Saarland University
Faculty of Natural Sciences and Technology I
Department of Computer Science
Information Security and Cryptography Group

Master's Thesis

Generic Access Control for Extensible Web Applications within the SAFE Activation Framework

submitted by

Florian Michael Schröder

on December 4, 2012

Supervisor:
Prof. Dr. Michael Backes

Advisor:
Raphael M. Reischuk, Ph.D. cand.

Reviewers:
Prof. Dr. Michael Backes
Prof. Johannes Gehrke, Ph.D.

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides Statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,	Datum/Date	Unterschrift/Signature

Acknowledgements

Writing this thesis would not have been possible without the support that I have received from many people.

In the first place, great thanks go to my supervisor Professor Michael Backes for giving me the kind opportunity to write this thesis in the inspiring working atmosphere of the Information Security and Cryptography Group.

I would like to extend this thanks to my second reviewer, Professor Johannes Gehrke. He made himself available for numerous discussions, from which many ideas emerged. His profound input regarding this thesis and further research challenges was always of great value to me and is much appreciated.

Special thanks go to my advisor, Raphael Reischuk, whose excellent guidance made this thesis actually possible. I owe him manifold insights and the great opportunity to contribute to a particular interesting research field. Due to his comprehensive assistance, I was very fortunate to have Raphael as my advisor.

Furthermore, I am indebted to my fellow friends Bernd and Martin, who supported and encouraged me throughout my whole studies.

Finally, I owe my deepest gratitude to my dearest father, mother, and sister for their care and love.

Abstract

In times of massive and still increasing use of online resources, *Rich Internet Applications* are considered of high importance. In particular, recent Web application trends strive towards so-called *mashup* applications, in which multiple users interact with multiple disjoint software components.

These novel kinds of applications introduce very specific demands on *access control*: As software components within the overall application are possibly contributed by third-parties, it is not sufficient to implement access control on a per-user basis – instead, data access has to be restricted additionally according to components.

Developed at Saarland University and Cornell University, the SAFE framework addresses the common pattern of mashup Web applications. By this means, SAFE applications are composed of functional self-contained units. SAFE, however, lacks a built-in user management mechanism as well as a stringent access control abstraction with respect to both users and (potentially untrusted) software components.

In this thesis, we develop a novel access control mechanism that complies with the demands of contemporary Web environments. In particular, we take account for a generic notion of access control in the SAFE environment, thereby not leaving the enforcement of access control to possibly untrusted components. In order to show the feasibility of our approach, we extend the SAFE framework with a new component-based data model that corresponds to our findings.

Contents

Chapter 1

Introduction

In times of massive and still increasing use of online resources, platform-independent *Rich Internet Applications* (RIAs) are considered of high importance. Called *Web 2.0* in the consumer environment and *Software as a Service* (SaaS) in the professional environment, these kinds of applications are often database-driven and predominantly make high demands on their underlying technology.

In addition, recent Web application trends in which multiple users interact with so-called *mashup* applications composed of multiple disjoint software components introduce particular demands on *access control*: It is not sufficient to implement access control on a per-user basis, as data access has to be additionally restricted according to the particular software component. Such components are possibly contributed by third-parties, which have to be considered untrusted. Boundaries for both users and components have to be enforced centrally and simultaneously – we cannot assume any component to securely and consistently implement user-based access control for itself.

Access control modeled in the core of the Web application itself is a common pattern, for the following reasons. Contemporary database engines include various user-management methods for the access to data and to the system as a whole. Usually, the user management is only performed on a per-connection basis, which requires a new connection whenever different privileges are desired. Besides the lack of per-user queries, there is in general no dynamic row-based access control that would consider the content of a particular record. A typical Web application maintains a persistent database connection for performance reasons with essentially universal permissions. The application itself is thus responsible for tracking the user and for enforcing appropriate access control policies.

Environments in which extensible and component-based Web applications can be conveniently modeled are provided by Web application frameworks, one of which is SAFE. The *Safe Activation Framework for Extensibility* is a framework aiming for a unified handling of the common techniques largely used by today's RIAs, namely HTML, CSS, SQL, and JavaScript. SAFE is designed for a modularized structuring of the application into components, allowing for easy extensibility, e.g., by third-party customizations. The modularization is achieved by dividing an application into semantically coherent features that are provided by functional self-contained pieces of code, the so-called *f-units*. However, SAFE lacks a user management mechanism as well as a stringent access control model with respect to both users and f-units, which have to be considered potentially malicious.

Despite the generally untrusted environment caused by the composition of potentially malicious f-units, the design of SAFE comprises a central and trusted entity. This entity is considered suitable in order to provide a generic and secure interface for defining access control policies. Such policies should protect both honest users and f-units from their dishonest counterparts by generally limiting database access according to appropriate scopes. Furthermore, f-units should be able to state additional policies as perceived by the developers – thereby shifting the responsibility of access control logic to a lower layer orthogonal to the business logic. With respect to crucial properties of modern RIAs, such as modularity, customization, and extensibility, both types of access control policies should be flexible enough to support the ability of extending an application in unforeseen directions.

Overview. In this thesis, we take account for a generic notion of access control in the SAFE environment. By this means, Chapter 2 outlines fundamental concepts and the overall architecture of the SAFE framework. Common access control aspects are reviewed in Chapter 3. Moreover, we introduce an attacker model and derive the requirements of our particular scenario, according to a generic access control model that comprises data separation between both users and software components. In order to meet the derived requirements, Chapter 4 proposes a method that reliably establishes particular boundaries between users and f-units on the database layer. Furthermore, Chapter 5 introduces interfaces for well-defined data exchange allow for customizable information sharing and collaboration. Chapter 6 proposes a flexible though convenient approach for maintaining custom invariants on the datasets of a particular f-unit. The implementation of the presented techniques and a showcase application are shown in Chapter 7. In Chapter 8, we formalize our implemented access control approach and evaluate whether its semantics resembles the generic model stated before. Chapter 9 discusses particular aspects that are considered for further investigation. The retrospection in Chapter 10 concludes the thesis.

Chapter 2

The SAFE Framework

An increasing amount of software is developed for Web applications, which are platform-independent and benefit from increased availability. One of the major concepts for multi-tier Web application development is the combination of the PHP, HTML, CSS, JavaScript, and MySQL techniques.

The diversity and purpose-oriented nature of the deployed techniques makes feature-oriented programming difficult: Code reflecting the same functionality of an application has to be coherently maintained at various locations and scopes. Likewise, integration or modification of new functionality requires many code locations to be touched. Each newly introduced feature has to be reflected by the database layout, the PHP business logic, and the presentation layout as formed by CSS and JavaScript. These common maintenance problems in combination with security concerns limit the general innovation drive: As there is a strong reservation against third parties accessing or even modifying the application code, the integration of every new functionality or extension has to be done and/or reviewed by one of the application's main developers.

Besides the distributed nature of functionality, access control and user management are often spread over the whole application's business logic as well. While a central point to define and review security policies would be desirable in general, a generic access control layer usually comes with a lot of conceptional effort and overhead.

Developed at Saarland University and Cornell University, the SAFE *Activation Framework for Extensibility* [20] claims to encounter many of those challenges in today's Web application development. This chapter outlines the most important aspects of SAFE.

2.1 The Declarative Modeling Language SFW

SAFE applications are implemented using a dedicated modeling language named *Secure FORWARD* (SFW). By extending the HTML syntax, SFW provides additional language primitives and shortcuts for common and re-occurring programming patterns. While HTML in a `.sfw` source file basically remains untouched, SFW is interpreted by the SFW compiler `sfwc` and is translated to corresponding PHP, JavaScript, and/or CSS code (Figure 2.1).

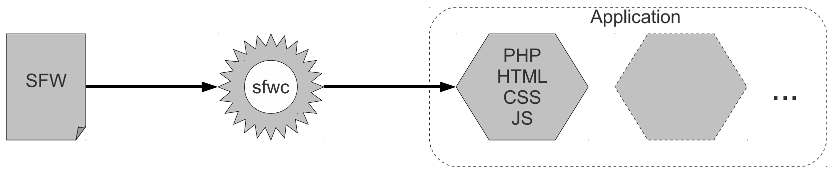


Figure 2.1: The SFW compiler `sfwc` translates SFW statements into PHP code that runs in the application's context.

As an example, Web applications often contain a substantial amount of constant and context-independent code for dealing with database queries. For this reason, SFW allows for direct database access, as presented in Listing 2.2 and Listing 2.3. Additionally, SFW's shortcuts contain abstractions for control flow, database management, form handling, and variable expansion – a list of all features can be found in the official SAFE user manual [19].

```
<form ...>
  <select name="city">
    <for query="SELECT postalcode AS pc, city AS name FROM cities ORDER BY city ASC">
      <option value="$$pc">$$name</option>
    </for>
  </select>
  <input ...>
</form>
```

Listing 2.2: A `for` loop in SFW that is embedded in HTML allows for dynamic content generation using variable expansion for datasets retrieved from the database.

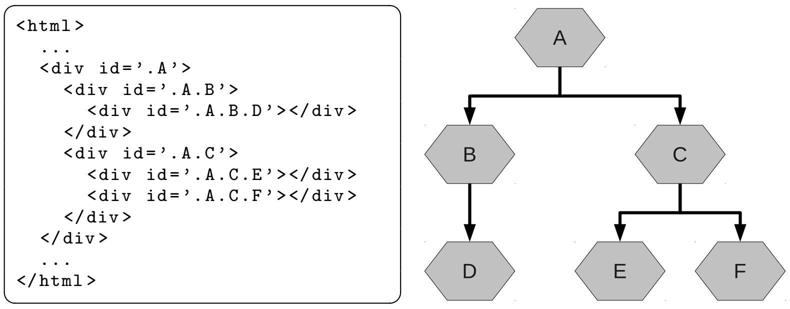
```
<form>
  <input type="text" name="name">
  <input type="text" name="msg">
  <input type="button" value="Send"
         onclick="query:INSERT INTO messages SET msg='$#msg', name='$#name'">
</form>
```

Listing 2.3: Seamless access to form values for embedded database queries in SFW with the advantage of specifying objects and corresponding event-driven code at the same location.

2.2 F-units

Due to the strong relationship between HTML and SFW, SFW inherits the hierarchical character that comes with HTML or XML-related languages in general. In HTML, the particular parts building up a whole page can be unambiguously addressed via the *Document Object Model* (DOM). Figure 2.4 shows the hierarchical arrangement of HTML-defined elements according to the DOM, resulting in a DOM *tree*.

