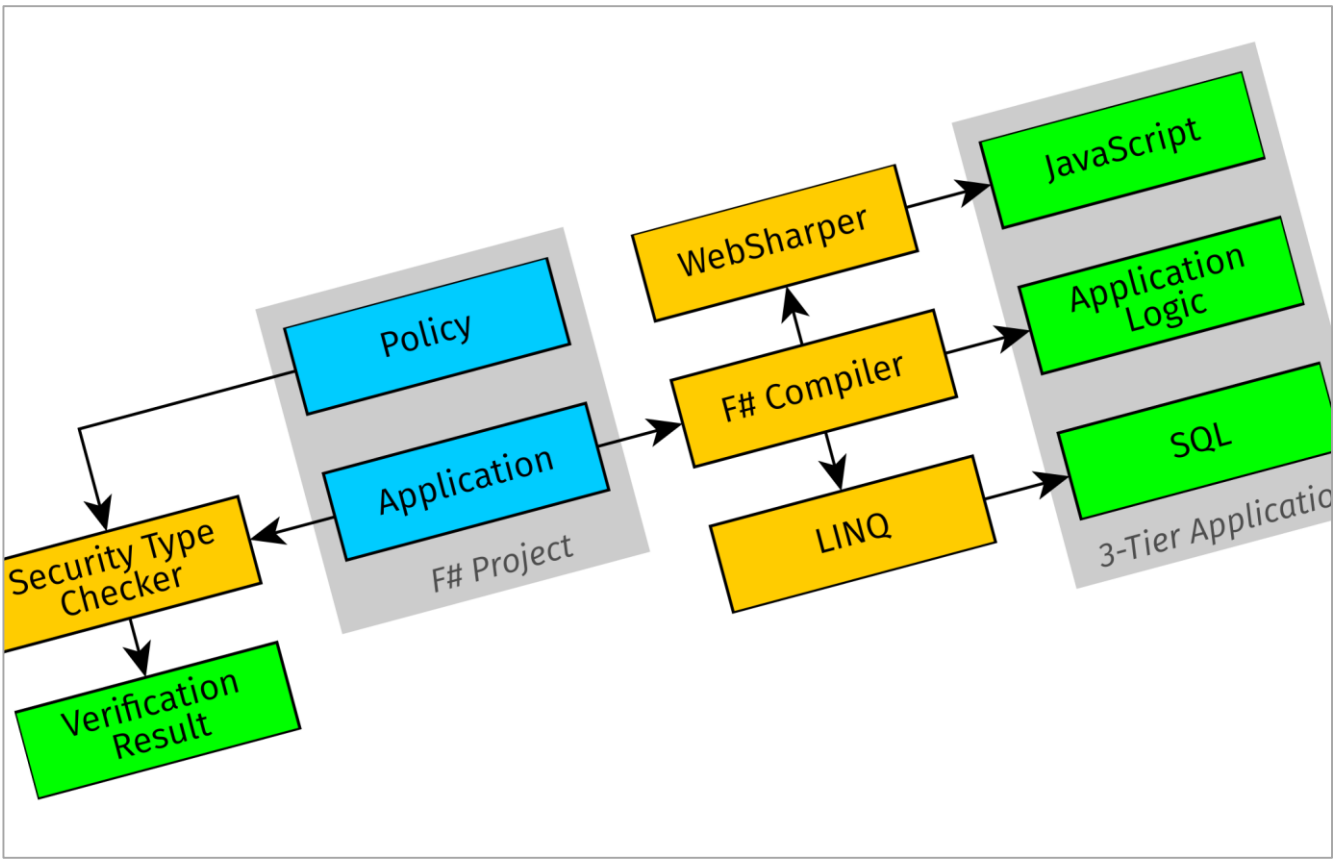
CHALMERS
UNIVERSITY OF TECHNOLOGY



Practical Cross-Tier Information Flow Control for Web Applications

Master's thesis in Computer Systems and Networks

BENJAMIN LIEBE

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2015

Practical Cross-Tier Information Flow Control for Web Applications

BENJAMIN LIEBE

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Cover:
Architecture of the prototype implementation JSLINQ.

Göteborg, Sweden 2015

Abstract

Web applications are increasingly processing critical data. Maintaining information security in them is therefore a very important task. This is however a hard problem, as web applications typically split their functionality between different components in a three-tier architecture. One promising approach for this problem is to apply methods of Information Flow Control (IFC) across all tiers of web applications. These methods go beyond the possibilities of traditional security mechanisms such as access control and allow to tightly control where for example confidential information may or may not end up. Embedded into current research at Chalmers, this thesis aims to put the theory into practice: it first takes a closer look at what IFC actually means for web applications, which yields a discussion of how IFC policies can be used to better protect trust relationships and the business logic of the application. As a second step does the thesis use a given formal model for a security type system and turn it into a working prototype that extends the F# programming language in an unobtrusive way. Viability of this prototype is finally demonstrated by developing and discussing six different case studies that touch different aspects of web application development. The results show for the prototype that practical IFC requires a large initial effort but allows later a good integration into existing languages and development processes.

Acknowledgements

One of my personal goals of joining a Master's program at Chalmers was to get an insight into actual research activities. I am therefore very grateful that Andrei Sabelfeld offered me a thesis topic in his research group and acted as the examiner. I also owe a big thank you to the JSLINQ team consisting of Musard Balliu, Daniel Schoepe and Andrei Sabelfeld: the various discussions provided valuable feedback for the progress of this thesis and in my learning. Musard did also a great job as my supervisor and Daniel was very helpful to get me started on the topic and in providing the implementation for the constraint solver. Thanks go also to the opponents, who provided valuable feedback that should make the final version of the report more understandable. Finally, I would not have been able to complete this without the support of my partner Katharina, who also took care of our son Erik when I needed to focus on the thesis: thank you very much!

Benjamin Liebe, Göteborg 2015-10-27

Contents

Chapter 1

Introduction

1.1 Motivation

Information security becomes increasingly a basic need for today's information society. A broadly accepted approach to information security is to divide it into the three attributes *confidentiality*, *integrity* and *availability*. The security of information that is processed in computer systems has a big impact on personal and corporate life. Individuals use them to manage their finances, social contacts and personal data. Organizations rely on them for mission critical business processes, trust them their trade secrets and make them accessible to employees, partners, customers or even the general public. Both groups are required to trust the security of the data processing. Alone the violation of integrity can for example have serious consequences, as we are reminded again by recent data breaches [18, 22]. Maintaining security is however not trivial, as it touches many topics in design, implementation and operation of a system. Real-life systems typically rely for their security on generally accepted building blocks such as cryptography, access control and the adherence to design principles, as for example summarized early on by Saltzer and Schroeder [29].

This thesis emphasizes the implementation, more specifically the security issues that arise on the *application level*. The discussion is focused on *web applications*, since sensitive information is increasingly processed within web applications. They are nowadays a very popular way to build portable and user-friendly applications. Web-based applications are however also known for a number of special security issues, of which the most prominent are pointed out by the widely recognized OWASP Top 10 project [23]. Some of the more subtle issues arise from the fact that under the hood of a modern web application, functionality is provided by different layers called *tiers*. It is not trivial to guarantee that the intended behavior of the application is always preserved within a multi-tier application, and it is even less trivial to guarantee integrity and confidentiality of the processed data in such a setting. One reason for this is the mix of technologies and paradigms that results from the high specialization of each tier.

1.2 Goals and Results

The work in this thesis explores how Information Flow Control (IFC) can be used to protect confidentiality and integrity in three-tier web applications, which split functionality across browser, application server and database. The work is based on theoretical foundations from as yet unpublished research work by the Programming Language-Based Security research group at Chalmers [2], to which it con-

tributes the implementation and documentation of a proof-of-concept system called JSLINQ[1]. IFC research has produced already similar applications, of which SeLINKS [9] is a particularly strong example. This thesis aims however to be unique in providing a better consideration of practical aspects of web application development: the implementation builds on a mature and commercially supported web application framework in an unobtrusive way and it provides a discussion of how IFC can contribute to correctly model the business logic of an application. Finally, the thesis aims to provide an introduction into the underlying concepts and technologies to make it self-contained. The results obtained during the work and presented in this report are: a discussion of what IFC policies mean in practice for web applications, a working prototype implementation for an IFC system for web applications as well as corresponding example programs that demonstrate how it can be used. The feasibility of the approach is demonstrated by the discussion of six case studies, which represent different kinds of IFC policies.

1.3 Structure

The remainder of this thesis report is organized as follows:

Chapter 2: Background gives an introduction into basic terms and concepts that are important for the understanding of the thesis.

Chapter 3: Information Flow across Tiers provides a discussion of the role that IFC plays in three-tier web applications. It aims to make the concepts more tangible and give them a meaning, as well as linking it to real-life problems that application providers are facing.

Chapter 4: Formal Model presents the theoretical foundations on which the JSLINQ prototype is built. The emphasize lies on the security type system, which serves as the theoretical foundation of the implementation.

Chapter 5: Implementation explains the architecture of the prototype JSLINQ and provides details of its implementation.

Chapter 6: Case Studies introduces case studies, the policies they demonstrate and how they are specified in JSLINQ, together with examples that violate the policy.

Chapter 7: Discussion discusses design choices, provides a discussion of how the results of this work could be used in practice and also addresses limitations of the approach. It concludes with a discussion of related work.

Chapter 8: Conclusion gives an outlook to future developments of the approaches presented in JSLINQ.

[1] See Appendix A for a more detailed account of the contributions.

Chapter 2

Background

2.1 Multi-Tier Architectures

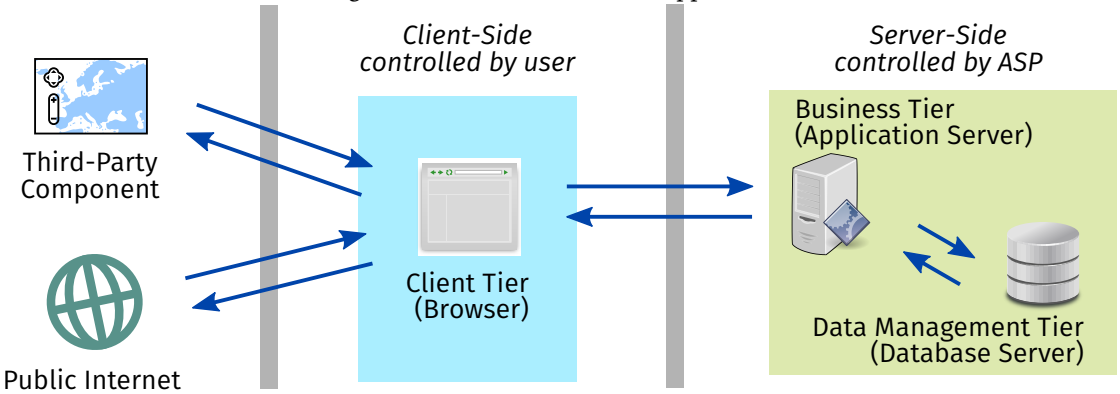
Terms and notions for multi-tier architectures are not always precisely defined. This thesis uses the definition given by Schuldt [31], as it covers all important aspects required for the following explanations. According to Schuldt does a multi-tier architecture spread out functionality across several components that work with each other in "a client/server style interaction". To keep things simple and to stay within the focus of this work, the discussion of multi-tier architectures is limited to web applications. It is however worth to note that this architecture is not limited to web applications and has been used before them.

