workshop on Computer Security Foundations*, pages 115–, Washington, DC, USA, 2004. IEEE Computer Society.

[32] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19:236–243, May 1976. ISSN 0001-0782.

[33] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, SP '10, pages 109–124, Washington, DC, USA, 2010. IEEE Computer Society.

[34] Cynthia Dwork. Differential privacy. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *ICALP (2)*, volume 4052 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2006. ISBN 3-540-35907-9.

[35] ECMA International. Common language infrastructure (cli) partitions i to vi. Standard ECMA-335, ECMA, 2006.

[36] Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. of Computing*, 1:3–37, March 1994. ISSN 1236-6064.

[37] Michael Furr, Jong-hoon (David) An, Jeffrey S. Foster, and Michael Hicks. The ruby intermediate language. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 89–98, New York, NY, USA, 2009. ACM.

[38] Andreas Gampe and Jeffery von Ronne. Efficient incremental information flow control with nested control regions. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, PLASTIC '11, New York, NY, USA, 2011. ACM.

[39] Andreas Gampe and Jeffery von Ronne. Information flow control with message-not-understood errors. *International Workshop on Foundations of Object-Oriented Languages (FOOL)*, 2011.

[40] Andreas Gampe and Jeffery von Ronne. A framework for composing security-typed languages. In *Proceedings of the Workshop on Foundations of Computer Security*, FCS '13, pages 34–48, 2013.

[41] Andreas Gampe and Jeffery von Ronne. Security completeness: Towards noninterference in composed languages. In *Proceedings of the Eighth ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '13, pages 27–38, New York, NY, USA, 2013. ACM.

[42] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *10th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 47–60. USENIX, 2012.

[43] J A Goguen and J Meseguer. *Security Policies and Security Models*, volume pages, pages 11–20. IEEE, 1982.

[44] Vivek Haldar. Verifying data flow optimizations for just-in-time compilation. Technical Report SMLI TR-2002-118, Sun Microsystems, 2002.

[45] Phillip Heidegger and Peter Thiemann. Recency types for analyzing scripting languages. In *Proceedings of the 24th European conference on Object-oriented programming*, ECOOP'10, pages 200–224, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14106-4, 978-3-642-14106-5.

[46] Nevin Heintze and Jon G. Riecke. The slam calculus: programming with secrecy and integrity. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 365–377, New York, NY, USA, 1998. ACM. ISBN 0-89791-979-3.

[47] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '06, pages 79–90, New York, NY, USA, 2006. ACM. ISBN 1-59593-027-2. doi: 10.1145/1111037.1111045. URL `http://doi.acm.org/10.1145/1111037.1111045`.

[48] Sebastian Hunt and David Sands. From exponential to polynomial-time security typing via principal types. In *Proceedings of the 20th European conference on Programming languages and systems*, ESOP'11/ETAPS'11, 2011.

[49] Stefan Kahrs. Limits of ML-definability. In *Proceedings of the 8th International Symposium on Programming Languages: Implementations, Logics, and Programs*, PLILP '96, pages 17–31, London, UK, UK, 1996. Springer-Verlag.

[50] Stefan Kahrs. Well-going programs can be typed. In *Proceedings of the 6th international conference on Typed lambda calculi and applications*, TLCA'03, 2003.

[51] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 413–428, 2011. doi: 10.1109/SP.2011.19.

[52] Christoph Kerschbaumer, Gregor Wagner, Christian Wimmer, Andreas Gal, Christian Steger, and Michael Franz. Slimvm: a small footprint java virtual machine for connected embedded systems. In *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 133–142, New York, NY, USA, 2009. ACM.

[53] L. J. LaPadula and D. E. Bell. Secure computer systems: A mathematical model. Technical Report MTR-2547, Vol. 2, MITRE Corp., Bedford, MA, 1973. Reprinted in *J. of Computer Security*, vol. 4, no. 2–3, pp. 239–263, 1996.

[54] Peng Li and Steve Zdancewic. Practical information-flow control in web-based information systems. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations*, CSFW '05, 2005.

[55] Peng Li and Steve Zdancewic. Encoding information flow in haskell. In *Proceedings of the 19th IEEE workshop on Computer Security Foundations*, CSFW '06, 2006.

[56] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. The Java Series. Addison-Wesley, second edition, 1999.

[57] Sam Lindley, James Cheney, and Philip Wadler. Shredding higher-order nested queries. Available at `http://homepages.inf.ed.ac.uk/slindley/papers/shredding-draft-august2012.pdf`.

[58] Luigi Liquori. An extended theory of primitive objects: First order system. In *In Proc. of ECOOP*, pages 146–169. Springer-Verlag, 1997.

[59] Luigi Liquori. On object extension. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 498–522, London, UK, 1998. Springer-Verlag. ISBN 3-540-64737-6.

[60] Luigi Liquori. Bounded polymorphism for extensible objects. In Thorsten Altenkirch, Bernhard Reus, and Wolfgang Naraschewski, editors, *Types for Proofs and Programs*, volume

1657 of *Lecture Notes in Computer Science*, pages 149–165. Springer Berlin / Heidelberg, 1999. ISBN 978-3-540-66537-3.

[61] 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 *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 321–334, New York, NY, USA, 2009. ACM.

[62] Heiko Mantel, David Sands, and Henning Sudbrock. Assumptions and guarantees for compositional noninterference. In *Proceedings of the 2011 IEEE 24th Computer Security Foundations Symposium*, CSF '11, 2011.

[63] M. Marcus and Amir Pnueli. Using ghost variables to prove refinement. In *Proceedings of the 5th International Conference on Algebraic Methodology and Software Technology*, AMAST '96, pages 226–240, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61463-X. URL `http://dl.acm.org/citation.cfm?id=646057.678341`.