SAFE addresses Web applications that predominantly show the common pattern of providing various functionality on a single page – with a "clustered"



```
<html>
  ...
  <div id='.A'>
    <div id='.A.B'>
      <div id='.A.B.D'></div>
    </div>
    <div id='.A.C'>
      <div id='.A.C.E'></div>
      <div id='.A.C.F'></div>
    </div>
  </div>
  ...
</html>
```

Figure 2.4: A DOM tree as defined by HTML elements and its schematic representation.

layout, as for example in Figure 2.5. Concerning HTML, the arrangement by *syntax* often strongly conforms with the *semantics* of a page's components. As the inheritance of properties throughout the DOM is a fundamental concept in JavaScript and CSS as well, even those additional parts can often be mapped to a corresponding cluster of DOM elements. The hierarchy in syntax as represented by the DOM thus reflects the hierarchy in semantics as represented by the functionality. All the code needed for a particular feature can hence be subsumed within a single functional self-contained unit – a so-called *f-unit*.

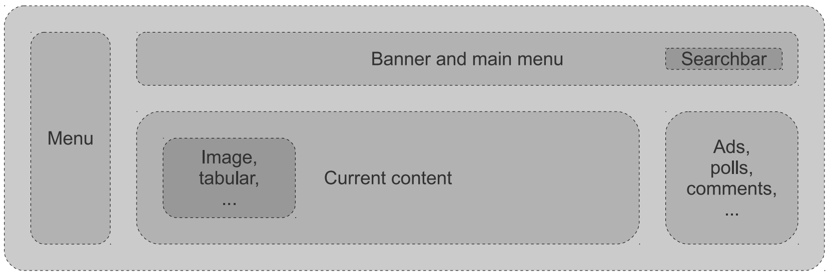


Figure 2.5: Schematic showcase of a common application pattern with clustered functionality.

F-units are structured according to the tree ordering as given by the DOM, but with functionality-based granularity, which often can be directly derived by semantic intuition. As for example in Figure 2.5, application components such as images, menus, or searchbars can be represented by corresponding f-units. Following the hierarchical programming model, an f-unit may invoke child f-units that represent some sub-feature by an SFW `<activate/>` statement. An activated f-unit may receive runtime arguments according to a particular interface in order to define the concrete instance and/or the data to work on.

The approach of clustering an application's code into self-contained f-units by providing an interface and a hierarchy for major and minor prominent tasks supports both the *reusability* and *modularity* paradigms: Because of the partitioning into functionally distinct and the modularization of functionally dependent pieces of code, f-units may be added, modified, or replaced in an transparent way.

2.3 The Centralized Reference Monitor (CRM)

The *Centralized Reference Monitor* (CRM) is part of the so-called *application kernel.* While the application kernel comprises the entire management code of the SAFE framework in general, the CRM is responsible for dedicated tasks such as the management of f-units and the maintenance of database connections. The CRM connects the f-unit structure with the client who is interacting with the application (cf. Figure 2.6). As the CRM is the only interface to the database, the CRM forwards database queries on behalf of f-units.

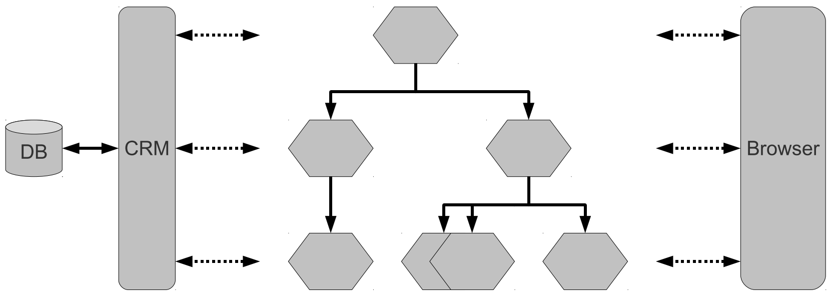


Figure 2.6: The CRM delivers the application consisting of f-units to the client's browser and manages database connections.

The central monitoring of database accesses ensures that all persistently stored information has to pass the CRM. In particular, the supervisor characteristics makes the CRM ideally suited for use as a central authority that is capable to enforce certain constraints on the behavior of an f-unit – i.e., to control access to the database.

2.4 Access Control

The SAFE topology allows the CRM to exclusively inspect all database queries that are issued by an f-unit. In order for the CRM to perform database access control, appropriate scopes have to be stated for each f-unit, on which basis the CRM may decide whether an access is considered permissive or not.

Each f-unit has to provide an *interface* that enumerates which columns of which table the f-unit requests to read from and/or write to. Every incoming query of this particular f-unit is hence parsed and restricted to the corresponding interface.

2.5 Drawbacks

The possibility of reviewing interface files was introduced in order to reveal unintended database access and thereby to assist the responsible application developer who composes applications out of different f-units in estimating the trustworthiness of a particular f-unit. However, this approach requires an f-unit to know any accessed table and column in advance – with respect to extensibility in unforeseen directions, an undesirable property.

Additionally, stateless database access control for basically untrusted f-units on the basis of tables and columns is too coarse-grained in some scenarios. Consider, for example, an f-unit that allows users of an application to publish a personal profile. While the f-unit has to be granted virtually full access on all *columns* of its particular profiles table, there is no need to grant access on all *rows*: If connected to a particular user, neither the content of another user profile must be readable per default, nor must it be available for write operations.

The particular demands on runtime flexibility, user management, and f-unit monitoring suggest a revised access control approach, which will be discussed throughout this thesis. The upcoming Chapter 3 derives the corresponding requirements for a generic access control model more suitable to the SAFE scenario.

Chapter 3

Security Principles – from a Requirements Perspective

Data-driven Web applications require user management and accounting of user activity, mainly in order to enforce *confidentiality* and *integrity* properties: No user should be able to read or modify data that affects other users without explicit affirmation. The reasoning about *access control* implies two preliminary actions, namely, *identification* and *authentication* [23, pp. 135]. In common terms, identification requires a username and authentication a shared secret (e.g., a password) that proves the authoritative use of the username. We will refer to both of these two steps using the term authentication only. The input of the authentication process can be considered a compound value called *credentials*. As the credentials may incorporate some password, we assume that they inherit the properties of a shared secret: Only the user knows the credentials [25, p. 20] that uniquely correspond to a user entity [25, pp. 4].

In order to identify and evaluate threats that might violate either one of the access control properties, the upcoming section states an attacker model that conforms to our particular setting. Subsequently, the attacker model will be used to derive a corresponding access control approach.

3.1 Attacker Model

Applications implemented in SAFE involve three parties (cf. Figure 3.1): The application kernel $\mathcal{K}$ mainly maintaining the database, the f-units $\mathcal{F}$ providing the main functionality and user interaction, and the clients $\mathcal{C}$. While the kernel itself is considered a trusted entity, f-units and clients have to be treated as potentially malicious. On this basis, the SAFE-specific attacker model focuses on both f-units and clients as untrusted entities and presents corresponding threat scenarios.

Before discussing the attacker model, we state three basic assumptions that are anticipated to mitigate the threat scenarios derived afterwards.

Sandbox Assumption (*SB*): In order to prevent communication between f-units possibly leading to unintended sharing of data or even of credentials, the presence of a *sandbox* is assumed. The sandbox suppresses connections to destinations other than the application kernel, i.e., the CRM, and thereby restricts the information flow of any f-unit towards the database.

Query Structure Assumption (*QS*): Queries reaching the CRM must be validated against a well-defined set of acceptable queries with predefined structures, thereby verifying that a client cannot execute self-crafted queries. In other words, a client may only adapt certain values inside the query, while preserving the structure of a query.

Query Data Assumption (*QD*): We assume that every query corresponds to a particular request, which in turn contains reliable information about the authenticated user (as discussed in Section 3.2). On the basis of this particular user information, we assume an access control mechanism to make sure that only well-defined database access occurs. More precisely, the scope of affected information should be limited according to the included credentials and the database semantics.

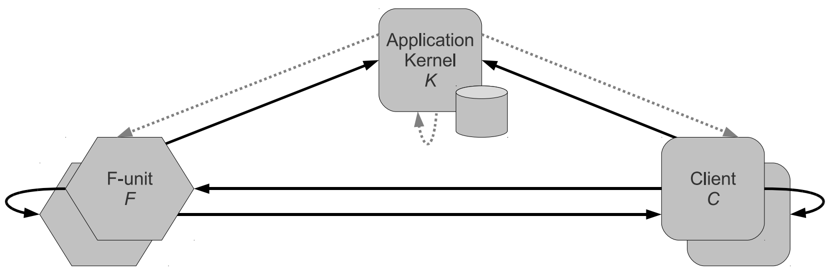


Figure 3.1: Attacker model, stating three involved parties and nine possible threat scenarios.

The applicability of these assumptions is investigated in Section 3.2. For now, the classification according to nine possible threat scenarios, whose mitigation is claimed to be covered by the *SB*, *QS*, and *QD* assumptions, is shown. Threats originating from the application kernel ($\mathcal{K} \rightharpoonup *$) will not be considered, since $\mathcal{K}$ is not assumed to intentionally take part in any malicious activities.

$\mathcal{C} \rightharpoonup \mathcal{F}$ The client provides fake credentials, which are used by the f-unit to perform its operation – thus fooling the f-unit into requesting or providing improper datasets from or to $\mathcal{K}$. This attack is prevented by the authenticity properties for credentials and thus covered by the *QD* assumption.

$\mathcal{C} \rightharpoonup \mathcal{C}$ In preparation of providing faked credentials as in $\mathcal{C} \rightharpoonup \mathcal{F}$, the client tries to gain access to the credentials of another client. However, by the definition of credentials and the implications of the *QD* assumption, credential secrecy with respect to other clients is be assumed.

$\mathcal{C} \rightharpoonup \mathcal{K}$ The client accesses the kernel directly, circumventing possible f-unit checks regarding query, credentials, and user input: Besides the credential properties and affected information checks as implied by the *QD* assumption, unintended query modifications would be required in contradiction to the QS assumption. In other words, no client can place more potent queries as any f-unit is able to place on its own behalf.

$\mathcal{F} \rightharpoonup \mathcal{C}$ The f-unit abuses the credentials obtained from the client to perform unintended actions (possibly including credential leakage): Even if the f-unit's actions are limited to database access (according to the SB assumption), the f-unit can still leak information through the database.

By this means, consider an f-unit that is used for messaging purposes and that maliciously sends a copy of each message to the developer's account. Both reading and sending of messages is covered by the f-unit's normal behavior and is therefore unsuspicious. Thus, preventing information flow between instances is hardly possible, especially in a framework that leaves as much freedom to the developer as SAFE does.

The messaging scenario shows that the functionality of an f-unit can hardly be monitored automatically by means of database communication as a side-channel. In particular, supervision of f-unit behavior would require a semantic scheme of the whole application logic. This leads to the conclusion that potential leaking of own data or other unintended properties have to be considered by the user before the f-unit in question is used – in doubt, the decision should depend on the developer's "trustworthiness".