A typical web application is based on a *three-tier architecture*, consisting of three tiers as described in Table 2.1 and illustrated in Figure 2.1. We start off with the discussion of the *business tier*, as it ultimately governs what the other tiers are doing. The business tier contains the business logic of the application and is usually the place where most of the effort for preserving integrity and confidentiality of data is focused. A wide choice of technologies exists for this tier, including those that seamlessly integrate the adjacent client and data management tiers. This thesis refers to the business tier also as application server, since this is in most cases the technical foundation of it. The *client tier* is in modern applications formed by the web browser that displays content and executes code on the end of the user using technologies such as HTML and JavaScript. This is the only tier where developers are bound to a given set of standardized technologies. The *data management tier* is used for the structured and persistent storage of data, for which usually a relational database management system (RDBMS) such as MySQL or Microsoft SQL Server is used. Since the three-tier architecture is a well-established standard in web application development, do the following discussions focus on this scenario.

Table 2.1: Typical tiers and tasks according to [31] and corresponding web technologies.

Tier	Task	Technology
Client	Presentation	Browser (HTML, JavaScript, CSS)
Business	Application Logic	Application framework (e.g. PHP, ASP.NET, J2EE)
Data Management	Database Server	RDBMS (e.g. MySQL, MSSQL, Oracle)

Figure 2.1: A Three-Tier Web Application.



2.2 Uniform Programming Models

Tiers within a multi-tier architecture are specialized according to their role, which is why they use different technologies. Looking at the example of the three-tier architecture for web applications explained in Section 2.1, we can see that development requires the use of at least three different languages. They can follow different paradigms (giving for example rise to a problem known as the *object-relational impedance mismatch* [38, 5]) or encode data differently. Furthermore, since tiers interface with each other in a client/server style interaction, we effectively have a distributed system with all its complexity. All these issues result in additional sources for programming mistakes that can have a negative effect on the security of the application, as for example evident by the prevalence of injection attacks. *Uniform programming models* are a way to shield the developer from these problems, by allowing to perform all development in one programming language and transparently bridging the gap between tiers. This can for example be achieved by automatically generating the code for the different tiers from a code base written in a single source language. Assuming that this transformation is implemented correctly, uniform programming can already on its own result in improved application security, as the application is closer to the intended business logic and avoids certain kinds of injection vulnerabilities. Most existing implementations are only bridging two tiers, such as LINQ [19] , Rails or GWT [16]. Solutions that span across all tiers are still more of an academic nature [8], with WebSharper [39] being a notable exception that enjoys industrial-grade support.

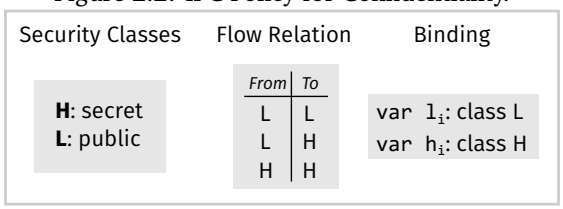
2.3 Information Flow Control

Information Flow Control (IFC) has a simple purpose: it restricts the flow of information within an application according to a policy in order to protect confidentiality and integrity of data. The following explanations give a brief overview of important points in IFC research and introduces key concepts. A good starting point for more detailed explanations is the survey performed by Sabelfeld and Myers [28] as well as an introduction into the topic by Smith [33].

2.3.1 Origin of IFC

The roots of IFC research go back to the mid-1970s with publications by the Dennings [11], which propose to use static program analysis in order to control how information propagates through a system. This is done by enforcing an *information flow policy* which contains the available *security*

Figure 2.2: IFC Policy for Confidentiality.



classes that can be assigned to a piece of information (often only two classes named "high" and "low" are used in examples, while more classes are possible), a list of allowed transitions between these classes (called *flow relation*) and a *binding* which tells for each storage object (e.g. memory locations, variables) to which security class it belongs. A *flow* occurs in this setting when the program reads from one storage object and derives from it a value that is written to another storage object. Such a flow is only permitted when the transition between the involved security classes is explicitly contained in the flow relation.

2.3.2 Examples of Flows

Let us look at a few simple examples for information flows within a simple confidentiality policy. We start with two security classes that represent data confidentiality: security class "L" represents public data while security class "H" represents secrets/confidential data. The goal is to avoid flows from security class H to L, which conversely means that flows from L to H and between the same security classes are allowed and thus part of the flow relation. Figure 2.2 summarizes this policy in a more graphical way. It also shows a simple binding which is used by the examples: the name of each variable corresponds to its security class, so variable $h_{23}$is for example of class H and variable $l_{42}$of class L. The following expressions are examples of *explicit information flows* within this policy, where content of one variable is copied to a second one using a simple assignment operator ":=". The first two cases are valid flows, since they are contained in the flow relation. The third example is however a violation of the confidentiality policy, since the expression attempts to copy a secret value to a public variable.

1. $h_0 := l_0$ ✔
2. $h_1 := h_0$ ✔
3. $l_0 := h_0$ ✘

A second important class are *implicit information flows*, which involve conditional expressions such as if-statements. Implicit flows are one instance of *covert channels* through which information can flow. Sabelfeld and Myers distinguish in their survey up to six different types of such covert channels [28]. The following example shows an implicit flow that violates the confidentiality policy. The value of the public variables is not directly derived from a secret variable, but the publicly accessible content in l_0 still depends on the secret information:

$if\ \ h_0\ \ then\ \ l_0 := l_1\ \ else\ \ l_0 := l_2$ ✘

Depending on the used programming language, there exists also a third kind of channel through which information can leak. Normally is it very common to assume a purely functional language when talking about IFC, as this greatly simplifies reasoning about the programs: the following F# function of the type $unit \rightarrow unit$ does for example in the purely functional case never cause any information flows. It does neither receive nor return information and cannot modify any variables

that are outside of the function, since it uses the unit data type that stands for “no value”, which is written as `()`.

```
let doNothing () =
  ()
```

The web does however not fit easily into such a pure view, as different APIs allow various kinds of state changes that affect the environment. Examples for this are the local storage capabilities of HTML, DOM manipulation, Cookies and even the simple fact that externally observable HTTP requests can be triggered. Things like this are called *side-effects* and they are the reason why a function of the inconspicuous type $unit \rightarrow unit$ might still be able leak information. Side-effects are not restricted to the web and do for example also typically occur in functions that interface with the operating system. The following example shows a function with the same type signature as before, but this time does it exhibit side-effects by showing a message to the user:

```
let doSomething () =
  JS.Alert ("Hello World!")
  ()
```

We will see that the formal model and the implementation therefore take special care in the handling of side-effects.

2.3.3 Declassification

Tracking of information flows alone is not enough, since even programs that properly process confidential data need at some point to disclose parts of a secret. A simple example is a login program: it receives the confidential password as input and discloses by its behavior that the typed in password is either correct (successful login) or wrong (failed login) [25]. This shows that IFC systems require a method to make confidential information public again, in a process that is called *declassification*. For the purpose of this thesis do we avoid the complexities of this topic by resorting to a simple declassification method called *escape hatches*. These are nothing more than functions that receive confidential input and return public output, which are however only allowed to perform this under special circumstances (e.g. because they are part of the trusted policy, while they would cause a policy violation when being used outside the policy). What exactly constitutes declassification is left to the person that implements the escape hatch, simple examples are encryption or hashing of data. Making sure that escape hatches cannot be abused is not trivial, Sabelfeld and Sands have for example identified four dimensions of declassification that need to be considered [27].

2.3.4 Security Type Systems

The Denning model serves as the foundation of IFC research up to today. A proof of its correctness was not part of the original publications, which is why in the 90s Volpano and Smith [37] expressed it as a type system within type theory and additionally combined it with the notion of *non-interference* [14], which formally expresses the requirement that secrets may not be disclosed. This laid the foundation for the use of *security type systems* in IFC research, which is an area of ongoing research. In the new millennium several researchers worked on the creation of the first type systems that capture information flow for modern programming languages: in 2001 Myers proposed and implemented Jif [21] that resembles Java, while Pottier and Simonet took care of functional languages by presenting Flow Caml in 2003 [32]. The formal model on which this thesis is based can be traced back to their work and uses a similar proof technique.

Chapter 3

Information Flow across Tiers

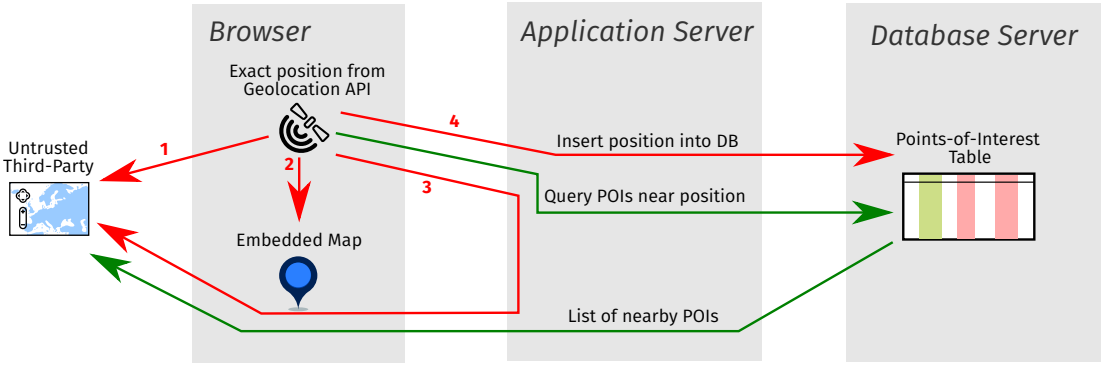
The application of IFC to the protection of confidentiality and integrity in three-tier web applications raises some interesting questions. We first look at what IFC actually means for web applications before we take a look at more specific applications. The discussion follows a top-down approach, it first gives the big picture and presents the involved parties, proceeds to view information flows on the level of the tiers and finally takes a look at what is happening inside a tier.

3.1 Overview

Figure 3.1 gives an example for the complexity of information flow within a three-tier web application. It assumes a simple web application which provides a location-based service, for example showing a list of points of interest (POIs) based on the user's current location. The exact location of the user is considered confidential and must not leak to third-parties, including the provider of the external map service that is used. It is however acceptable for us that the application server uses the exact user location in order to retrieve a list of nearby points of interest, which are in turn displayed on the map. It is assumed that each displayed map marker results in the disclosure of the marker's position to the map provider. The intended flow of information is indicated by the green arrows. In order to protect the user's location do we need to make sure that the JavaScript code in the browser does not leak it directly to the third party via HTTP requests (arrow 1 in the figure) or add it as a marker to the map via JavaScript (arrow 2), but we also must make sure that it cannot leak indirectly via the other tiers. Arrow 3 in the figure shows a case where the exact position is disclosed to the application server as intended, but then the programming of the application server appends the received exact position to the list of POIs that are displayed on the map, eventually disclosing it to the map provider. The same applies to the case shown by arrow 4, where the application server inserts the exact user position into a new POI in the database, which will be part of the resulting POI list and thus also be displayed on the map and get disclosed. Since the transmission of the user's location to the application server cannot be removed without breaking the desired functionality, we have to actually monitor the flow of the information through the code across the tiers and make sure that no confidential information flows back to the third party.