[64] Daryl McCullough. Specifications for multi-level security and a hook-up. *Security and Privacy, IEEE Symposium on*, 0:161, 1987.

[65] Robin Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-115007-3.

[66] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, 1999.

[67] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *In Proc. 17th ACM Symp. on Operating System Principles (SOSP*, pages 129–142. ACM Press, 1997.

[68] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *J. Comput. Secur.*, 14(2):157–196, April 2006.

[69] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[70] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for java. In *In 12th International Conference on Compiler Construction*, pages 138–152. Springer-Verlag, 2003.

[71] Jens Palsberg. Efficient inference of object types. *Inf. Comput.*, 123:198–209, December 1995. ISSN 0890-5401.

[72] François Pottier and Vincent Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '02, pages 319–330, New York, NY, USA, 2002. ACM. ISBN 1-58113-450-9.

[73] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003.

[74] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 1–12, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3.

[75] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. Laminar: practical fine-grained decentralized information flow control. In *PLDI '09: Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, pages 63–74, New York, NY, USA, 2009. ACM.

[76] Alejandro Russo, Koen Claessen, and John Hughes. A library for light-weight information-flow security in haskell. In *Proceedings of the first ACM SIGPLAN symposium on Haskell*, Haskell '08, pages 13–24, New York, NY, USA, 2008. ACM.

[77] P. Y. A. Ryan and S. A. Schneider. Process algebra and non-interference. *J. Comput. Secur.*, 9(1-2):75–103, January 2001.

[78] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21:2003, 2003.

[79] Andrei Sabelfeld and AndrewC. Myers. A model for delimited information release. In *Software Security - Theories and Systems*, volume 3233 of *Lecture Notes in Computer Science*, pages 174–191. Springer Berlin Heidelberg, 2004.

[80] Andrei Sabelfeld and David Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE workshop on Computer Security Foundations*, CSFW '05, pages 255–269, 2005.

[81] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Comput. Secur.*, 17(5):517–548, October 2009. ISSN 0926-227X. URL `http://dl.acm.org/citation.cfm?id=1662658.1662659`.

[82] Koichi Sasada. Yarv: yet another rubyvm: innovating the ruby interpreter. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 158–159, New York, NY, USA, 2005. ACM.

[83] Deian Stefan, Alejandro Russo, John C. Mitchell, and David Mazières. Flexible dynamic information flow control in Haskell. In *Haskell Symposium*, pages 95–106. ACM SIGPLAN, September 2011.

[84] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *SP '08: Proceedings of the 2008 IEEE Symposium on Security and Privacy*, pages 369–383, Washington, DC, USA, 2008. IEEE Computer Society.

[85] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in fine. In *Proceedings of the 19th European conference on Program-*

ming Languages and Systems, ESOP'10, pages 529–549, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11956-5, 978-3-642-11956-9. doi: 10.1007/978-3-642-11957-6_28. URL http://dx.doi.org/10.1007/978-3-642-11957-6_28.

[86] Ta-chung Tsai, Alejandro Russo, and J. Hughes. A library for secure multi-threaded information flow in haskell. In *Computer Security Foundations Symposium. CSF '07*, pages 187–202, July 2007.

[87] Neil Vachharajani, Matthew J. Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A. Blome, George A. Reis, Manish Vachharajani, and David I. August. Rifle: An architectural framework for user-centric information-flow security. In *MICRO 37: Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[88] Steve Vandebogart, Petros Efstathopoulos, Eddie Kohler, Maxwell Krohn, Cliff Frey, David Ziegler, Frans Kaashoek, Robert Morris, and David Mazières. Labels and event processes in the asbestos operating system. *ACM Trans. Comput. Syst.*, 25 (4), December 2007. ISSN 0734-2071. doi: 10.1145/1314299.1314302. URL http://doi.acm.org/10.1145/1314299.1314302.

[89] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4:167–187, January 1996. ISSN 0926-227X.

[90] Gregor Wagner, Andreas Gal, and Michael Franz. Slim vm: optimistic partial program loading for connected embedded java virtual machines. In *PPPJ '08: Proceedings of the 6th international symposium on Principles and practice of programming in Java*, pages 117–126, New York, NY, USA, 2008. ACM.

[91] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115:38–94, November 1994. ISSN 0890-5401.

[92] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE workshop on Computer Security Foundations*, CSFW '01, pages 5–, 2001.

[93] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPs. In *ESOP '01: Proceedings of the 10th European Symposium on Programming Languages and Systems*, pages 46–61, London, UK, 2001. Springer-Verlag.

[94] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 19–19, Berkeley, CA, USA, 2006. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267308.1267327`.

[95] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, pages 293–308, Berkeley, CA, USA, 2008. USENIX Association. ISBN 111-999-5555-22-1. URL `http://dl.acm.org/citation.cfm?id=1387589.1387610`.

[96] Tian Zhao. Type inference for scripting languages with implicit extension. *International Workshop on Foundations of Object-Oriented Languages*, 2010.

VITA

Andreas Robert Gampe received his Dipl. Inf. at the Friedrich-Schiller Universität Jena, Germany, in 2006, with his thesis work supervised by Dr. Wolfram Amme. After working as a software developer in Jena, Andreas attended The University of Texas at San Antonio, pursuing a Ph.D. degree in the Department of Computer Science under the supervision of Dr. Jeffery von Ronne. His research interests broadly lie in the fields of programming languages and computer security. He is currently working on formalized notions of language compositions, and their implications for computer security, specifically in the context of security-typed languages.