The threatening implication of an f-unit's uncontrolled behavior by credential abuse can at least be weakened by reducing the consequences: Sensitive client data, e.g., credentials, should not be accessible to the f-unit, but should bypass the f-unit's logic. This hiding of credentials prevents at least their leakage and the consequences beyond the scope of the malicious f-unit. The approach of restricting credentials to the client while hiding them from f-units is discussed in the implementational part in Section 7.1.

Besides accessing the database, gathering information and/or triggering events for the same client but in the context of a different f-unit is possible by accessing the DOM, which the malicious f-unit is part of. Scripting across f-unit boundaries should be comprised by the sandbox assumption SB, though.

$\mathcal{F} \rightharpoonup \mathcal{F}$ In addition to f-unit interaction via the DOM, an f-unit could access database content that is associated to another f-unit, without explicit affirmation: In case of an inappropriate access control mechanism, unintended cross-f-unit data access is possible directly via the database. However, the database semantics as considered by the QD assumption should track an f-unit's scope and thereby prevent non-authorized data access across f-unit boundaries.

$\mathcal{F} \rightharpoonup \mathcal{K}$ The f-unit tries to access data not on behalf of the connected user: Assume for example, Alice is connected to an f-unit that maliciously updates database columns that are considered private by Bob. This threat is covered by the client-credential properties of the QD assumption and the implied semantic access control, based on the particular credentials.

The presented threats yield a more precise outline of the requirements of the particular assumptions. In the following Section 3.2, the presented assumptions are reviewed and formalized. In the scope of a framework such as SAFE in which, for the sake of extensibility, f-units may be provided by third-parties, we hence focus on the needs of the $\mathcal{C} \rightharpoonup \mathcal{K}$, $\mathcal{F} \rightharpoonup \mathcal{K}$ and $\mathcal{F} \rightharpoonup \mathcal{F}$ attacks.

3.2 Assumption-based Mitigations

This section discusses the assumptions stated in Section 3.1. For those assumptions, prerequisites are evaluated and implications are defined.

For an extensible framework, threats targeting at the application kernel or other functional parts are most critical – i.e., $\mathcal{C} \rightharpoonup \mathcal{K}$, $\mathcal{F} \rightharpoonup \mathcal{K}$, and $\mathcal{F} \rightharpoonup \mathcal{F}$ are considered most relevant. For this reason, the corresponding assumptions made for mitigation, i.e., the *QS* and *QD* assumptions, are in focus of the following discussion.

We refer to authentication as an abstract method $\bowtie_c$ that takes user credentials as input. The output called *uid* is only available if the authentication backend in use was able to map the verified input to that particular unique user id. The successful authentication process by credentials is used to derive the corresponding *uid* out of the request req_t at some time t:

$$\frac{req_t \vdash cred \qquad cred \bowtie_c uid}{req_t \bowtie_r uid} \tag{3.1}$$

The *uid* can be considered a *context* the request is bound to using the $\bowtie_r$ relation. The context of a request thus only relies on the verified credentials that have been obtained via the extraction operator $\vdash$. Section 7.1 on session management discusses whether credentials have to be included for every request.

3.2.1 Sandbox Assumption

According to the *SB* assumption, we do not want f-units to exchange, submit, or receive potentially sensitive information through other channels but the database, for which an appropriate sanitization mechanism in the CRM can be deployed.

The most obvious way of information leakage by an f-unit is by sending data to a foreign server that is out of the control of SAFE– which can easily be done using AJAX, forms, iframes, or even by the parameters of an embedded image. Furthermore, unintended local information flow between different f-units can for example be established by accessing the DOM. Hence, the possibilities for a subliminal channel are numerous and an approach claiming to cover them all would nevertheless be prone to incompleteness.

A satisfiable solution would thus most likely be out of the scope of this thesis. We thus leave a sandbox implementation for future work (Section 9.2). In Section 3.1, however, the sandbox assumption *SB* was solely introduced for $\mathcal{F} \rightharpoonup \mathcal{C}$ mitigation that strongly depends on the user's impression of the f-unit's trustworthiness.

3.2.2 Query Structure Assumption

The *QS* assumption stated in Section 3.1 requires the CRM to accept a query only from the f-unit that has previously defined the query's structure and thereby prevents a client from submitting self-crafted queries.

Similar to the concept of *prepared statements* in MySQL[1], SAFE requires f-units to register a prototype for each query. Every query prototype may contain placeholders, such that the CRM can safely embed user-defined data, while preserving the query's overall structure. Technically, the execution of a query *query* thus requires submitting a set of placeholder/data mappings *q_payload* along with a query identifier *q_id* that indicates one of the previously registered query prototypes. As the non-guessable identifier *q_id* is only available in an f-unit's private code and/or in its delivered JavaScript payload where it is protected by the sandbox assumption (Section 3.2.1), it can be assumed that the originating f-unit can be determined reliably.

Using the set of all available query prototypes, we hence assume the presence of a mapping

$$(q\_id, q\_payload) \mapsto (query, funit)$$

that returns the instantiated query *query* for a given query identifier *q_id* with appropriately embedded payload *q_payload*. Using the extraction operator $\vdash$, further retrieval of the source f-unit or the query carried by the request req_t is defined as:

$$\frac{req_t \vdash (q\_id, q\_payload) \qquad (q\_id, q\_payload) \mapsto (query, funit)}{req_t \vdash query \qquad req_t \bowtie_r funit} \tag{3.2}$$

The presented approach of query retrieval ensures the integrity of a query's structure, as required by the assumption. In subsequent steps, the CRM can hence inspect further properties of the query with respect to its associated f-unit, as introduced by the *QD* assumption of the next section.

3.2.3 Query Data Assumption

According to the *QD* assumption, the scope of any provided or received information has to comply with the *uid* the particular query is assigned to. However, an f-unit may access the database on behalf of a client ($\mathcal{F} \rightharpoonup \mathcal{K}$) – for now, responsibility lies solely at the f-unit in deciding which content may be accessed within the connected client's scope. Especially the presence of malicious f-units suggests some constraint enforcement at lower layers – in contrast to hard-coded security policies in the application or in the f-unit itself [15, 21].

In addition to enforcing query boundaries according to the *uid*, the scope of a query has to be validated according to the originating f-unit as well. In particular, mitigation of $\mathcal{F} \rightharpoonup \mathcal{F}$ and $\mathcal{F} \rightharpoonup \mathcal{K}$ requires a query to affect neither clients nor f-units that are considered beyond the issuer's responsibility.

As depicted in Figure 3.2, the CRM acting as monitoring instance has to derive the allowed input and output data for the current *uid* and f-unit – and possibly

[1] `http://dev.mysql.com/doc/refman/5.1/en/sql-syntax-prepared-statements.html`

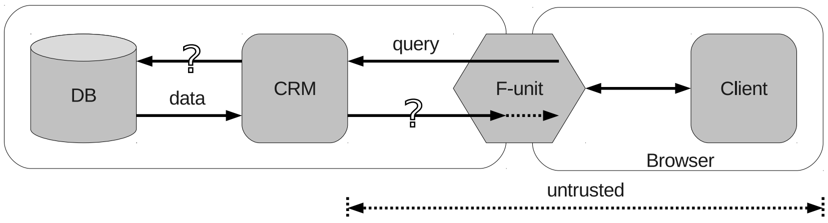


Figure 3.2: Query processing topology

restrict access. By this means, Equation 3.1 and Equation 3.2 enable the CRM to determine the origin of each encountered query with respect to both *uid* and f-unit:

$$\frac{req_t \vdash query_t \qquad req_t \bowtie_r uid}{query_t \bowtie_q uid} \qquad \frac{req_t \vdash query_t \qquad req_t \bowtie_r funit}{query_t \bowtie_q funit} \tag{3.3}$$

However, there is no clear intuition on the usage of this information in the scope of query validation, yet. Because of the substantial impact on any access control, major parts of this thesis will deal with concrete semantics and implementation issues of a sanitization for both received queries and returned data – by referring to the query/*uid* and query/f-unit mappings introduced in Equation 3.3.

3.3 Generic Access Control Abstraction

Traditional access control mechanisms consider the *user* assigned to a dataset in order to accept or reject an operation on this particular dataset. By this means, a trusted entity keeps track of *ownerships* that allow for enforcing appropriate boundaries. This trusted entity can be the application kernel in SAFE or, for example, the filesystem on a multi-user desktop computer, which prevents unintended cross-user file access.

Likewise, there are approaches for enforcing boundaries across *applications*: A sandbox prevents a particular application from accessing data in the scope of another application residing in the same environment. Besides the notion of f-units in the SAFE setting, a common multi-application scenario that suggests to deploy such a sandbox is the encapsulation of application-specific data on contemporary smartphones.

A common term in access control is the notion of a *principal* that usually refers to a user within a system. Especially in the context of both multi-user and multi-application scenarios, by a principal one can thus denote any first-class object for which an access control policy may be applied. A principal can hence be an authenticated user, an f-unit, an installed software component on a smartphone, or a specific physical location around a company's headquarters. More specifically, we consider the n-dimensional universe $\mathcal{P}^n$ of *principal classes*

$$\mathcal{P}^n := \langle \mathcal{P}_1, \ldots, \mathcal{P}_n \rangle$$

that subsumes all instances the particular class $\mathcal{P}_i$, e.g., users, f-units, or locations.

Furthermore, we define the data storage as set of all data entities $\mathcal{D}$. Each such entity is required to have a unique *owner* principal in each dimension, which would be affected by any operation on the particular data entity. For each data item $d \in \mathcal{D}$, we define

$$\mathit{aff}_{\mathcal{P}_i} \colon \mathcal{D} \to \mathcal{P}_i$$

to represent the affected principal in dimension i. The affected principals may be determined with arbitrary semantics, according to operation type, information flow, inference prevention, etc. We thus stay as general as possible here in order to permit a wide range of possible subsequent instantiations. For instance, occurrences in `WHERE` clauses or timing information in side-channels can be captured at particular demands.

In order to access data entities, a principal can issue a request $r \in \mathcal{R}$. The set of *affected data entities* of a particular request

$$\mathit{aff}_{\mathcal{D}} \colon \mathcal{R} \to 2^{\mathcal{D}}$$

assumes the presence of a method that determines the scope of such a request.

As a basic principle, one wants to enable *sharing* between principals of the same dimension, e.g., user Alice wants to share her favorite music files with user Bob. We thus require a sharing function $\mathit{sh}_{\mathcal{P}_i}$ for each dimension $\mathcal{P}_i$

$$\mathit{sh}_{\mathcal{P}_i} \colon \mathcal{P}_i \times \mathcal{P}_i \times \mathcal{D} \to \{0, 1\}$$

to decide whether a sharing from one principal to another is defined for a specific data entity.