We can typically identify two parties that are owning and controlling the tiers: the client tier is owned by the end-user, while the application server and database server are both owned by the application service provider (ASP). This thesis sometimes refers to application and database server jointly as *server-side*. Furthermore is the term *third-party* used for any other party located in the Internet, which could either be another ASP providing a component embedded into the application (e.g. Google providing Google Maps) but also an external attacker.

Figure 3.1: Examples of information flows in three-tier application.



End-users and ASPs may both have different motivations and thus requirements for the use of IFC. A user might want to protect himself from privacy violations originating from external attackers (e.g. using Cross-Site Scripting) or ASPs (e.g. abusing access to APIs in the browser). The user can however only verify the information flow of JavaScript code executed in the browser, as the code on the server-side is not accessible. There exists a large body of work that provides solutions for this use case, where the policy is predetermined by the user.

The ASP is in a more powerful position than the user, as it has full control over the server-side. Additionally, it is normal that the JavaScript code running within the client browser is also provided by the ASP. It is thus only the ASP who has enough control over all tiers in order for cross-tier IFC to make fully sense. A possible use case of IFC in this setting is to verify that the application complies with business-specific policies such as isolation or mandatory declassification. We will see a more detailed discussion about how an ASP can benefit from the use of cross-tier IFC in Section 7.2.

3.2 Assumptions

Discussing IFC in a concrete application can grow very complex and confusing, as it involves many degrees of freedom. In an attempt to focus on the aspects that are relevant for this work, does the discussion make the following assumptions:

- The public Internet and third-parties are never trusted.
- Security on the transport layer is maintained by the use of transport encryption, so attacks on this layer are not considered.
- Information flow is only tracked within the application and for example not on the level of the operating system or other components than the three tiers.
- Tiers are serially connected, thus information flows can only occur that way. For example does a flow from the database to the client need to pass through the application server. This also rules out scenarios where the application server requests web services from a third-party directly via the Internet.
- Security classes are limited to the two classes *high* and *low*, abbreviated as H and L. The flow relation allows information to flow from L to H and to stay within the same level. All other flows, especially from H to L, are not permitted. This corresponds to the policy introduced in Section 2.3.2.

- As a result from the above, do discussed example scenarios have only one class of confidential or high-integrity information. Real-world applications have usually different kinds of information that require different policies. The examples do not reflect this, but it can in principle be accommodated with the use of more than two security classes.
- When discussing integrity, we only consider integrity violations resulting from intentional actions by one of the involved parties. Changes resulting from bugs in hard- or software are not considered.

These assumptions remove complexity from the discussions and make it still possible to implement actual applications. It is worth to keep in mind that most of these assumptions can be removed or weakened as they are not fundamental limitations of IFC systems in general.

3.3 Flows between Tiers

It can make sense for an ASP to first make policy decisions on the level of application tiers, where individual storage objects are not considered. This allows us to focus on the trust-relationship between user and ASP and questions such as "which party do we trust with confidentiality and integrity of our data?" Answers to these questions provide already an outline for the required isolation within an IFC policy.

3.3.1 Client ⟷ Application Server

The biggest impact on information security comes from the link between the client and the application server (AS), as two different parties are involved. In the case of information flowing from the AS to the client, it is the point where information leaves the exclusive control of the ASP, which means a potential loss of confidentiality and integrity for that information. The ASP might thus want to specify for each class of sensitive information if it is allowed to reach the client *at all*, which represents an *isolation policy*.

Such an isolation policy can also be applied in the reverse direction: when information flows from the client towards the server-side, the ASP becomes responsible for the protection of that information, which is not always desirable. Processing certain kinds of information on the server-side can for example result in stricter compliance requirements for the operations of the ASP - being able to control that certain classes of sensitive information stay isolated on the client can be helpful in such cases.

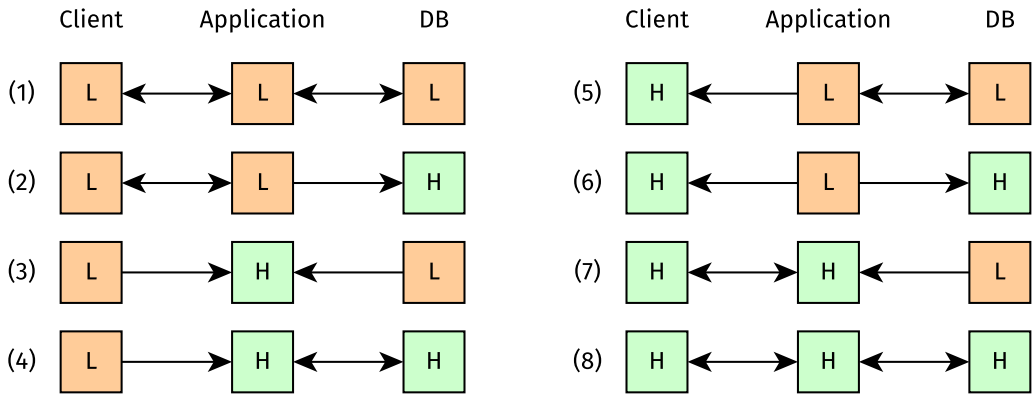
3.3.2 Application Server ⟷ Database

The link between application server and database is usually not as sensitive, as the information stays in many cases under the control of the same party. Reasonable scenarios for this link include for example that the ASP wants to control whether information processed by the application server may or may not be persistently stored in the database (for example credit card security codes, for which persistent storage is not allowed) or to ensure that the integrity of information within the database is not affected by the application server (which can however be more efficiently implemented with traditional access control measures for databases instead of IFC).

3.3.3 Isolation Variants

Figure 3.2 illustrates all possible combinations for IFC policies on the level of tiers, where an upper bound of the permitted security classes is assigned to each tier. The arrows illustrate the flows of

Figure 3.2: Secure information flows between tiers.



information as permitted by the flow relation introduced in Section 3.2. Note that a missing arrow does not mean that there is no information flow at all: since the shown security class is an upper bound, a tier can also contain information of the lower security class L. Information of security class L is always allowed to flow between tiers, as evident in case 1 of the figure, and even to third parties.

The selection of the upper bounds corresponds to the trust that the ASP grants each tier, which in turn depends on the use case that the application serves. Not every possible case of Figure 3.2 results necessarily in a useful policy. The following discussions neglect the two trivial cases 1 and 8. This leaves us with six remaining cases, for which we examine their meaning in the context of confidentiality and integrity.

First, we can interpret the cases of Figure 3.2 as a policy that specifies the *confidentiality* of the information processed by each tier, where H stands for confidential and L for public (see also the examples from Section 2.3.2, which use the same policy). One possible goal for a policy is *isolation*, where a tier handling confidential data is not supposed to leak this to the neighboring tiers. Case 5 resembles a policy of client isolation, which is suitable to prevent sensitive information stored on the client to leak to the server-side tiers. A possible example for this is a password meter, which should under no circumstances leak the password to any other party. The opposite of this is case 4, where confidential information resides on the server-side and should not be disclosed to the client. The model allows also more granular isolation on the server-side, as shown by cases 2 and 3. They might be helpful in special cases where different application servers share the same database connection, but are otherwise too restrictive and better solved with traditional access control.

Second, the cases can be interpreted in terms of *integrity*, since integrity is dual to confidentiality [3]. In this context, security class L expresses high integrity and security class H low integrity. One imaginable use case is to prevent client data from influencing certain server-side calculations (cases 5 and 6), which helps to enforce the invisible *security barrier* that can be drawn between client and server [17]. Case 7 represents another use case, where information stored in the database is protected from changes that originate from the other tiers. Case 4 allows to isolate data in the client from changes introduced by the server-side.

Table 3.1: Information flow through functions.

Activity	Flow direction for Parameters	Flow direction for Return Value
Calling another function	Outgoing	Incoming
Being called	Incoming	Outgoing

3.4 Flows between Functions

The previous section discussed coarse-grained policies on the tier-level, which mostly reflected the trust of the involved parties for certain classes of information. We now take a look at what is going on inside a tier, which is mostly about the trust that is put into individual functions.

3.4.1 Storage Objects

Section 2.3 talks about storage objects being used in IFC, which are bound to security classes by a policy. To make this less abstract, we take the example of the storage objects inside the tiers supported by the prototype implementation JSLINQ, which was implemented as part of this thesis:

- Database: Security classes are bound to length of a table as well as the content of table columns, so they can be seen as storage objects. For our purpose do we neglect the table length and always assume it to be of security class L.
- Client and Application Server: Programs for JSLINQ are written in the functional programming language F#. Such programs consist mostly of function definitions that in turn make use of other functions. It is thus natural to treat both, the arguments that are passed to a function and the return value, as storage objects.

We will see later that this is reflected in the policy language used by JSLINQ and that policies on function definitions constitute the majority of policy definitions. This is due to the fact that function definitions are involved in all information transfer that happens between tiers. Data is exchanged either by one component calling the functions of another one or reversely by being called. This determines the direction of the information flows, as shown in Table 3.1. In the case of web applications, it is usually the client tier that initiates function calls towards the application server or third-party services.

3.4.2 Expressing Isolation Policies

Tier-level policies as introduced in Section 3.3 are a method to model isolation policies. They can however not be directly turned into an IFC policy, since they do not specify individual storage objects. It is thus necessary to translate a tier-level policy down to the more detailed level of storage objects, which eventually means to specify policies for individual functions. The same applies to the isolation of a tier from other components, such as the underlying operating system, which is not part of the IFC system. For each function call it is thus necessary to decide if it violates the isolation policy. If a policy for example distrusts all third-parties, this means that the parameters of all functions that result in data transmission to third parties must be identified and annotated accordingly. Similarly, if a client needs to be isolated from the application server, the parameters of all RPC functions must be annotated. There are also other functions which allow information to leave the IFC system, such as for example the interface towards the operating system (e.g. file operations to the local disk), which have to be captured as well. Considering all of the above, it is clear that ensuring these properties for all functions is a laborious task, especially when looking at the all interface functions that a modern

operating system and web application framework provide. One way to handle this for the purpose of demonstration is to strongly limit the amount of functions that are made available to the application. The JSLINQ prototype thus only allows the use of a limited subset of functions provided by F# and WebSharper.

Chapter 4

Formal Model

This chapter gives an overview of the theoretical foundation that are backing the implementation of JSLINQ (which will be explained in the next chapter). It is based on the formal model and explanations given in [2] and focuses only on those parts that are relevant for implementation. Since the source paper is not publicly available at the time of this writing, are the necessary definitions included in full detail. The formal model can be roughly divided into four parts as illustrated in Figure 4.1, where each arrow can be read as "contributes to". As visible in the figure, do syntax and semantics as well as the security type system directly influence the implementation. The others do not directly influence the implementation but help to gain confidence in the soundness of the underlying model.

4.1 Syntax and Semantics

Initially, the target programming language is formalized using methods from programming language theory. This requires to exactly formulate the expressions of the language and their behavior, referred to as *syntax and semantics* of the language. A common method to avoid unnecessary complexity in this stage is to support only the essential expressions from the language, resulting in a core language. Development in the core language is not very convenient, since namespaces and syntactic sugar are not available, but it greatly simplifies the formal reasoning that follows in the subsequent steps. In the case of JSLINQ, the formal model corresponds to a simplified version of the F# language with

Figure 4.1: Components of the formal model.