Finally, the main access control policy decides whether a given request is valid for all principals associated with a particular request (issuers):

$$\mathit{req\_valid} \colon \mathcal{R} \times \mathcal{P}_1 \times \ldots \times \mathcal{P}_n \to \{0, 1\}$$

More specifically, a request is considered permissive if for each affected principal p_i, we have that either p_i is the issuer itself, or p_i has explicitly shared the requested data with the actual issuer in its principal class.

$$\begin{aligned} \mathit{req\_valid}(r, p_1, \ldots, p_n) \mapsto \forall d \in \mathit{aff}_{\mathcal{D}}(r) \colon \\ \bigwedge_{i=1}^{n} \mathit{aff}_{\mathcal{P}_i}(d) = p_i \vee \mathit{sh}_{\mathcal{P}_i}(\mathit{aff}_{\mathcal{P}_i}(d), p_i, d) \end{aligned} \tag{3.4}$$

Given an extensible Web application, or in particular, the setting of SAFE, consider the set of all users $\mathcal{U}$ and the set of all f-units $\mathcal{F}$ as an instantiation of two different principal classes, such that $\mathcal{P}^2 = \langle \mathcal{U}, \mathcal{F} \rangle$. If f-unit $f \in \mathcal{F}$ issues a query (i.e., a request) $q \in \mathcal{R}$ on behalf of user $u \in \mathcal{U}$, then q is considered permissive if one of the following holds for any affected data entity $d \in \mathit{aff}_{\mathcal{D}}(q)$:

No sharing: $\mathit{aff}_{\mathcal{U}}(d) = u$ and $\mathit{aff}_{\mathcal{F}}(d) = f$, i.e., the query would only access data that is in the scope of both u and f.

Cross-user sharing: $\mathit{aff}_{\mathcal{U}}(d) = u'$ for some $u' \neq u$, $\mathit{aff}_{\mathcal{F}}(d) = f$, and the user u' has shared the requested data with u, i.e., $\mathit{sh}_{\mathcal{U}}(u', u, d)$.

Cross-f-unit sharing: $\mathit{aff}_{\mathcal{U}}(d) = u$, $\mathit{aff}_{\mathcal{F}}(d) = f'$ for some $f' \neq f$, and the f-unit f' has shared the requested data with f, i.e., $sh_{\mathcal{F}}(f', f, d)$.

Cross-user and cross-f-unit sharing: $\mathit{aff}_{\mathcal{U}}(d) = u'$ for some $u' \neq u$, $\mathit{aff}_{\mathcal{F}}(d) = f'$ for some $f' \neq f$, and both the user u' and the f-unit f' have shared the requested data with f running on behalf of u, i.e., $sh_{\mathcal{U}}(u', u, d)$ and $sh_{\mathcal{F}}(f', f, d)$.

Upon reception of a query, knowledge of the originating user/f-unit, the affected data entities, and the affected users/f-units is thus a crucial prerequisite in order to restrict both users and f-units to their "own" data – except for well-defined exceptions by means of sharing.

By Equation 3.3, queries can be mapped to their particular f-unit and the currently connected user. The upcoming chapter considers the basic scenario in which no sharing occurs and thus the extent of affected data entities is limited to the current user's and f-unit's scope.

Chapter 4

A Novel Access Control Approach

During the security analysis in Chapter 3, we introduced the need for a mechanism that ensures confidentiality and integrity of data on a per-user and per-f-unit basis. Furthermore, both the associated *uid* and f-unit of a query can be reliably determined for each request (and thus for each query) – allowing the CRM to perform access control on the basis of *ownerships*, i.e., to enforce appropriate boundaries for users and f-units.

Section 4.1 introduces the tracking of user information for each created dataset (i.e., for each database row), which allows to limit upcoming operations on this particular dataset to the specific user. Likewise, Section 4.2 introduces a mandatory f-unit/database table mapping and thereby implements a well-defined scope that defines permissive queries for a particular f-unit.

4.1 Owner Invariant

We will consider any data modification attempt as authorized only if the operation either adds a new dataset with valid user information, or modifies (or deletes) a dataset that was created on behalf of the user before. By this means, we implement the separation in the dimension of users, as required by our access control model. As this definition requires to keep track of the creating user and the extent of each dataset, we require each dataset – more technically, each row of a table – to hold an *owner* column.

We define the owner-preserving integrity invariant as transition constraints [4, p. 14] on the basis of owner column values:

`INSERT` The owner field of the dataset to be written must match the authenticated user the f-unit is currently connected to.

`UPDATE` The owner field of the dataset to be modified must match the authenticated user the f-unit is currently connected to. In addition, the owner column must not change due to the update operation.

`DELETE` The owner field of the dataset to be deleted must match the authenticated user the f-unit is currently connected to.

We stress that the invariants mentioned above do only take the preservation of metadata into account, i.e., only preserve the owner information of a dataset. In particular, they do not provide any guarantee on the functionality of an f-unit or any security constraints in general. However, the owner invariants ensure accountability and strict access control for deletions.

For implementing the above requirements, the *trigger* concept provided by MySQL is a suitable choice[1]. Before a particular operation takes place, a trigger is able to inspect a column's pending new and old value (where appropriate) by the `NEW` or `OLD` pseudo-table, respectively. In addition, direct access to a column value supersedes the need for parsing the query string and thus reduces the risk for the check of being bypassed. In Listing 4.1, the `UPDATE` trigger ensures both requirements of our stated invariant by raising an error if the value of the owner column would either change or cannot be validated against the connected user. The purpose and semantics of the function `verify_uid()`, its arguments, and the variable `@uid` are derived in the following.

```
CREATE TRIGGER tbl_upd_t BEFORE UPDATE ON tbl FOR EACH ROW CALL assert(
  NEW.owner<=>OLD.owner AND NEW.owner<=>@uid AND verify_uid('unit', 'sk')
);
```

Listing 4.1: MySQL trigger that will be executed before `UPDATE` operations on the table *tbl*. Modifications of the *owner* column and unauthorized modifications are generally prevented.

Although we have to represent many application-level users, an individual database user for each application user is not considered feasible, as motivated earlier – we thus have to use a shared connection with generic f-unit permissions. For a trigger being able to verify our stated owner invariants, the *uid* for each query, which is known at the CRM by Equation 3.3, must be made available to the trigger in a flexible though authentic way. When relying on the fact that there is a single connection per CRM and a single CRM per f-unit processing lifetime, we assume a single connection per f-unit and user. This allows the usage of a connection-specific MySQL session variable[2] to pass the current *uid* along to the trigger with each query. After establishing the connection to the database, the CRM thus sets the following session variables, using the f-unit name *unit*, a secret key *sk*, and a cryptographic hash function $H(\cdot)$:

$$\begin{aligned} @\texttt{uid} &:= \mathit{uid} \\ @\texttt{uid\_h} &:= H(@\texttt{uid} \mid \mathit{unit} \mid \mathit{sk}) \end{aligned} \tag{4.1}$$

Before the query takes effect, the `verify_uid()` function in the trigger of Listing 4.1 is thus able to compare `@uid_h` with the outcome of its own hash computation using `@uid`. The included *sk* inside the hash of `@uid_h` prevents an f-unit from creating valid hashes for arbitrary users on its own, as the *sk* is only available to the CRM and hard-coded in the trigger. Consequently, no f-unit

[1]http://dev.mysql.com/doc/refman/5.1/en/create-trigger.html
[2]http://dev.mysql.com/doc/refman/5.1/en/user-variables.html

should be granted the TRIGGER or SUPER privileges[3]. The *unit* string ensures that even in case the @uid_h is leaked, the security impact is limited to the scope of a particular f-unit and user.

For the sake of automatic database management, the developer may hence provide an *funit*.db file, which declares the tables the f-unit *funit* uses. The declarations (cf. Listing 4.2) are parsed, validated, and interpreted in a dedicated step during the integration process. Each table is forced to specify exactly one column of the hereby introduced column type OWNER. This convention allows the creation of appropriate triggers that verify this particular column against the @uid variable that was set by the CRM prior in the connection – and thereby enforce the invariants as specified above.

```
LOCAL TABLE profiles (
  uid   OWNER PRIMARY,
  phone VARCHAR(50)
  ...
)
```

```
LOCAL TABLE groups (
  id    AUTO PRIMARY,
  name  TEXT,
  owner OWNER
)
```

Listing 4.2: Defining OWNER columns for local tables such that only own profiles or groups may be added, modified, or deleted. (PRIMARY and AUTO denote shortcuts for MySQL's PRIMARY KEY and AUTO_INCREMENT, respectively.)

4.2 Query Sandbox

Our implementation of the owner invariant uses f-unit-specific credentials and thereby guarantees that f-units can only modify datasets of their associated tables – which is, in fact, a desirable property. In order to clarify the semantics of database integration in this regard, we explicitly assign tables to f-units and prevent any cross-references. Moreover, this *sandboxing* approach explicitly ensures integrity across f-unit boundaries and thus limits the impact of the intents of a malicious f-unit. As we have to ensure global uniqueness in the table namespace anyhow, we further clarify this f-unit/table relation by prefixing all names of f-unit-associated tables with the f-unit name.

By using shared database credentials, MySQL table permissions cannot be granted with f-unit granularity. We thus have to ensure that incoming queries only access tables of their originating f-unit, either at runtime in the CRM or statically upon integration. By this means, we turn the notion of a single, global database into multiple, per-f-unit databases.

MySQL's EXPLAIN statement[4] allows for the inspection of accessed tables inside a query. However, the inspection is limited to SELECT queries. Furthermore, a query validation using EXPLAIN could easily be bypassed by hiding references to foreign tables behind AS aliases to own tables, which are considered permissive.

[3]http://dev.mysql.com/doc/refman/5.1/en/show-triggers.html
[4]http://dev.mysql.com/doc/refman/5.1/en/explain.html

A demonstration of this flaw[5] for aliases in "explained" statements is shown in Listing 4.3.

```
> EXPLAIN SELECT * FROM funit_a_tbl AS funit_b_tbl;
+----+-------------+----------+-...
| id | select_type | table    |
+----+-------------+----------+-...
|  1 | SIMPLE      | funit_b_tbl |
+----+-------------+----------+-...
```

Listing 4.3: While the statement would return the contents of a table belonging to f-unit A, `EXPLAIN` reports the referenced table to be belonging to f-unit B.

The CRM thus has to parse each query on its own in order to determine the accessed tables and check for their permissibility according to the particular f-unit.

We hence introduce the *table prefixing*, which prefixes the name of each table with the name of its associated f-unit. For a convenient usage, we do not expose the prefixing to the developer – instead, the CRM automatically replaces each encountered table in the query by its prefixed version, such that each table can be accessed in a query by the name it was defined in the `.db`-file. The table prefixing prevents both name clashes and data access across f-unit borders and thus implements our *query sandbox*.

Each f-unit has to authenticate itself at the CRM before being able to place queries, the f-unit name can hence be determined reliably. The CRM can thus prefix each received query on-the-fly with the f-unit name and is able to expose a clean interface to the f-units and to their developers while completely hiding the sandbox implementation. The security of this approach solely relies on the robustness of the used table-reference recognition and replacing algorithm[6].

Both the query sandbox and the owner invariant as introduced in Section 4.1 establish boundaries between principals of a particular dimension – for f-units and users, respectively. If combined, both approaches thus represent the basic concepts of our overall access control design.

[5] `http://bugs.mysql.com/bug.php?id=24693`

[6] The query parsing and table prefixing algorithm was provided during the work on this thesis and is omitted.

Chapter 5

Data Sharing – the Wiring Methodology

In the previous chapter, we introduced the concept of *owner invariants*, which prevent a user from modifying datasets that were created on behalf of another user. Furthermore, the *query sandbox* limits the scope of a query in the f-unit dimension by enforcing queries to only refer to their particular f-unit's tables.

However, due to limiting f-units to access only their associated tables, we considerably lose flexibility, as cross-f-unit collaboration via the database is prevented – in contradiction to the extensibility paradigm of SAFE. According to the notion of sharing as introduced by our access control model in Section 3.3, we thus need well-defined interfaces for exchanging data across f-unit boundaries, while preserving our integrity and confidentiality constraints.

5.1 Input Tables and Output Tables

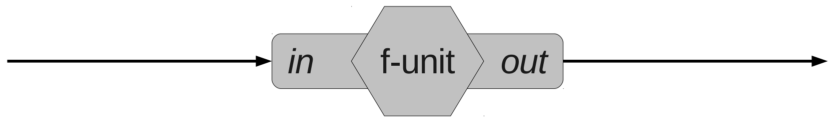


Figure 5.1: An f-unit defines an input table and an output table as part of its interface for data exchange.

In order for an f-unit to expose and receive arbitrary data (cf. Figure 5.1), we introduce the interface concept of *input tables* and *output tables*, which are stated in the f-unit's `.db`-file: Input tables provide a table-like signature for receiving datasets, while output tables implement `SELECT` statements for providing such datasets, allowing f-units to decide on their own, which data in which format shall be exposed. For example in Listing 5.2, an f-unit providing groups exposes each group with its owner, while a statistics f-unit can receive entities of various types. The new keyword `KEY`, the `key` column, and the handling of duplicate keys are introduced in the upcoming section and can be ignored for now.

```
INPUT TABLE stats (
  key     KEY
  owner   OWNER
  type    TINYTEXT
)
```

```
OUTPUT TABLE allgroups =
  SELECT id AS `key`,
         owner,
         name
    FROM groups
```

Listing 5.2: Example: Defining input tables and output tables.

```
<!-- Count all datasets of each particular type -->
<for query="SELECT count(*) AS cnt, type FROM stats GROUP BY type">
  $$type: $$cnt
</for>
<!-- The same for datasets belonging to the current user -->
<for query="SELECT count(*) AS cnt, type FROM stats WHERE owner='$%me'
                                                   GROUP BY type">
  Your $$type: $$cnt
</for>
```

Listing 5.3: Example: Using an input table for statistics.

A major motivation behind input tables is to define an interface for semantically coherent data, whose (not necessarily single) source is unknown in advance but whose structure is defined by column names and the corresponding types in the receiving f-unit. Since the representation of data in the providing f-unit does not necessarily match the intended signature of the input tables, we do not want to limit the power of an output table in collecting its information from other tables. We thus allow arbitrary queries, which are automatically table-prefixed and are thus restricted to the boundaries of the source f-unit. Implemented as a `VIEW`, an output table's signature (column names and types) can be determined reliably after creation using MySQL's `information_schema.columns` table[1]. Together with the input table definitions, we can provide full signatures of both input tables and output tables to a new step of the f-unit integration process – the *wiring*.

During the wiring, an input table schema can be matched to one ore more output table schemata. Input tables are represented by an overall union that combines queries over the involved output tables – an approach in data integration terms usually referred to as *global-as-view* [10]. Each of those input table queries form a *schema matching*, which returns the columns of the output table in the naming and order as defined by the input table. Multiple schema-matched output tables contributing to a single union of an input table can be seen in Figure 5.4. There exist several schema matching techniques that could be used for automatically deriving input/output table correspondences – these techniques are still prone to mistakes, suggesting at least a human-aided approach [3]. However, we leave further improvements of the wiring process between input tables and output tables, such as an algorithm-aided schema matching, for future work.

When composing the overall application, the developer in charge selects the f-units to be integrated. During the integration process, the developer is presented the list of all input tables and output tables and may connect these table columns after reviewing their types and semantics, as shown in Figure 5.5. A column-

[1]`http://dev.mysql.com/doc/refman/5.1/en/columns-table.html`

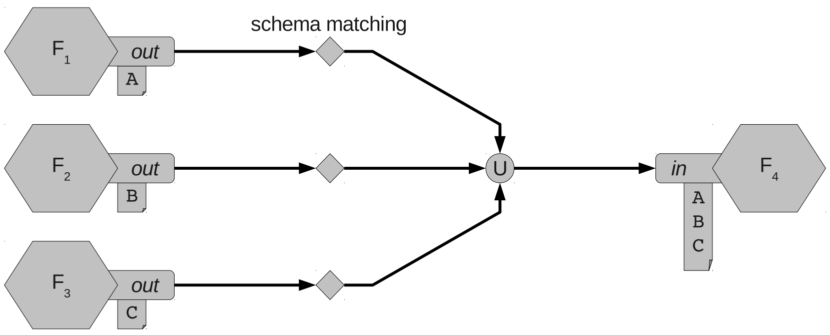
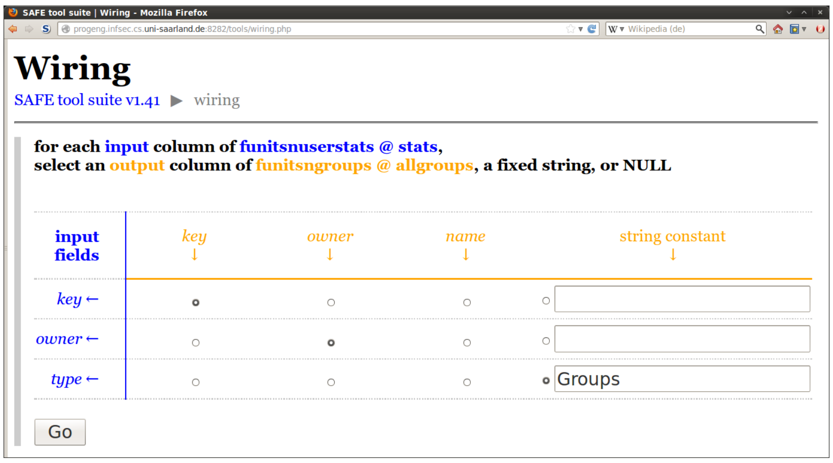


Figure 5.4: An input table of f-unit F_4 consists of a union, which multiple output tables are contributing to. Each output table's schema is mapped to the requirements of the input table, as defined by the wiring.


SAFE tool suite | Wiring - Mozilla Firefox
progeng.infsec.cs.uni-saarland.de:8282/tools/wiring.php
Wikipedia (de)

Wiring

SAFE tool suite v1.41 ▶ wiring

for each input column of funitsnuserstats @ stats,
select an output column of funitsngroups @ allgroups, a fixed string, or NULL

input fields	key ↓	owner ↓	name ↓	string constant ↓
key ←	●	○	○	○
owner ←	○	●	○	○
type ←	○	○	○	● Groups

Go

Figure 5.5: GUI screenshot, showing the wiring of the input table and the output table of Listing 5.2 before usage in Listing 5.3.

wise mapping of the tables in Listing 5.2 could hence consist of `key` $\mapsto$ `key`, `owner` $\mapsto$ `owner`, and the string literal `'Groups'` $\mapsto$ `type`. This wiring would allow the statistics f-unit to use its input table as depicted in Listing 5.3. The possibility of multiple source output tables wired as a `UNION` into a single input table enables the f-unit of our example to display arbitrary statistics without knowing the application's environment in advance.

5.2 Data Consistency via Foreign Keys

The intention of input tables is not only to provide the possibility of collecting data for presentation, but instead, f-units may want to build their own data based on the received entries and thus extend existing datasets according to their own functionality. As an example, consider the pattern of an f-unit managing particular objects (images, groups, profiles, . . .), while a wired child f-unit provides some per-user functionality on top of each parent object (comments, votes, . . .). This 1:1 or 1:N dependency can be expressed as values in a local table that explicitly refer to a value in another local table or even in an input table. Upon deletion of the referenced value, all referencing entries that have become stale shall be implicitly deleted, in order to ensure consistency between both involved tables, e.g., if an image has been deleted, all associated comments shall be deleted as well.

A common term to refer to such natured dependencies is the notion of a *foreign key*, whose semantics is implemented by MySQL using the concept of foreign key constraints[2]. However, foreign key constraints can be used in MySQL only for involved tables being "real" tables. However, we also want to support foreign keys on input tables, which are implemented as a `UNION` over the column-mappings to arbitrarily crafted output tables. We thus implement a generic custom approach in order to emulate foreign key semantics with increased flexibility.

In order to guarantee well-defined dependencies, every referenced table must provide a unique key that can be referred to. For local tables, we require a single `PRIMARY` column (as shown in Listing 4.2), which implies a `PRIMARY KEY` and thus is a valid foreign key target. Furthermore, input tables must provide a `KEY` column, and output tables must provide a column named `key`, whose uniqueness is assumed across the output table. When wiring output table *otable* to an input table *itable*, unique keys across all datasets of *itable* are ensured by the implicitly added column *otable*.`ukey`, which consists of the hash value $H(otable|key)$. Each output table holds thus two keys – one for internal and one for external usage. As *otable* is subject to the table prefixing via the query sandbox, even global uniqueness can be assumed for the generated `ukey`.