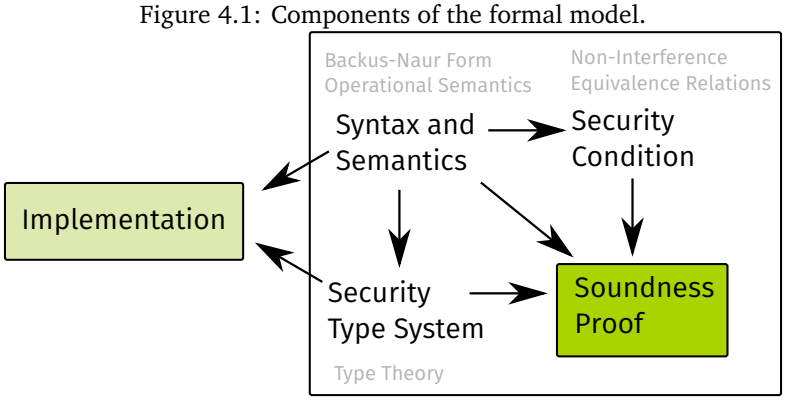


Figure 4.2: F# subset of the formal model. Borrowed from [2].

$\ell ::=$ `L` | `H` (security types)

$b ::=$ $\mathbf{bool}^{\ell}$ | $\mathbf{int}^{\ell}$ | $\mathbf{float}^{\ell}$ | $\mathbf{string}^{\ell}$ (base types)

$t ::= b \mid \mathbf{unit} \mid t \xrightarrow{\ell} t \mid t\ \mathbf{ref}^{\ell} \mid t * t \mid \{\overline{f : t}\} \mid (t\ \mathbf{list})^{\ell} \mid \mathbf{Expr}\langle t \rangle$ (general types)

$T ::= (\{\overline{f : b}\})\ \mathbf{list}^{\ell}$ (database tables)

$$
\begin{aligned}
e ::= {} & () \mid c \mid x \mid l \mid op(\overline{e}) \mid \mathbf{lift}\ e \mid \mathbf{fun}(x) \rightarrow e \text{ (terms)} \\
\mid {} & \mathbf{rec}\ f(x) \rightarrow e \mid (e, e) \mid \mathbf{fst}\ e \mid \mathbf{snd}\ e \mid \{\overline{f = e}\} \mid e.f \\
\mid {} & \mathbf{yield}\ e \mid [] \mid e\ @\ e \mid \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ e \mid \mathbf{exists}\ e \\
\mid {} & \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e \mid \mathbf{if}\ e\ \mathbf{then}\ e \mid \mathbf{run}\ e \mid \texttt{<@}\ e\ \texttt{@>} \mid \texttt{(\%}\ e\ \texttt{)} \\
\mid {} & \mathbf{database}(x) \mid \mathbf{ref}\ e \mid\ !e \mid e := e
\end{aligned}
$$

the addition of security levels to the syntax of types, as shown in Figure 4.2. The notation is using the following abbreviations [2]:

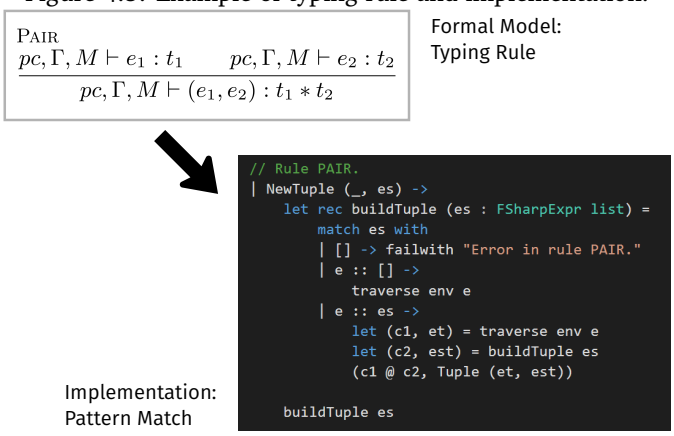
c	constant values, as for example definitions like `let foo = 42`.
op	built-in operators, as for example mathematical operators.
f	name of a field, as used in F# records.
x	a variable.
$\overline{x}$	a sequence of entities x.
l	a memory location.

This language features custom additions to standard F# which are necessary to provide a simplified model of how database interactions are expressed: **database** and **run** are used to open the database and run queries, **exists** is used to check if a database query returns an empty result set and **lift** is used to turn an expression into a query. We omit a detailed discussion of those, as their primary purpose is to allow formal reasoning and the implementation aims to use the native facilities for database interaction as much as possible. Similarly are quotes (`<@` e `@>`), anti-quotes (`%`e), the **if**-condition without an else branch and the keyword **yield** are only intended for the purpose of expressing database queries. Quotes are not further discussed, since the prototype uses the more intuitive query expressions as supported by LINQ [24]. A complete description of the language also requires inclusion of the operational semantics. Their discussion is however skipped in this report, since the prototype directly relies on the original semantics of the F# language. This greatly simplifies implementation but comes with the price of a bigger gap between formal model and implementation.

4.2 Security Condition

The second step is to formulate a *security condition*, a formal definition of what security exactly means in this model. JSLINQ's security condition is based on a widely used property called *non-interference* that was defined in 1982 by Goguen and Meseguer [14]. It essentially requires that data of one class

Figure 4.3: Example of typing rule and implementation.



of information (e.g. confidential or untrusted) can never have an influence (as small as it might be) on data of another class of information (e.g. public or trusted). A model satisfying non-interference never discloses secrets to an attacker and never lets the attacker modify trusted information. Real-life applications do however usually need to disclose at least small parts of the secret, as shown in Section 2.3.3. This is the reason why the formal model additionally relies on another property called *delimited release* [25].

4.3 Security Type System

As a third step a *security type system* is specified. It makes use of type theory as used in programming language theory. It consists of a set of logical rules that specify for each expression the resulting security type. The security type system exists parallel to the "normal" type system for data types, with which it shares the theoretical foundations. The model splits the language into a *host language*, which resembles normal top-level F# expressions and a *quoted language*, which is used inside F# quotations and represents database queries. The rules are shown in Figures 4.4 and 4.5 respectively, while Figure 4.6 shows rules for the comparison of security levels and security types. The following symbols and abbreviations are used:

pc	security level of the program counter. Used to track implicit information flows and side-effects.
Γ	typing context mapping variables to types.
M	typing context mapping memory locations to types.
$\ell \bigsqcup \ell'$	join of two levels. Returns the highest security level that occurs on one of the sides.
H	shorthand notation for pc, Γ, M (typing context of the host language).
Δ	typing context of the quoted language.

The distinction between host and quoted language is mainly of importance for the formal reasoning and has thus not a big influence on the implementation. This can be seen by the fact that the typing rules for both are almost the same, with the quoted language most notably missing the

security variables for side-effects. Translating the security type system into code resembles the main contribution of this thesis, besides solving the involved engineering challenges. Rules are turned into a pattern match that is considered while the abstract syntax tree is traversed. Figure 4.3 gives one example for this.

Figure 4.4: Typing rules for the host language. Borrowed from [2].

$$\textsc{Const}\ \frac{\Sigma(c) = t}{pc, \Gamma, M \vdash c : t^{\ell}} \qquad \textsc{Unit}\ \frac{}{pc, \Gamma, M \vdash () : \mathbf{unit}} \qquad \textsc{Var}\ \frac{x : t \in \Gamma}{pc, \Gamma, M \vdash x : t} \qquad \textsc{Loc}\ \frac{l : t \in M}{pc, \Gamma, M \vdash l : t}$$

$$\textsc{Nil}\ \frac{}{pc, \Gamma, M \vdash [] : (t\ \mathbf{list})^{\ell}} \qquad \textsc{Fun}\ \frac{pc, \Gamma, x : t, M \vdash e : t'}{pc', \Gamma, M \vdash \mathbf{fun}(x) \to e : (t \xrightarrow{pc} t')} \qquad \textsc{Rec}\ \frac{pc, \Gamma, x : t, f : t \xrightarrow{pc} t', M \vdash e : t'}{pc', \Gamma, M \vdash \mathbf{rec}\ f(x) \to e : t \xrightarrow{pc} t'}$$

$$\textsc{Lift}\ \frac{pc, \Gamma, M \vdash e : t}{pc, \Gamma, M \vdash \mathbf{lift}\ e : \mathbf{Expr}\langle t \rangle} \qquad \textsc{Exists}\ \frac{pc, \Gamma, M \vdash e : (t\ \mathbf{list})^{\ell}}{pc, \Gamma, M \vdash \mathbf{exists}\ e : \mathbf{bool}^{\ell}} \qquad \textsc{Op}\ \frac{\Sigma(op) = \overline{t} \to t \qquad \overline{pc, \Gamma, M \vdash e : t^{\ell}}}{pc, \Gamma, M \vdash op(\overline{e}) : t^{\bigsqcup \ell_i}}$$

$$\textsc{Yield}\ \frac{pc, \Gamma, M \vdash e : t}{pc, \Gamma, M \vdash \mathbf{yield}\ e : (t\ \mathbf{list})^{\ell}} \qquad \textsc{Apply}\ \frac{pc, \Gamma, M \vdash e_1 : t \xrightarrow{pc'} t' \qquad pc, \Gamma, M \vdash e_2 : t \qquad pc \sqsubseteq pc'}{pc, \Gamma, M \vdash e_1\ e_2 : t'}$$

$$\textsc{Pair}\ \frac{pc, \Gamma, M \vdash e_1 : t_1 \qquad pc, \Gamma, M \vdash e_2 : t_2}{pc, \Gamma, M \vdash (e_1, e_2) : t_1 * t_2} \qquad \textsc{Fst}\ \frac{pc, \Gamma, M \vdash e : t_1 * t_2}{pc, \Gamma, M \vdash \mathbf{fst}\ e : t_1} \qquad \textsc{Snd}\ \frac{pc, \Gamma, M \vdash e : t_1 * t_2}{pc, \Gamma, M \vdash \mathbf{snd}\ e : t_2}$$

$$\textsc{Record}\ \frac{\overline{pc, \Gamma, M \vdash e : t}}{pc, \Gamma, M \vdash \{\overline{f = e}\} : \{\overline{f : t}\}} \qquad \textsc{Project}\ \frac{pc, \Gamma, M \vdash e : \{\overline{f : t}\}}{pc, \Gamma, M \vdash e.f_i : t_i} \qquad \textsc{Union}\ \frac{pc, \Gamma, M \vdash e : (t\ \mathbf{list})^{\ell} \qquad pc, \Gamma, M \vdash e' : (t\ \mathbf{list})^{\ell'}}{pc, \Gamma, M \vdash e'\ @\ e : (t\ \mathbf{list})^{\ell \sqcup \ell'}}$$

$$\textsc{For}\ \frac{pc, \Gamma, M \vdash e : (t\ \mathbf{list})^{\ell} \qquad pc, \Gamma, x : t, M \vdash e' : (t'\ \mathbf{list})^{\ell'}}{pc, \Gamma, M \vdash \mathbf{for}\ x\ \mathbf{in}\ e\ \mathbf{do}\ e' : (t'\ \mathbf{list})^{\ell \sqcup \ell'}} \qquad \textsc{If1}\ \frac{pc, \Gamma, M \vdash e : \mathbf{bool}^{\ell} \qquad pc, \Gamma, M \vdash e' : (t\ \mathbf{list})^{\ell'}}{pc, \Gamma, M \vdash \mathbf{if}\ e\ \mathbf{then}\ e' : (t\ \mathbf{list})^{\ell \sqcup \ell'}}$$

$$\textsc{If}\ \frac{pc, \Gamma, M \vdash e : \mathbf{bool}^{\ell} \qquad pc \sqcup \ell, \Gamma, M \vdash e_i : t \qquad \ell \sqsubseteq t \qquad i \in \{1, 2\}}{pc, \Gamma, M \vdash \mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2 : t} \qquad \textsc{Deref}\ \frac{pc, \Gamma, M \vdash e : t\ \mathbf{ref}^{\ell} \qquad \ell \sqsubseteq t}{pc, \Gamma, M \vdash !e : t}$$

$$\textsc{Quote}\ \frac{pc, \Gamma, M, \cdot \vdash e : t}{pc, \Gamma, M \vdash \texttt{<@}\ e\ \texttt{@>} : \mathbf{Expr}\langle t \rangle} \qquad \textsc{Sub}\ \frac{t \sqsubseteq t' \qquad pc, \Gamma, M \vdash e : t}{pc, \Gamma, M \vdash e : t'} \qquad \textsc{Run}\ \frac{pc, \Gamma, M \vdash e : \mathbf{Expr}\langle t \rangle}{pc, \Gamma, M \vdash \mathbf{run}\ e : t}$$

$$\textsc{Ref}\ \frac{pc, \Gamma, M \vdash e : t \qquad pc \sqsubseteq t}{pc, \Gamma, M \vdash\ \mathbf{ref}\ e : t\ \mathbf{ref}^{pc}} \qquad \textsc{Assn}\ \frac{pc, \Gamma, M \vdash e_1 : t\ \mathbf{ref}^{\ell} \qquad pc, \Gamma, M \vdash e_2 : t \qquad pc \sqcup \ell \sqsubseteq t}{pc, \Gamma, M \vdash e_1 := e_2 : \mathbf{unit}}$$

Figure 4.5: Typing rules for the quoted language. Borrowed from [2].

$$\textsc{ConstQ}\ \frac{\Sigma(c) = t}{H, \Delta \vdash c : t^{\ell}} \qquad \textsc{FunQ}\ \frac{H, \Delta, x : t \vdash e : t'}{H, \Delta \vdash \mathbf{fun}(x) \to e : t \to t'} \qquad \textsc{VarQ}\ \frac{x : t \in \Delta}{H, \Delta \vdash x : t} \qquad \textsc{ApplyQ}\ \frac{H, \Delta \vdash e_1 : t \to t' \qquad H, \Delta \vdash e_2 : t}{H, \Delta \vdash e_1\ e_2 : t'}$$

$$\textsc{OpQ}\ \frac{\Sigma(op) = \overline{t} \to t \qquad \overline{H, \Delta \vdash M : t^{\ell}}}{H, \Delta \vdash op(\overline{M}) : t^{\bigsqcup \ell_i}} \qquad \textsc{Antiquote}\ \frac{H \vdash e : \mathbf{Expr}\langle t \rangle}{H, \Delta \vdash (\%\ e\ ) : t} \qquad \textsc{PairQ}\ \frac{H, \Delta \vdash e_1 : t_1 \qquad H, \Delta \vdash e_2 : t_2}{H, \Delta \vdash (e_1, e_2) : t_1 * t_2}$$

$$\textsc{FstQ}\ \frac{H, \Delta \vdash e : t_1 * t_2}{H, \Delta \vdash \mathbf{fst}\ e : t_1} \qquad \textsc{SndQ}\ \frac{H, \Delta \vdash e : t_1 * t_2}{H, \Delta \vdash \mathbf{snd}\ e : t_2} \qquad \textsc{RecordQ}\ \frac{\overline{H, \Delta \vdash M : t}}{H, \Delta \vdash \{\overline{f = M}\} : \{\overline{f : t}\}} \qquad \textsc{ProjectQ}\ \frac{H, \Delta \vdash L : \{\overline{f : t}\}}{H, \Delta \vdash L.f_i : t_i}$$

$$\textsc{YieldQ}\ \frac{H, \Delta \vdash M : t}{H, \Delta \vdash \mathbf{yield}\ M : (t\ \mathbf{list})^{\ell}} \qquad \textsc{NilQ}\ \frac{}{H, \Delta \vdash [\,] : (t\ \mathbf{list})^{\ell}} \qquad \textsc{ExistsQ}\ \frac{H, \Delta \vdash M : (t\ \mathbf{list})^{\ell}}{H, \Delta \vdash \mathbf{exists}\ M : \mathbf{bool}^{\ell}}$$

$$\textsc{IfQ}\ \frac{H, \Delta \vdash L : \mathbf{bool}^{\ell} \qquad H, \Delta \vdash M : (t\ \mathbf{list})^{\ell'}}{H, \Delta \vdash \mathbf{if}\ L\ \mathbf{then}\ M : (t\ \mathbf{list})^{\ell \sqcup \ell'}} \qquad \textsc{UnionQ}\ \frac{H, \Delta \vdash M : (t\ \mathbf{list})^{\ell} \qquad H, \Delta \vdash N : (t\ \mathbf{list})^{\ell'}}{H, \Delta \vdash N\ @\ M : (t\ \mathbf{list})^{\ell \sqcup \ell'}}$$

$$\textsc{ForQ}\ \frac{H, \Delta \vdash M : (t\ \mathbf{list})^{\ell} \qquad H, \Delta, x : t \vdash N : (t'\ \mathbf{list})^{\ell'}}{H, \Delta \vdash \mathbf{for}\ x\ \mathbf{in}\ M\ \mathbf{do}\ N : (t'\ \mathbf{list})^{\ell \sqcup \ell'}} \qquad \textsc{SubQ}\ \frac{t \sqsubseteq t' \qquad H, \Delta \vdash M : t}{H, \Delta \vdash M : t'} \qquad \textsc{DatabaseQ}\ \frac{\Sigma(db) = \{\overline{f : t}\}}{H, \Delta \vdash \mathbf{database}(db) : \{\overline{f : t}\}}$$

Figure 4.6: Security annotation constraints. Borrowed from [2].

$$\frac{\ell \sqsubseteq \ell'}{\ell \sqsubseteq t^{\ell'}} \qquad \frac{}{\ell \sqsubseteq \mathbf{unit}} \qquad \frac{\ell \sqsubseteq pc \qquad \ell \sqsubseteq t}{\ell \sqsubseteq t' \xrightarrow{pc} t} \qquad \frac{\ell \sqsubseteq t_1 \qquad \ell \sqsubseteq t_2}{\ell \sqsubseteq t * t} \qquad \frac{\ell \sqsubseteq t_i}{\ell \sqsubseteq \{\overline{f : t}\}} \qquad \frac{\ell \sqsubseteq t}{\ell \sqsubseteq \mathbf{Expr}\langle t \rangle}$$

4.4 Soundness Proof

The final *soundness proof* combines all aforementioned components: for a program written in the core language that complies with the security type system, it guarantees that the security condition (i.e. non-interference) is preserved. This guarantee is very powerful and applies to all possible execution paths of the program. Details about the proof technique are provided in [2] and beyond the scope of this report.

Chapter 5

Implementation

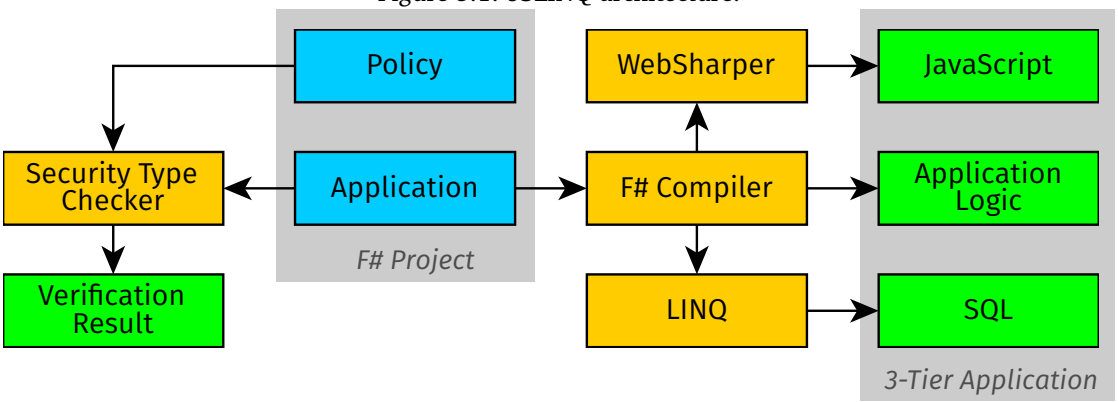
5.1 Architecture

Figure 5.1 shows the architecture of JSLINQ. The input is an F# project consisting of the security policy and the application code. The right branch of the figure shows how a project is first compiled to a three-tier application using the unmodified build process for web applications based on WebSharper. The code of the project is used to create a 3-tier application consisting of JavaScript created using WebSharper, .NET assemblies for server-side application logic and SQL queries for the database, created using LINQ. Upon successful compilation, JSLINQ's security type checker can be used on the F# project in order to determine if the application complies with the specified information flow policy. How the resulting 3-tier application and the verification result are used depends on the use case of JSLINQ: one possibility is to discard non-compliant application builds and to deploy compliant applications into production. The remainder of the section discusses JSLINQ components in more detail.

5.1.1 WebSharper

A key element in supporting client-side code in JSLINQ is the WebSharper library. WebSharper is a fully-featured and commercially supported framework for web application development in F#, pro-

Figure 5.1: JSLINQ architecture.



viding powerful functional abstractions such as sitelets for document definition, formlets for data entry forms and flowlets for workflows [15]. Moreover, it offers abstractions for essential web concepts such as the DOM or JavaScript code. Importantly, these abstractions enjoy type safety properties, allowing to leverage the F# type system to build robust applications. One of WebSharper's key features is the translation of F# functions into JavaScript code for execution in the browser. Server-side functions can be designated as remote procedure call (RPC) functions, so that they can be transparently called in client-side code, as shown in the following example:

```
// Server-side function called by the client via AJAX.
[<Remote>]
let getText () = "JSLINQ"

// Client-side function translated to JavaScript.
[<JavaScript>]
let Main () = Text (getText ())
```

WebSharper supports extension of the client with third-party libraries, for example a map service. Third-party libraries usually consist of JavaScript code that is embedded into the page. Calls from the client-side F# code to the embedded third-party library are handled by wrappers that provide an F# interface to the JavaScript code. This approach requires full trust on the JavaScript code provided by the third party. However, JSLINQ can be used to type-check third-party libraries written in F#. This allows rewriting crucial third-party JavaScript libraries in F# to make them amenable to security analysis using JSLINQ.