Every foreign key of a local table can hence refer to a uniquely defined entry of either another local table or of an input table. We propose a mechanism that ensures consistency by automatically deleting stale child entries upon deletion of the corresponding parent entry. A key-based mapping of the parent/child relationship between local tables through the output table would thus be required. As output tables may define their `key` column arbitrarily before being hashed to `ukey`, we cannot derive a direct key mapping between wiring-dependent local tables in general – i.e., the key values in the parent and the child local table might not be related obviously. However, for all foreign keys f_i defined for the child column $ccol_{f_i}$ referring to the column $pcol_{f_i}$ of the parent table $ptbl_{f_i}$ (whether local or input), we can state the relational algebraic [18, pp. 100] expression

$$\forall f_i : \pi_{ccol_{f_i}} ctbl_{f_i} \subseteq \pi_{pcol_{f_i}} ptbl_{f_i} \tag{5.1}$$

that we will refer to as the *foreign key invariant* defined by the child local table $ctbl_{f_i}$. As for the owner invariant, this additional foreign key invariant can be ensured by triggers that are registered before `INSERT` and `UPDATE` operations on *ctbl*. Inside these triggers, the soundness according to Equation 5.1 of all defined foreign keys f_i can thus be ensured by requiring:

`EXISTS(SELECT * FROM` $ptbl_{f_0}$ `WHERE` $ptbl_{f_0}.pcol_{f_0}$ `= NEW.`$ccol_{f_0}$`) AND ...`

[2]http://dev.mysql.com/doc/refman/5.1/en/innodb-foreign-key-constraints.html

On the other hand, stale entries not satisfying the invariant (anymore) can be removed by a generic delete statement stored in a procedure called *ctbl*`_cleanup()`:

`DELETE FROM` *ctbl* `WHERE` $ccol_{f_0}$ `NOT IN (SELECT` $pcol_{f_0}$ `FROM` $ptbl_{f_0}$`) OR ...`

In order to call the cleanup procedure of *ctbl* whenever any precondition of the invariant might have changed in *ptbl*, we have to keep track of the foreign key dependencies. Within the scope of a particular f-unit, foreign key dependencies are known at integration time, namely from a local table either to an input table or to another local table. Upon wiring, the wired output tables and input tables form a new path in the dependency tree, as depicted in Figure 5.6. For a particular local table, we can follow the paths to directly reachable local tables and thus determine the tables to be monitored for changes. For these invariant-involved tables, successful changes, which passed the *before* triggers ensuring the owner invariant (Section 4.1), can be catched by *after* triggers. We create those triggers during the wiring process in order to call the `cleanup()` procedures of all tables that state a corresponding foreign key. We cannot generally assume a delete operation to be the only reason for an entry to disappear from an output table and thus have to initiate the cleanup for possibly affected tables by the triggers after `INSERT`, `UPDATE`, and `DELETE`. The usage of triggers allows us to neglect dependencies introduced by transitivity, as events raised by a performed cleanup will spread by triggering the after-operation triggers of the affected local tables.

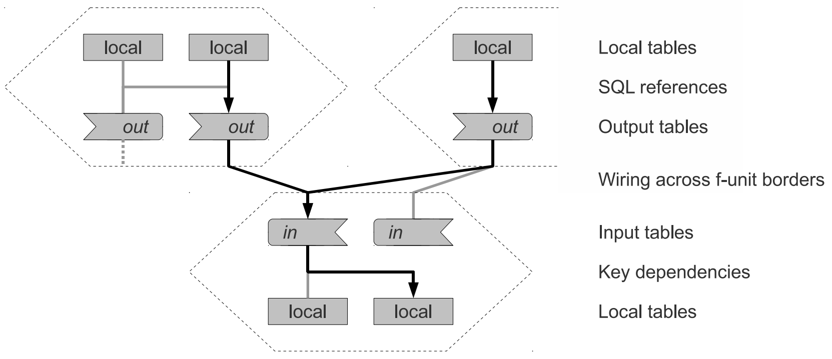


Figure 5.6: Wiring dependency tree: Due to foreign keys, a local table may depend on several other local tables, while a local table may have several depending local tables, too. (Possible direct foreign key dependencies between local tables as well as SQL references between output tables and input tables are not shown.)

However, deletions triggered for consistency reasons might not satisfy our owner invariant as stated in Section 4.1. When using foreign keys, each child object has to be deleted upon deletion of its parent object – the *uid* that accounts for the parent's deletion (its owner) does not generally match the owner of all child objects[3], though. Consequently, the whole transaction could be rejected due to the owner invariant check of the children table *ctbl*: The `UPDATE` and `DELETE` triggers of *ctbl* rely on `verify_uid()`, on the owner column $owner_{ctbl}$, and on

[3]As for example when images can be commented that are handled by another f-unit.

the `@uid` variable, in order to reject deletions of connections not in the scope of the particular user (cf. Listing 4.1):

`assert(OLD.`‘$owner_{ctbl}$‘`<=>@uid AND verify_uid(`’*unit*’, ’*sk*’`))`

In order to overcome the permission inheritance problem, the owner invariant verification has to be circumvented whenever a deletion is caused by the cleanup procedure.

Overriding the trigger checks and deleting entries not belonging to the currently connected user violates our owner invariant. However, this exception is inevitable for ensuring consistency in many use cases and happens only on behalf and with explicit knowledge of the f-unit that defines the foreign key. The cleanup procedure *ctbl*`_cleanup()` thus overwrites the connection's *uid* by calling `set_uid()`, using the global shared secret *sk* for a valid `@uid_h` generation:

`... AND set_uid(`‘$owner_{ctbl}$‘, ’*unit*’, ’*sk*’`);`

As `set_uid()` unconditionally returns *true*, the logic of the preceding conditions is not affected. However, upon evaluation of the `WHERE` clause for every stale entry, the side-effect of setting `@uid` and `@uid_h` according to the current owner column will be executed, thereby simulating the correct *uid* and allowing the deletion to be "approved" by the corresponding `DELETE` trigger.

At a glance, input tables, output tables, and the wiring concept are crucial features in order to cope with the limitations introduced by the query sandbox in Section 4.2: Clear responsibilities and increased security by data encapsulation and scope restriction come at the price of reduced f-unit interoperability and mere impossible collaboration – in contrast to the modularity paradigm of SAFE. In addition to the possibility to directly work on wired input data, we propose an approach that clones basic foreign key behavior, even for arbitrarily crafted input tables as foreign key targets. The wiring concept together with foreign keys thus increases flexibility while keeping an f-units data encapsulated, functionality modular, and responsibility in place.

5.3 Presentation Consistency via Updates

One of the major features and challenges of today's data-driven and reactive Web applications – in particular of SAFE – is to ensure server-client *consistency.* If an f-unit modifies the state of the database, the changes should be reflected by the visual representation of dependent f-units and by the instances of the f-units at the client. In SAFE, any f-unit can *activate* arbitrarily many other f-units:

$$act \colon \mathcal{F} \to 2^{\mathcal{F}}$$

Upon an activation initiated by an f-unit $f \in \mathcal{F}$, activation data is passed from f through the particular activation interfaces of $act(f)$. The behavior of the activated child f-unit instances thus depends on the state of the parent f-unit f. Consequently, the set $act(f)$ is possibly *data-dependent* of f.

F-units generate HTML content that is enclosed by a particular node in the DOM tree of the HTML page. F-units and their activations hence constitute a hierarchical, cycle-free structure, the *activation tree*:

$$G_{act} = \langle V_{act}, E_{act} \rangle = \langle \mathcal{F}, \{(f, f') \in \mathcal{F} \times \mathcal{F} \mid f' \in act(f)\} \rangle$$

Due to the activation data dependencies, each change in an f-unit's data realm causes possibly outdated f-units in the corresponding subtrees of G_{act}.

The concept of wiring introduces an additional possibility of receiving data such that propagation of changes is not necessarily reflected by the edges of the activation tree. The set of all f-units that contribute to some input table of f-unit f is given by $it(f)$:

$$it\colon \mathcal{F} \to 2^{\mathcal{F}}$$

This additional data dependency that was imposed by our sharing mechanism is covered by the *combined graph*

$$G_{comb} = \langle V_{act}, E_{act} \cup E_{sh} \rangle$$

that includes edges representing the presence of an input table/output table channel from one f-unit to another:

$$E_{sh} = \{(f, f') \in \mathcal{F} \times \mathcal{F} \mid f \in it(f')\}$$

The definitions of both E_{act} and E_{sh} can be considered as an over-approximation since they do not respect the extent of the actually changed data – their combination, however, clearly captures all possible dependencies.

Using the combined graph, we can determine and update f-units that rely on stale data. If a local table of a particular f-unit f changes due to a modifying query, the transitive closure starting at f contains all potentially stale f-units that should be considered for updating. For the combined graph, we determine a topological f-unit ordering, which takes the partial orderings as defined by the particular dependencies into account. The global topological ordering is well-defined, as wirings and/or activations that would result in a cycle are rejected at first place, and thereby ensures that the rebuilding step propagates on yet refreshed data. Using the partial ordering, the order of required activations and thus f-units to be rebuilt can be determined – in other words, all f-units are rebuilt in an order such that all data requirements are satisfied. The freshly generated content is merged into complete subtrees of the activation tree (and thus also into the DOM tree). Finally, the rebuilt content is pushed to stale client browser instances.

Chapter 6

Data Invariants

Data-driven Web applications usually rely on numerous assumptions on their maintained data – for both integrity and access control reasons. Consider an instant messaging feature, for which the developer wants to allow a user to write messages only to its friends (or otherwise related users). Typically, such constraints that depend on the application semantics are enforced directly by the application's business logic in a fuzzy, decentralized, and thus hardly maintainable manner. This habit suggests the responsibility of data validation to be placed at the lowest possible layer [15, 21], in order to prevent circumvention and to provide central management.

In this chapter, we first propose invariants to hold for data that is stored in local tables. Subsequently, the approach is extended to allow for invariants on data that is provided by output tables.

6.1 Generic Invariants

In order to give the developer of an f-unit the opportunity to define constraints on his data at the database layer, the table `messages` of the previously mentioned instant messaging feature could only allow users in the `to` column that are "friends" of the owner stated in the `from` column. By enforcing the existence of foreign key targets, we explicitly introduced our first table-specific *invariant* (cf. Equation 5.1) and a mechanism to ensure its validity throughout the particular record's lifetime. However, providing owner and foreign key invariants might not satisfy all needs regarding consistency and/or access control, as seen with the `messages` example.

We thus need an additional mechanism for the developer to provide table invariants, which cannot be modeled in terms of foreign keys or owner columns. Similar to foreign key invariants, *generic invariants* shall provide increased expressivity, reject `INSERT` and `UPDATE` operations for invalid datasets, and automatically delete formerly valid datasets when becoming invalid.

```
LOCAL TABLE messages (
         id AUTO PRIMARY
       from OWNER
         to USER
        msg TEXT
  INVARIANT friends(*, from, to)
)
```

```
INPUT TABLE friends (
  key  KEY
  uid1 OWNER
  uid2 USER
)
```

Listing 6.1: Local table invariant that allows messages to be sent only between particular users, as defined by the input table `friends`.