5.1.2 F# Project

JSLINQ is designed to perform the verification step after successful compilation of the project. This means in terms of Figure 5.1 that a successful build using the right branch is mandatory before running the left verification branch. JSLINQ processes MSBuild projects, which allows it to integrate with Microsoft Visual Studio. Code within a project is either part of the policy or part of the program that implements the application. The policy controls information flow via security type signatures which are added to the definitions of functions and databases. The program implements the application and is subject to the security type check according to the policy. Since the policy is expressed within normal F# syntax, the use of JSLINQ does not interfere with the normal build process of the application and the use of standard tools.

5.2 Policy Definition

The policy is specified by adding custom attributes with security type signatures to declarations. Signatures are represented as strings that follow the security type language which is part of the formal model (see Section 4), with the addition of variables for security levels in order to support polymorphism between those levels. If no security level is specified within a signature, the corresponding level variable is unconstrained. The following code fragment demonstrates how signatures are added to F# declarations:

```
[<SecT("_^H")>]
let boolH = true

[<SecT("unit ->^L  _^L")>]
let f () = 1
```

The brackets are required by F#'s syntax for attributes, while `SecT` is the name of the custom .NET attribute that requires a string which contains the actual security type signature. The attributes shown in the code snippet above correspond to the signatures $boolH : bool^H$ and $f : unit \xrightarrow{L} int^L$ in the syntax from the formal model. The syntax of security types is similar to F#'s own syntax for data types, with the following important differences:

- Base types are not distinguished and always written as an underscore (`_`). Thus a function `float -> int` is written as `_ -> _`. A special case are WebSharper's `Pagelet` and `Element` types, which are also expressed as underscore. They are required to support functions that build user interfaces. This simplification is done because the data type is already determined by F#'s type inference, so that it does not need to be repeated in the security type signature.
- Security levels can be one of two levels (`H`, `L`) or variables (`'a`, `'foo` etc.). They are written as raised expressions in the signature: `_^H ->^L _^L`.
- The use of security levels is optional. Unspecified levels result in an unconstrained fresh level variable. Thus `_ -> _` is equivalent to `_^'a -> _^'b`.

We divide a web application policy into three types: a library policy, an RPC policy and a database policy. Each type of policy deals with different application tiers and the meaning of a security type signature depends on the part in which it is located.

The policy for library functions is defined in a separate module, which is marked with a policy attribute. All library functions used by the program (with a few exceptions such as core F# operators) need to be wrapped in the policy, otherwise their use is not allowed. Since HTML and JavaScript abstractions of WebSharper are also library functions, the policy for client-side functionality is specified in this part. Each wrapper function has a mandatory security type signature that governs which security levels are used when the wrapper is called. The following snippet demonstrates a wrapper that uses WebSharper functions to generate a masked input field for passwords, labelled as high:

```
[<Policy>]
module Policy =
  [<JavaScript>]
  [<SecT("unit -> _^H")>]
  let InputPW () = Input [Attr.Type "password"]
```

The policy for remote procedure calls (RPC) from the client to the server consists of attributes to the declarations of RPC functions within the program. In our implementation, the RPC policy and the program are defined in the same file for the sake of simplicity. However, JSLINQ allows a complete separation of policy and program into separate files, as it is done for the other parts of the policy. Type signatures on RPC functions restrict the information flow from the client to the server (via function arguments) and from the server to the client (via return values). The following fragment demonstrates flows in both directions between client and application server:

```
[<Remote>]
[<SecT("unit -> _^L")>]
let untrustedClient () = true

[<Remote>]
[<SecT("_^L ->^L unit")>]
let untrustedServer (x:bool) = ()
```

The database policy is defined by adding security type signatures to an attribute-based mapping for LINQ [1]. Security type signatures are added to table and column definitions as shown in the following example:

```
[<Table>]
[<SecT("_^L")>] // Public table length
type Account =
  [<Column>]
  [<SecT("_^L")>] // Public username
  abstract member Username : string

  [<Column>]
  [<SecT("_^H")>] // Confidential password
  abstract member Password : string
```

5.3 Security Type Checker

The design of JSLINQ as an additional verification step after compilation allows us to assume that the analyzed code has correct syntax, data types and satisfied dependencies, which allows the implementation to only focus on the actual security type check. Noteworthy, we leave the existing F# type system untouched and maintain a completely separate security type system during verification. Security type checking is performed in two steps, which are repeated for each top-level declaration found in the code of the project: first the abstract syntax tree (AST) for the declaration is recursively traversed, which yields a set of constraints and a security type signature. User-defined security type signatures are parsed with the FParsec library [13]. The second step substitutes level variables with actual security levels by solving the constraint set. The resulting types and possible remaining constraints are added to the environment before proceeding with the next declaration. JSLINQ uses the AST generated by the F# compiler, which is retrieved using the library FSharp Compiler Services [12]. We thus do not have to duplicate compiler features that are unrelated to the security type check and benefit automatically from F#'s desugaring. This is a clear advantage over prototypes that enhance existing type systems, for instance SELINQ or SIF.

Chapter 6

Case Studies

We have used JSLINQ to implement different case studies as F# projects. In this section we first describe the general design of the policy language and then remark on the policy requirements for the case studies that we have implemented.

6.1 Library Policy

The largest part of the library policy are the security type signatures for WebSharper's DOM and JavaScript abstractions. The documents shown in the browser are constructed using these abstractions at runtime. For simplification, we consider the HTML elements as trusted sinks. The rationale behind this is that the user has full access to the data once it has arrived in the browser, independently of that data being displayed or not. However, this assumption does not hold for the full WebSharper API, as it would allow to write and read the elements in the DOM tree in various ways. Therefore, the policy only permits basic operations on the DOM. An important exception from our trusted sink assumption are HTML elements which load external resources, such as images and IFrames. These elements can be used to leak data either directly within the source attribute or indirectly via externally observable HTTP requests. Therefore, we annotate the creation of the source attribute with low security level, both for the URL argument and the side-effects. The library policy for the case studies has in total three functions that are supposed to be able to leak information to third-parties:

1. The HTML source attribute, for the reasons mentioned above.
2. Initialization of a Google Map, as it results in externally observable requests.
3. Panning of a Google Map, for the same reason as the initialization and additionally the transmission of the coordinate to which the map is panned.

The corresponding security type signatures are expressed in the policy code as follows:

```
[<SecT("_^L ->^L _")>]
let Src (x:string) = ...

[<SecT("unit ->^L _^L")>]
let InitGoogleMapDiv () = ...

[<SecT("_^L -> _^L ->^L unit")>]
let PanGoogleMap lat lon = ...
```

6.2 Scenario Discussion

We now comment on different aspects of the policy and provide examples for vulnerabilities that are captured by JSLINQ, for each scenario.

6.2.1 Password Meter

We have included the password meter to demonstrate a policy with full client isolation, where the password is not allowed to leak from the browser at all. The policy declares password fields as sensitive sources. Leaks to third parties and to the application server are prevented by assigning low levels to the source attribute and to the arguments and side-effects of RPC functions, respectively. The scenario assumes that the server is untrusted, as it should not receive the password. Looking at tier-level trust, this corresponds to case 5 of Figure 3.2. A problem with this view is that the JavaScript code executed by the client is usually delivered by the same or another untrusted server. This means that the integrity of the client-side code after the security type check is not guaranteed. Such changes outside the normal build process are not subject to the security policy and can thus be abused to leak confidential data. Therefore we have to put trust in the integrity of the code delivered by the application server, which we summarize as *partial trust*. Alternatively, remote attestation methods such as code or certificate signatures can be used to remove this assumption. The following code snippets show a password check that is accepted by JSLINQ and two leaks via the source attribute that are correctly blocked.

```
let content = // Allowed: Secret only in browser.
  if (containsLetters password)
  then Text "Passed" else Text "Failed"

let content' = // Blocked: Leak via source attribute.
  Image [Src ("http://example.com/img.png?" + password)]

// Blocked: Leak via side effects.
let content'' = Src (if secret == "jSL!Nq42"
  then "http://example.com/true.jpg"
  else "http://example.com/false.jpg")
```

This application consists of 53 F# and 6215 generated JS LOCs.

6.2.2 Location-Based Service (LBS)

This scenario demonstrates declassification of a client-side secret, in this case the user's position. The position is considered secret because of the following declaration on the library policy, which marks all data received from the callback function as confidential:

```
[<SecT("(_^H -> _^H -> unit) -> unit")>]
let GetPosition (callback : float -> float -> unit) = ...
```

Third parties and the application server may only receive declassified obfuscated coordinates. Our implementation performs declassification by defining a function that adds a random offset to a float value. The function is part of the library policy and declared as follows (implementation details omitted for clarity):

```
[<SecT("_ -> _^L")>]
let addRandomOffset (x : float) = ...
```

Its security type signature means, that this function accepts a float value of any security level (either secret or public) and always returns a float value of low security value (public), which effectively results in declassification of the input value. It is worth to note that such a security type signature may only occur within the policy, using the same security type signature in the untrusted code that is subject to the policy results in a constraint violation. The randomization is applied to the confidential latitude and longitude values. The exact location coordinates are isolated in the browser in the same way as the password in the previous scenario, which is why it also corresponds to case 5 of Figure 3.2. This level of isolation also means that the exact coordinate should not be disclosed to the application server as well, which requires to specify a RPC policy. Thus the only existing RPC function has its parameters marked as public in the security type signature:

```
[<Remote>]
[<SecT("_^L -> _^L ->^L {lat:_; lon:_; note:_; dist:_} list")>]
let GetPois (refLat:float) (refLon:float) = ...
```

We provide two variants of the location-based service to showcase two different attacker models. The first example embeds a map via an IFrame, where the position is an argument to the source attribute of the IFrame. The following snippet shows how the use of declassified coordinates is permitted, while exact coordinates are blocked:

```
let iframeSrc = Src // Allowed: Obfuscated coordinate.
  "https://maps.example.com/?q=" +
  (string (randomize Lat)) + "," + (string (randomize Lon))

let iframeSrc' = Src // Blocked: Exact coordinate.
  "https://maps.example.com/?q=" +
  (string Lat) + "," + (string Lon)
```

The second example includes third-party library called via F#. We use the Google Maps extension for WebSharper and wrap the initialization and panning of the map within the policy, both having low side-effects and low values. Since the extension wraps the original JavaScript code, we have to fully trust the F#-to-JavaScript extension and JavaScript code implementing the WebSharper APIs. The scenario consists of 76 F# and 6279 generated JS LOCs.