Using tables as *predicates*, an f-unit may specify an additional `INVARIANT` expression for each table. As shown in Listing 6.1, an invariant stating the predicate

```
friends(*, from, to)
```

holds true if there exists a tuple in the `friends` table that matches the contents of the `from` and the `to` column in the current dataset. Likewise, a predicate that contains a negation as one of the following

```
!ignores(to, from)
!supervisor(to, !from)
```

prevents messages from users on an ignore list, or to foreign supervisors, respectively. With the possibility of negation by `!`-prefixing, the semantics of a single predicate using the table `tbl` is shown in Equation 6.1, expressed using existential quantifiers:

$$\begin{aligned} \texttt{tbl}(\mathtt{x_0}, \dots, \mathtt{x_k}) \quad &\leftrightarrow \quad \exists (p_0, \dots, p_k) \in \mathit{tbl} : \bigwedge_{i=0}^{k} \mathit{match}(x_i, p_i) \\ !\texttt{tbl}(\mathtt{x_0}, \dots, \mathtt{x_k}) \quad &\leftrightarrow \quad \nexists (p_0, \dots, p_k) \in \mathit{tbl} : \bigwedge_{i=0}^{k} \mathit{match}(x_i, p_i) \end{aligned} \tag{6.1}$$

The conjunction over $\mathit{match}(\cdot, \cdot)$ compares the given tuple with the predicate tuples according to the semantics stated in Equation 6.2: The variables x_i represent the given column names and can be bound to their value in the current dataset environment using $\mathit{val}(\cdot)$. In addition, column names can also be `!`-prefixed, denoting an inequality condition at this particular tuple position, as implied by $\mathit{neg}(\cdot)$.

$$\mathit{match}(x, p) = \begin{cases} \mathit{true}, & x = * \\ \mathit{true}, & \mathit{val}(x) = p \land \neg \mathit{neg}(x) \\ \mathit{true}, & \mathit{val}(x) \neq p \land \mathit{neg}(x) \\ \mathit{false}, & \text{else} \end{cases} \tag{6.2}$$

The existential quantifier semantics with conjunctive matching allows for easy deployment of common environments, in which access control bases on group memberships, permissions, and/or user relationships. As predicates can even refer to input tables, the wiring process provides the flexibility and modularity needed for incorporating extensions at runtime – knowing the input table's column semantics is sufficient for an f-unit to state senseful invariants.

While supporting basic checks in an invariant, a single predicate only might not be sufficient in order to express the desired semantics. For increased functionality, we thus support predicates to be linked by `AND` and `OR` clauses. Together with obeying parenthesis for convenience, invariants may consist of more complex expressions such as:

`(friend(from, to) OR admin(from)) AND ...`

Intuitively, this invariant allows *members* of the *group* `admin` posting messages to everyone, even though not being a `friend`.

The increased power in expressivity on the predicate level also allows the column matching to be of virtually arbitrary semantics. For example, one can express *forall* semantics in addition to *exists* for completeness reasons, by exploiting quantifier duality and by passing matching logic to the predicate level:

$$\begin{aligned} &\forall(p_0, \ldots, p_k) \in tbl : \bigwedge_{i=0}^{k} match(x_i, p_i) \\ = &\nexists(p_0, \ldots, p_k) \in tbl : \bigvee_{i=0}^{k} \neg match(x_i, p_i) \\ \leftrightarrow &\ \mathtt{!tbl(!x_0, *, \ldots)\ OR\ \ldots\ OR\ !tbl(\ldots, *, !x_k)} \end{aligned} \tag{6.3}$$

$$\begin{aligned} &\neg\forall(p_0, \ldots, p_k) \in tbl : \bigwedge_{i=0}^{k} match(x_i, p_i) \\ = &\exists(p_0, \ldots, p_k) \in tbl : \bigvee_{i=0}^{k} \neg match(x_i, p_i) \\ \leftrightarrow &\ \mathtt{tbl(!x_0, *, \ldots)\ OR\ \ldots\ OR\ tbl(\ldots, *, !x_k)} \end{aligned} \tag{6.4}$$

As the main intuition and syntax still implies comparison of the full tuple that was given as argument, this *forall* representation comes with a certain additional effort, which should only be needed in rare cases, though.

Until now, valid predicate parameters that can be passed to $match(\cdot, \cdot)$ are the asterisk `*`, a column name `col` of the current table, and its negation `!col`. However, this might not include the possibility of referencing all relevant data – i.e., if the local table uses foreign keys, only the key values themselves are referable by their column name as predicate argument. This contradicts our intuition, as foreign key values are predominantly used for pointing to a unique parent object, whose attributes are extended by the children's data. We thus want to provide the possibility of foreign key expansion (or foreign key traversal), as known from the object-oriented paradigm in general.

We choose the dot (`.`) notation to indicate foreign key-based object expansion. If for example an f-unit provides the functionality of posting a comment to a published image, one may wish to restrict commenting to friends of the image owner: The input table `comments_for` provides the set of images with at least the required columns `id` and `owner`, while the local table `comments` holds at least the columns `for_id` (the foreign key to `comments_for.id`), `owner`, and `comment`. By using object expansion, one can access attributes of the object referred to by `for_id` and state the desired invariant as `friends(owner, for_id.owner)`.

In order to evaluate the dot notation with respect to $match(\cdot,\cdot)$ in Equation 6.2, we have to augment the $val(\cdot)$ semantics by not only returning the value of a single column name in the current transaction. If called with multiple arguments $a_0, \ldots, a_i$ (implicitly splitted according to the dot character "`.`"), val' will recursively resolve foreign keys and ultimately return the value of the last referenced column a_i in its particular table.

$$\begin{aligned} val'(a_0) &= val(a_0) \\ val'(a_0,\ldots,a_i) &= \pi_{a_i}\sigma_{fkc(a_0,\ldots,a_{i-1})=val'(a_0,\ldots,a_{i-1})} fkt(a_0,\ldots,a_{i-1}) \end{aligned} \tag{6.5}$$

Upon successful key traversal, the projection π returns at most a single 1×1 tuple, since foreign key target values are unique and thus unambiguous. As foreign keys can be specified within the particular f-unit's scope only, foreign key reference chains $a_0, \ldots, a_i$ can be resolved at integration time using the implicitly available functions *fkc* and *fkt* for resolving foreign key target columns and their tables, respectively: Starting with column a_0 of the current table, the graph spanned by all foreign keys can be traversed until reaching a_i, which determines the target column and table.

So far, our approach basically requires predicate parameters to be columns, whose names can be bound to their particular values. Amongst other things, the lack of free variables or runtime-instantiable variables restricts functionality if one wants to express a relation's transitivity, for example the presence of an intermediate friend:

```
friend(from, to) OR (friend(from, $i) AND friend($i, to))
```

Although being able to work with a "friends of friends" semantics could be a desirable feature, we rely on f-units to export all its possibly senseful relations and leave the implementation for this or a similar approach for future work.

As with owner invariants (Section 4.1) and in particular foreign key invariants (Section 5.2), we use triggers to implement `INSERT/UPDATE` invariant verification and `DELETE` propagation. For invariant verification, we augment the particular triggers by an additional `AND`-condition that we derive from the whole invariant expression, for example from the single predicate `tbl`$(\mathtt{x}_0,\ldots,\mathtt{x}_i)$:

`EXISTS(SELECT * FROM` *tbl* `WHERE` p_0=$val'(x_0)$ `AND ... AND` p_i=$val'(x_i)$`)`

A straight-forward approach can be applied for `*`-parameters or `!`-prefixed and thus negated parameters or predicates. Resolving foreign key references and returning column values can be done by replacing $val'(\cdot)$ with the output of a suitable implementation of Equation 6.5 at integration time. For delete propagation, we augment the foreign key dependency edges in Figure 5.6 by edges that represent used predicates and traversed foreign key tables of a local table's invariant. This will call the `cleanup` function (Section 5.2) of the table that defined the invariant, whenever a dataset's validity according to the invariant might have been compromised. The condition of the cleanup function's delete statement is thus augmented by an `OR` condition, which can be formed by the negation of the whole invariant expression as stated above.

In this section, we proposed a simple and generic syntax for defining invariants on data held by a particular table. With the combined use of local tables and even input tables as predicates, complex expressions can be given, which allow each dataset's values to be set into relation with the properties of the developer's intention. Apart from logic primitives like conjunction, disjunction, and negation, the expressivity comprises foreign key semantics by allowing for object expansion on a foreign key basis. Finally, the trigger concept was augmented to prevent invalid datasets finding their way into the table, while datasets becoming invalid for external reasons are automatically and recursively deleted.

6.2 Output Table Invariants

Apart from rejecting unauthorized data modifications on local tables by means of owner invariants (Section 4.1) or custom invariants (Section 6.1), leakage of potentially sensitive user data has to be prevented. In the scope of *confidentiality*, we thus have to provide a mechanism that allows to limit read access on particular datasets.

Consider an f-unit F that provides the functionality of friendships between users. While friendship information might be valuable for other components, e.g., for an f-unit that provides some messaging feature, each particular dataset of the corresponding output table of F shall only be accessible for either one of the involved users. This intuition reflects that a user that provides some information to F has to trust F in implementing appropriate access control, whether in the scope of business logic or output tables.

Every persistently stored dataset might be processed arbitrarily by an f-unit before ultimately reaching the particular local table. As f-units "see" thus any dataset anyhow upon insertion, every information represented in a local table could be reconstructed by the corresponding f-unit. It is thus irrelevant, with respect to confidentiality, whether we explicitly allow f-units to directly access arbitrary datasets of an associated local table or not. Despite that, access control directly on top of local tables could be achieved by parameterized views that exclusively provide *uid*-based restricted access to the underlying tables. This so-called *Truman model* suffers from various drawbacks – e.g., transparent views that hide particular datasets may introduce subtle logical inconsistencies for aggregate functions such as `AVG` or `COUNT` [21].

Due to the limited gain and the anticipated problems of a restriction directly on local tables, we explicitly neglect this approach for now. However, the fact of local tables being public in their f-unit's scope[1] does not necessarily contradict our fundamentally pessimistic perception of an f-unit, as f-units do not gain any additional knowledge as a result. In addition, the potential leakage or abuse of information has to be considered anyhow by the user before providing sensitive data to a particular f-unit – the trustworthiness of an f-unit as a crucial assumption on the behavior of an f-unit was introduced in Chapter 3.

However, regardless of the fact that an f-unit may access all its explicitly provided data, data that is passed along via wiring shall be restricted: The source

[1]Due to the query sandbox (Section 4.2).

f-unit, whose access control logic is assumed to be trusted by the user, has currently no control on whether a receiving f-unit performs appropriate access control as well. In the scope of confidentiality, we thus propose access control solely for *output tables*. In particular, we introduce the possibility of hiding datasets for output tables according to the *uid* the receiving f-unit is currently connected to. The decision on whether and to which extent an output table's information has to be restricted requires semantic information on the data's representation and thus has to be under the responsibility of the source f-unit. By *uid*-based filtering of output tables, we maintain the property that f-units are only able to receive possibly sensitive data if they are in use by the particular user.