6.2.3 Movie Rental

This scenario demonstrates the use of security policies on databases. The database consists of a list of items (e.g. movies) subject to events (e.g. movie rentals) happening at a certain location and time. The location of an event consists of latitude and longitude and it is confidential and all other information is public. The policy is modeled as follows: the database policy assigns to the latitude and longitude a high security level, as expressed by the following column definitions in the database policy (omitting getters and setters for clarity):

```
[<Column>]
[<SecT("_^H")>]
member this.Lat

[<Column>]
[<SecT("_^H")>]
member this.Lon
```

Leaks to the client are prevented by labeling the return values of RPC functions as public. Note how compared to the previously discussed location-based service this time the "L" labels are in the return value. On the level of tiers, this corresponds to case 4 of Figure 3.2. The RPC policy of the only existing RPC functions thus looks as follows:

```
[<Remote>]
[<SecT("_ -> {item:_^L; eventCount:_^L} list^L")>]
let GetAreaRanking (areaIndex : int) = ...
```

The following LINQ query joins rentals with movies and returns a list of movie titles. Movie titles are input to an RPC function which is only allowed to return public values. As a result the first `yield` statement is allowed to return the movie title. If instead we use the second `yield` statement, JSLINQ will reject the program.

```
let events =  query {
    for e in db.Event do
    for i in db.Item do
    if e.ItemId = i.Id then
      (* Allowed *) yield i.Name
      (* Blocked *) yield (string e.Lat) }
```

Moreover, we allow the user to retrieve a ranking of movies that are popular within an area. The implementation contains a pre-defined set of areas which are addressed using indexes. The user is only allowed to specify the index for an area of interest. The application server filters the list of movie rentals based on the coordinate values. JSLINQ will infer a high security level for the length of the resulting list, as it depends on the secret coordinate values. Our policy allows that geographic information about rentals is disclosed on the granularity of fixed-size areas, therefore we can directly declassify the length of that list. The scenario consists of 87 F# and 6231 generated JS LOCs.

6.2.4 Friend Finder App

In this scenario we consider a completely untrusted application server. The client obtains the code from a trusted source. We use the Apache Cordova framework [10] to package the client-side functionality as an app that can be distributed to mobile devices via a trusted channel. Cordova also provides access to the address book of the device. The program can access the address book only via a callback function defined in the policy, whose main purpose is to assign a high security level to the contact details:

```
[<SecT("({Name:_^H; Phone:_^H} list -> unit) -> unit")>]
let GetPhoneContacts (handler : Contact list -> unit) = ...
```

The policy allows declassification by means of a hash function on strings, which allows an arbitrary security level for the resulting string and thus performs declassification:

```
[<SecT("_ -> _")>]
let Hash (s:string) = ...
```

Leakage of plain contact details to the untrusted server is prevented by assigning a low security level to the arguments and side-effects of RPC functions:

```
[<SecT("_^L ->^L _")>]
[<Remote>]
let lookup (hash : string) = ...
```

Overall is this again a client-isolation policy similar to the password example, thus it corresponds to case 5 of Figure 3.2 as well. The following snippet illustrates a secure and an insecure RPC call:

```
// Allowed: Look-up of hashed phone number
let rpcResult = remoteLookup (Hash phoneNumber)

// Blocked: Look-up of plain phone number
let rpcResult' = remoteLookup phoneNumber
```

The scenario has 62 F# and 9966 generated JS LOCs.

6.2.5 Battleship

We implement a simplified version of the classical Battleship game, which has already some history as an IFC example [21, 34]. The client uses the browser to play against the server and both players want hide the exact position of their ships on a grid. Both sides trust each other to correctly follow the rules of the game, so we are only concerned about confidentiality and not integrity. A desirable IFC policy for this game is to mark the values indicating individual ship positions as confidential and all parameters and return values of RPC functions as public, so that confidential information is not allowed to pass the barrier between the browser and the server. This is implemented by setting in the RPC policy the parameters and return values of every RPC function as public:

```
[<Remote>]
[<SecT("unit ->^L {size: _^L; ships: _^L list^L}")>]
let Parameters () = ...

[<Remote>]
[<SecT("unit ->^L {size: _^L; ships: _^L list^L}")>]
let Initialize () = ...

[<Remote>]
[<SecT("{hit:_^L; shot:{x:_^L; y:_^L}; defeated:_^L} ->^L unit")>]
let Report (r:Response) = ...

[<Remote>]
[<SecT("{x:_^L; y:_^L} ->^L {hit:_^L; shot:{x:_^L; y:_^L}; defeated:_^L}")>]
let Play (s:Position) : Response = ...
```

This allows us to re-use the same security policy on both sides, as shown in Figure 6.1. Note that since we have two different kinds of confidential information, it is not possible to map this to one of the tier-level policies from Figure 3.2. The game rules require declassification, since the response to a shot requires disclosure of one bit of information (“hit” or “miss”) to the other player per round. Safe use requires a manual review of every function that makes use of declassification, while all others can remain untouched. This is because only one bit is declassified, so that the meaning depends strongly on the context. An alternative would be to have a declassification function that has greater awareness of the game state. On each side we have to perform declassification twice: firstly for the hit/miss response to a shot, as it directly depends on the presence of a ship at that location, and secondly for indicating to the opponent if a player is defeated, which requires to test all occupied cells. The latter can also be done locally, but for implementation reasons players report their own defeat to the opponent. The following example shows this for the client-side:

Figure 6.1: Symmetric IFC policy for Battleship.

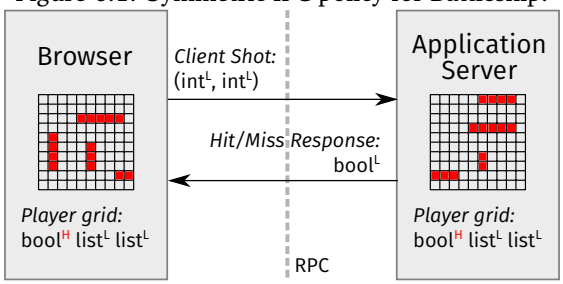


Table 6.1: Overview of implemented scenarios.

	Trust			# of Annotations		
Scenario	Client	3rd Party	Server	API	RPC	DB
Password Meter	Yes	No	Partial	10	0	0
POI IFrame	Yes	No	Yes	10	1	5
POI Embedded	Yes	Yes	Yes	11	1	5
Movie Rental	No	No	Yes	9	1	8
Friend Finder	Yes	No	No	9	1	0
Battleship	Isolated	No	Isolated	12	4	0

```
let serverShotResult =
  {
    shot = response.shot;
    hit = DeclassifyBool !serverTarget.occupied;
    defeated = DeclassifyBool clientDefeated
  }
```

The scenario has 255 F# and 6348 generated JS LOCs.

6.3 Case Study Results

Table 6.1 summarizes our case studies. The different combinations of client, third party and server trust illustrate the attacker models handled by JSLINQ. The battleship scenario is special due to its symmetric policy, which is why we specify the trust for client and server-side separately as *isolated*, meaning that their secrets remain isolated from each other and third-parties. The *partial* server trust of the password meter scenario is used with the meaning as introduced in Section 6.2.1. The initial effort of defining the API policy annotations comes with the benefit of minor burden on application programmer side. The policy for JSLINQ requires only very few annotations within the application code. As reported above, the LOCs for F# and JavaScript refer to the application (excluding comments and blank lines) and wrappers in the policy. The difference between the number of lines in F# code and resulting JavaScript shows WebSharper and its libraries at work. This allows the programmer to focus on the application logic and its security-critical parts (subject to security type check in JSLINQ) while standard boilerplate code is automatically generated by the framework. applications contain considerably more code to offer a better user experience. We omit the verification time, as execution time mostly consists of the compilation required to retrieve the AST. As the security type check is based on a simple constraint solver, we expect it to scale well to larger programs.

Chapter 7

Discussion

7.1 Design Choices

The design of JSLINQ as presented in Section 5 is a result of various design choices. The following sections discuss the alternatives that were not considered for the implementation, thereby illustrating the effort that went into the design of the implementation.

7.1.1 Building upon the SeLINQ prototype

The formal model used by JSLINQ is largely based on SeLINQ, which also includes a proof-of-concept implementation done in Haskell. It was therefore natural to first consider the implementation of JSLINQ as an extension of the already existing code base. SeLINQ performs its own parsing and (data) type checking for an F#-like language in order to be able to introduce security type annotations and the security type check. After successful verification does SeLINQ generate valid F# code, which mainly means the removal of security type information from the source program. While this approach allows SeLINQ to customize all aspects of the language and to stay close to the formal model, full language support requires re-implementation of a significant amount of functionality from the F# compiler. As a prototype, SeLINQ supports therefore only a very limited language subset and uses a simplified type system for data types. Since the technical research provided usable solutions to analyze F# and extend its syntax, the approach of extending SeLINQ was therefore neglected in favor of a new prototype written in F# itself.

7.1.2 Code Quotations and Reflection

F# features a built-in technique for working with the abstract syntax tree which is called code quotations [7]. This technique is primarily intended to be used by programs that transform F# code into another language, as for example WebSharper converts F# definitions into JavaScript code. When the F# compiler generates the assembly during compilation, a special attribute in the source code tells it to also store the abstract syntax tree of the compiled program within the same assembly. This information can then be read and traversed by another program when necessary. We had to neglect this approach, since it does not support anti-quotations. A further motivator was that it is not designed with program verification in mind, while F# Compiler Services state this as one of the main goals.

7.1.3 Security Type Annotations

The design decision to retrieve the abstract syntax tree using F# Compiler Services means that the source code needs to be a valid F# program. This made it necessary to evaluate the different ways in which a policy can be expressed using existing language features. An early on neglected option was to provide the security type signatures in a separate file, as this results in very poor usability and gives rise to various kinds of errors that need to be handled. Another candidate that is closer to the original type system is to use F# type abbreviations, as in the following example:

```
type int_L = int
type int_H = int
```

This does however not work, since F#'s type inference does not really handle them as types that are separate from int. It is therefore necessary to completely bypass F#'s type system, which makes it necessary to find a new method for expressing the additional security type signatures. We focused on .NET attributes, which can be used on top-level function definitions of the F# code and provide a great degree of freedom. One initial idea for attributes was to directly annotate parameters and return values as in the following example:

```
let add ([<H>] a) ([<L>] b) : [<H>] int = a + b
```

This approach does however not allow to add security type signatures to more complex data structures such as lists or records. It thus can only be used in simple cases such as the shown example.

7.1.4 F# Type Providers for Database

F# provides a powerful way for the access to structured data, which is based on a language feature called *type providers* [36]. In essence, type providers are components that dynamically create proxy objects for access to structured information such as databases. Type providers are first invoked when an F# project is loaded into Visual Studio, so that type information is available for auto-completion. But more importantly are they also invoked when the F# compiler compiles the project. The types created by type providers can however not be examined using F# Compiler Services due to their dynamic nature. As JSLINQ relies on the use of F# Compiler Services, this makes it necessary to avoid the use of type providers completely. F# allows to work around this by using attribute-based mappings [1], which coincidentally turns out to be also very useful for expressing the security policy for the tables and columns of the database.

7.2 Applications of JSLINQ

In Section 3.1 did we make the observation that for three-tier web applications only the ASP is in the right position to properly use IFC techniques, as it is the only party that has access to the source code for all tiers. We consequently only consider applications of JSLINQ by the ASP. We see JSLINQ and similar systems primarily as development tools that help the ASP with secure software development. This turns out to be close to a scenario described by Smith [33]. Our underlying assumption is that changes to the implementation happen frequently, while changes to the general business logic and trust relationships modeled in the IFC policy are rare in comparison. In such a case can JSLINQ greatly reduce efforts, as it allows to automatically and frequently verify code before it moves from development into production. By integrating JSLINQ already in the build process and reporting policy violations to the developer in the IDE, can errors be spotted and avoided very early in the implementation process. The focus of this application lies on quality assurance and catching mistakes, but not

necessarily in mistrusting application developers. JSLINQ can also be used to verify code supplied from an untrusted source, a scenario which is for example described by Sabelfeld and Sands [26]. This can be the case when the ASP outsources software development to contractors or otherwise uses an implementation that is provided by a third party (e.g. open source libraries or code snippets). In this case can JSLINQ help to automatically verify that the used code does not secretly violate confidentiality and integrity, thereby avoiding that a trusted developer has to verify the implementation manually.

7.3 Related Work

JSLINQ is closely related to SeLINQ [30], which focuses on information flow between application and database, leaving out the client tier that is present in web applications. SeLINQ's implementation is very close to the formal model and uses the Backus-Naur form of the language to automatically generate lexing and parsing routines with BNFC [4], which are then used to translate the subset into valid F# code. This contrasts with JSLINQ, which has a wider gap between the formal model and the implementation, as the analysis is based on the AST of the full language. JSLINQ is not able to guarantee that this translation preserves the semantics of the formally defined language. JSLINQ provides also less sophisticated support for the database tier, as the support for algebraic data type is for example missing.

SIF by Chong et al. [6] builds upon Jif [21], which in turn builds upon Java. A subset of Java's class library is made available using signatures and wrappers, which add the missing security type information and restrict the functionality in order to preserve the security properties. They make incompatible changes to the Java syntax but stay otherwise within the Java runtime for the execution of the verified applications.

Mettler [20] proposes Joe-E, which allows capability-based verification of security properties for a subset of Java. Its theoretical background is quite different from JSLINQ, but for the implementation do both share many engineering challenges and basic approaches. The work introduces the term *overlay type system* for the refinement and extension of the type system of an existing programming language and provides a tight integration into the base language. Similar to JSLINQ, does Joe-E aim to be compatible to the unmodified base language in order to increase usability. Additional type information is for example added through special Java interfaces that have no effect on normal Java compilation but are evaluated by the Joe-E verifier. Mettler uses *honorary markers* to make a subset of library functions available, whereas wrappers are only used when the behavior of the functions needs to be changed in order to preserve the security properties. He also acknowledges the importance and difficulty of creating correct markers for the library.

Similar to the discussion in Section 3.3 do Swamy et al. [35] also use coarse-grained trust on the level of tiers in their discussion of the FABLE type system. Corcoran et al. [9] use FABLE to extend a previously published language for cross-tier development called LINKS [8] with a security type system. The papers are missing details about the prototype implementation, but a look at the published source code suggests that the authors have started with the LINKS source code and implemented their extensions in it, which is a rather invasive approach.

7.4 Limitations and Future Work

The support for the database in JSLINQ is rather basic, especially compared to SeLINQ. A worthwhile goal is to add the support for algebraic data types to JSLINQ. One step further, would it be beneficial to make processing more efficient by picking up the idea of compiled UDFs (User-Defined Functions, provided by SQL code that is stored and executed on the database) as used by SELINKS.

JSLINQ's prototype focuses on providing a pass/fail result for verification. Usability in the case of failed verification is not a strong point of the current prototype, as it is not easy to pinpoint the expression that caused the violation. We think that a worthwhile implementation goal is to provide developers a better user interface for JSLINQ by giving better feedback about policy violations within VisualStudio, down to the level of highlighting expressions within VisualStudio that are involved in policy-violating information flows.

As explained in Section 7.2, is JSLINQ supposed to be used as a development tool, as which it can verify the implementation. Based on the notion of the more abstract tier-level policies discussed in Section 3.3, one possible future direction is to extend the verification into the design phase by combining it with existing solutions in the field of software engineering. Tier-level policies would thus allow to spot invalid information flows already in the design phase of software development.

JSLINQ does not consider the full complexity of web application development, as evident by the assumptions introduced in Section 3.2 and the simplifications made in the implementation of JSLINQ and the case studies. A look at JSLINQ and the related work shows that IFC prototypes that build upon and remain compatible with an existing language have common problems: library functions are extended with signatures, functionality of library functions is restricted using wrappers and special care needs to be taken so that these steps do not violate the security properties. Further research on how to efficiently increase library coverage and preserve security properties is thus another interesting direction for future work.

Chapter 8

Conclusion

This thesis discusses cross-tier information flow control as an additional building block that helps to increase application security. The obtained results contribute a practical approach for the protection of information in web applications from unauthorized disclosure or modification, as they may result from insecure implementation. It connects IFC techniques with trust relationships between tiers and the parties involved, thereby giving an overview of what IFC means in practice. This is implemented in the next step: starting from an existing security type system, this thesis provides a prototype implementation named JSLINQ that automatically verifies three-tier web applications against an IFC policy, which required to make suitable choices from multiple alternatives. The feasibility of this approach is demonstrated using six different example applications which cover browser-based web applications and a mobile app. The resulting implementation favors practicality, as it integrates into the existing development process for the commercially supported F# language and the WebSharper framework. The experiences made during the development of JSLINQ are consistent with similar prototypes: they show that there are important problems to be solved on the way to a useful development tool, especially when it comes to the complete and safe integration of existing standard libraries.

As it is natural for a prototype, there are many things left for potential future work, such as further improvements in completeness (e.g. language support, library coverage, more sophisticated declassification) and overall usability. Once mature enough, systems similar to JSLINQ could allow the introduction of advanced security mechanisms, such as policies that closely reflect the business logic that stands behind the application. These mechanisms go beyond the capabilities of classical security techniques and contribute to increased application security.

Bibliography

[1] Attribute-Based Mapping. https://msdn.microsoft.com/en-us/library/bb386971.aspx, 2015. Accessed: 2015-05-06.

[2] Musard Balliu, Benjamin Liebe, Daniel Schoepe, and Andrei Sabelfeld. Jslinq: Building secure applications across tiers. 2015. Manuscript submitted for publication.

[3] Kenneth J Biba. Integrity considerations for secure computer systems. Technical report, DTIC Document, 1977.

[4] BNF Converter. http://bnfc.digitalgrammars.com/, 2015. Accessed: 2015-10-12.

[5] Jian Chen and Qiming Huang. Eliminating the impedance mismatch between relational systems and object-oriented programming languages. *Monash University, Australia*, 56, 1995.

[6] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proc. USENIX*, pages 1–16, August 2007.

[7] Code Quotations (F#). https://msdn.microsoft.com/en-us/library/dd233212.aspx, 2015. Accessed: 2015-10-12.

[8] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *FMCO*, pages 266–296, 2006.

[9] Brian J. Corcoran, Nikhil Swamy, and Michael W. Hicks. Cross-tier, label-based security enforcement for web applications. In *SIGMOD Conf.*, pages 269–282, 2009.

[10] Apache Cordova. http://cordova.apache.org/, 2015. Accessed: 2015-04-29.

[11] D. E. Denning and P. J. Denning. Certification of Programs for Secure Information Flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[12] F# Compiler Services. http://fsharp.github.io/FSharp.Compiler.Service/, 2015. Accessed: 2015-04-28.

[13] FParsec. http://www.quanttec.com/fparsec/, 2015. Accessed: 2015-05-06.

[14] J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *IEEE SP*, pages 11–20, April 1982.

[15] Adam Granicz. Functional web and mobile development in F#. In *CEFP*, volume 8606 of *LNCS*, pages 381–406. Springer, 2013.

[16] Google Web Toolkit. http://www.gwtproject.org/, 2015. Accessed: 2015-10-12.

[17] Sverre H. Huseby. *Innocent Code*. Wiley, 2004.

[18] Brian Krebs. Online cheating site AshleyMadison hacked. http://krebsonsecurity.com/2015/07/online-cheating-site-ashleymadison-hacked/, 07 2015. Accessed: 2015-09-28.

[19] LINQ (Language-Integrated Query). http://msdn.microsoft.com/en-us/library/bb397926.aspx, 2015. Accessed: 2015-10-12.

[20] Adrian Matthew Mettler. *Language and Framework Support for Reviewably-Secure Software Systems*. PhD thesis, UC Berkeley, 2012.

[21] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nystrom. Jif: Java Information Flow. Software release. http://www.cs.cornell.edu/jif, July 2001.

[22] Ellen Nakashima. Chinese breach data of 4 million federal workers. http://wapo.st/1JpiimW, 06 2015. Accessed: 2015-09-28.

[23] OWASP Top 10 2013. https://www.owasp.org/index.php/Top_10_2013-Top_10/, 2015. Accessed: 2015-05-03.

[24] Query Expressions. https://msdn.microsoft.com/en-us/library/hh225374.aspx, 2015. Accessed: 2015-10-26.

[25] A. Sabelfeld and A. C. Myers. A Model for Delimited Information Release. In *ISSS*, volume 3233 of *LNCS*, pages 174–191, 2003.

[26] A. Sabelfeld and D. Sands. A Per Model of Secure Information Flow in Sequential Programs. *Higher Order and Symbolic Computation*, 14(1):59–91, March 2001.

[27] A. Sabelfeld and D. Sands. Declassification: Dimensions and Principles. *J. Computer Security*, 17(5):517–548, January 2009.

[28] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE JSAC*, 21(1):5–19, 2003.

[29] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[30] Daniel Schoepe, Daniel Hedin, and Andrei Sabelfeld. SeLINQ: tracking information across application-database boundaries. In *ICFP*, pages 25–38. ACM, 2014.

[31] Heiko Schuldt. Multi-Tier architecture. In *Encyclopedia of Database Systems*, pages 1862–1865. Springer US, 2009.

[32] Vincent Simonet. The Flow Caml system. Software release. http://cristal.inria.fr/~simonet/soft/flowcaml, 2003.

[33] Geoffrey Smith. Principles of secure information flow analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307. Springer, 2007.

[34] Alley Stoughton, Andrew Johnson, Samuel Beller, Karishma Chadha, Dennis Chen, Kenneth Foner, and Michael Zhivich. You sank my battleship!: A case study in secure programming. In *PLAS@ECOOP 2014*, page 2. ACM, 2014.

[35] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A Language for Enforcing User-defined Security Policies. In *IEEE Symp. on Security and Privacy*, 2008.

[36] Type Providers. `https://msdn.microsoft.com/en-us/library/hh156509.aspx`, 2015. Accessed: 2015-08-24.

[37] D. Volpano, G. Smith, and C. Irvine. A Sound Type System for Secure Flow Analysis. *J. Computer Security*, 4(3):167–187, 1996.

[38] Scott Wambler. The Object-Relational Impedance Mismatch. `http://www.agiledata.org/essays/impedanceMismatch.html`, 2015. Accessed: 2015-07-21.

[39] WebSharper. `http://websharper.com/`, 2015. Accessed: 2015-05-01.

Appendix A

Statement of Contribution

The work on this thesis is a joint effort with the Programming Language-Based Security research group at Chalmers, the other project members are Musard Balliu, Daniel Schoepe and Andrei Sabelfeld.

A.1 Implementation

Turning the theoretical model into an implementation represents the main task of this thesis. Therefore most of the F# implementation was designed and provided by Benjamin Liebe. The only exception is the constraint solver, which was contributed by Daniel Schoepe. The case studies were implemented entirely by Benjamin Liebe.

A.2 Report

Chapters 5 and 6 are based on the submitted paper [2], which was co-authored with Musard Balliu, Daniel Schoepe and Andrei Sabelfeld. The initial draft for these chapters was written by Benjamin Liebe and the co-authors performed minor changes of the text while preparing the paper for submission. The other chapters in this report were written exclusively for the purpose of the thesis.