Analog to invariants on local tables (Section 6.1 and accordingly Section 7.2), we thus allow a deviant output table syntax to incorporate the possibility of defining an `INVARIANT` expression for each output table. In addition to column names as predicate arguments, we introduce the variable `@uid`, which allows *uid*-based restrictions for every access at runtime. Listing 6.2 shows several examples of output tables that define invariants on their returned datasets. Unless overridden by explicit invariant specification, the default behavior of output tables assumes the invariant `is(@uid, owner)` and thereby generally protects any private data. In particular, consider the output table `friends_o` of Listing 6.2 being wired into an input table of a malicious f-unit F. Due to the invariant of `friends_o`, F would gain knowledge of all friends of a particular user U only if U used F once – in other words, F has no access granted to U's datasets until being activated in the scope of U's *uid*.

```
OUTPUT TABLE friends_o (
       SELECT CONCAT(uid1, '|', uid2) AS `key`,
              uid1 AS `owner`,
              uid2 AS `friend`
         FROM friends
    INVARIANT is(@uid, owner) OR is(@uid, friend)
)
```

```
OUTPUT TABLE messages_o (
       SELECT msgid AS `key`,
              from AS `owner`,
              to,
              msg
         FROM messages
    INVARIANT is(@uid, owner) OR is(@uid, to)
)
```

```
OUTPUT TABLE profiles_o (
       SELECT owner AS `key`,
              owner,
              profile
         FROM profiles
    INVARIANT is(@uid, owner) OR friends_i(@uid, owner)
)
```

Listing 6.2: Invariant examples for output tables, using the variable `@uid`, the relation `is()`, and the input table `friends_i()`.

The limitations of the Truman model concerning aggregate functions may also affect our presented approach of suppressing datasets by invariants on output tables. However, the source f-unit may provide sensefully aggregated data under particular circumstances, as the full dataset is locally accessible. Additionally, our approach does not explicitly consider information leakage as a result of inference – e.g., by indirect access, statistical inference, or data correlation [4, pp. 17]. Consider here again `friends_o` of Listing 6.2 wired into the f-unit F: F not only intentionally gains knowledge of `@uid`'s friends, but can also successively gather friendship information on the friends of `@uid`. In the worst case, F can know all the friends of user U for sure, even if U never used F – after all other users used F once. However, this applies only as the "is a friend of" relation is assumed to be semantically mutual and thus commutative, in contrast to e.g., "likes". Mitigation of such natured inference is left to the responsibility of the source f-unit, which should thus model the output table invariants according to the anticipated benefit-cost ratio.

Chapter 7

Implementation

By means of invariants and the query sandbox, we have introduced mechanisms that provide integrity and confidentiality across boundaries of both users and f-units. In addition, the concept of wiring allows for well-defined interaction and data sharing between f-units. This chapter discusses various implementational issues of the aforementioned approaches in greater detail.

7.1 Session Management

Chapter 3 has motivated the need for database access control according to both users and f-units. In order to enable the CRM to perform user-based access control, i.e., to enforce owner and output table invariants, the *uid* of the currently connected client has to be determined in a reliable way. More specifically, Equation 3.1 assumed the client's credentials to be present for each request – and thus for each query (Equation 3.3).

However, requesting user credentials for every query is not feasible while providing those credentials to f-units contradicts our basically pessimistic perception of f-units. We thus introduce a mechanism of maintaining state under control of the CRM: A so-called *session* mechanism provides the ability to recognize the source of a request. Thereby, a session mechanism allows a current request req_t at time t to be linked to an event that belongs to a past request $req_{t'<t}$ of the same source.

In particular, we introduce a session function $s(\cdot)$ that refers to the past event of a successful authenticated request $req_{t'}$ according to our $\bowtie_r$ semantics in Equation 3.1. Equation 7.1 makes use of the main functionality a session mechanism has to provide: With $src(\cdot)$ we refer to the intuitive matching of request origins, sticking both requests to the same client.

$$\frac{t' < t \qquad req_{t'} \bowtie_r uid \qquad src(req_t) = src(req_{t'}) \qquad s\_valid(t, req_t)}{s(req_t) = uid} \tag{7.1}$$

A past request $req_{t'<t}$ with valid credentials is thus interpreted as a *login* and

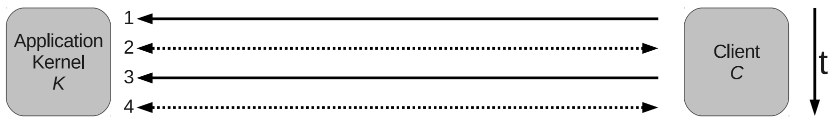


Figure 7.1: Four stages of the session model: Authentication, initialization, operation, and invalidation.

fixes the *uid* of requests having the same source until the session becomes invalid for arbitrary reasons[1] according to $s\_valid(\cdot, \cdot)$.

The session function $s(\cdot)$ allows for binding future requests $req_{t>t'}$ to the *uid* after authenticating once and thus provides a *uid* caching mechanism. This property can be used for extending $\bowtie_r$:

$$\frac{s(req_t) = uid}{req_t \bowtie_r uid}$$

According to Equation 3.3, a query inherits its request's *uid* information and can thereby be validated appropriately.

We apply this definition of a session to our scenario as a straight-forward authentication procedure consisting of four stages, as depicted in Figure 7.1:

1. The client authenticates at the application kernel. If req_1 contains valid credentials linked to the user id *uid*, this request represents the initial login with $req_1 \bowtie_r uid$.
2. The login procedure of the session mechanism may include the execution of a specific stage for session initialization that allows for $s\_valid(t, \cdot)$ being evaluated later on for some $t > 1$.
3. Requests can be considered authenticated for a particular *uid*, if their sources match the ones encountered in the first stage – in our example, it holds: $src(req_3) = src(req_1) \Rightarrow req_3 \bowtie_r uid$
4. The session may now become explicitly or implicitly invalid, such that requests otherwise being accepted for *uid* will now fail due to $\neg s\_valid(t, \cdot)$ for $t > 4$.

In order to implement session functionality in our scenario, we introduce a mechanism of maintaining a state on top of HTTP, which is considered stateless per se: HTTP offers no generic way of linking a current request to past requests and thus to past events. A widely used mechanism for implementing sessions by introducing state in HTTP is by using *cookies*. Introduced by Netscape in 1997 [8], the client browser is urged to echo the value obtained once by the `Set-Cookie` header back to the server using the `Cookie` header. The possibility of maintaining state by referring to past events provides a functionality to implement the semantics of a session.

As a basic intuition, one could hand out the plain *uid* using a cookie upon successful authentication. For upcoming requests of the same client, the corresponding *uid* would then be trivially determinable. However, shifting the task

[1]For example due to an implemented *logout* functionality or a session expiration.

of maintaining the whole authentication state to the cookie in a reliable and secure manner is a non-trivial task [14]. A common approach is thus to use a cookie only to refer to a state that is maintained by the application (kernel) itself.

Upon authentication of a client with user id *uid* in stage *1* (cf. Figure 7.1), the CRM chooses a unique and hardly guessable identifier (nonce) as session id *sid*. In order to later on recall the *uid* based on a received *sid*, we maintain a database table `sessions` over these two values. A logout functionality and/or a timeout deletes the corresponding row of this `sessions` table in stage *4*.

After mapping a *sid* to the *uid*, the *sid* can be passed along to the client by setting a cookie in stage *2* (cf. Figure 7.1). Since the corresponding *uid* has to be available at the CRM for each client query in stage *3*, every possible path a query can take must reliably provide a *sid*-carrying cookie. There are two kinds of queries:

1. *Initial Queries* in an f-unit's static PHP code that are directly executed by a local instance of the CRM. This is the case with `for` or `activation` statements (Chapter 2), for example.
2. *User-Triggered Queries* that are issued on behalf of client interaction (for example by a `query` statement of a `button`): These queries are sent to a particular `ajax.php` script and processed in the background.

CRM instances of both an `ajax.php`[2] and an f-unit's regular `.php` file can access cookies and thus retrieve the particular *sid*. After verification according to $s\_valid(\cdot,\cdot)$, whose semantics is still to be defined, the corresponding *uid* can be derived and used for access control purposes.

As there is no need for the *sid* to be available in client-side scripts, we use `HTTPOnly` cookies [12]. Additionally, our $s\_valid(\cdot,\cdot)$ implementation accounts for an expiration time for maintenance and security reasons: We augment the `sessions` table by an `expires` column that holds a timestamp, which is incremented for each access. Restricting a session's lifetime by using a cookie's `expires` or `max-age` flag is not authentic and thus no reliable solution on its own.

There is a main and well-known weakness in our approach so far: If the value of a cookie happens to get stolen or leaked, an adversary is able to access the application with the victim's privileges (*session riding* or *session hijacking*). While there is a lot of related work on the topic on securing cookies [11, 28, 17], we stick for now to the straight-forward method of binding the cookie value to the client IP address. By this means, we extend the *sid* by an IP-based signature, yielding:

$$sid' := sid \mid H(sid \mid ip \mid sk)$$

The signature uses an appropriate cryptographic hashing function H and consists of three parts: A static secret key *sk* to prevent third parties from generating valid signatures, the actual *sid* to prevent a replay of the signature for other sessions, and the client IP address *ip* that implements the binding property of sid'. The signature's non-forgeability property solely relies on the secrecy of *sk*,

[2] `http://stackoverflow.com/questions/1041285/does-jquery-send-cookies-in-a-post`

which makes the signature prone to *brute force* attacks. In addition, there are several problematic technical scenarios that are discussed in Section 9.3. However, the signature should prevent the basic attack of using a foreign cookie's value in our scenario.

To summarize, our presented cookie-based approach fulfills the semantics of the conditionals in Equation 7.1:

- The occurrence of a past $\bowtie_r$ event (login) is detected through the existence of a row in the `sessions` table corresponding to the cookie's value.
- The matching of upcoming request sources $src(\cdot)$ is ensured by the non-guessability of the randomness in the session id and the restriction to the IP address.
- The validity $s\_valid(\cdot)$ of the session is represented by the check for expiration and by the absence of a logout event.

Our session approach moves the assumed non-guessability and secrecy properties of the client's credentials to the *sid*. While the non-guessability property should be preserved, the secrecy property might be compromised by f-units that maliciously access cookies. However, the short-term impact of a *sid* leakage on the overall security is limited by binding a session to a particular IP-address.

7.2 F-unit Integration Overview

The database-related process of f-unit integration copes with features such as local/input/output table creation, invariant enforcement, dependency tracking, and trigger management. This section reviews and summarizes technical aspects of those concepts presented in Chapter 4, Chapter 5, and Chapter 6.

Basically, f-unit database integration takes an f-unit's `.db`-file as input and generates and executes the SQL-statements corresponding to encountered table definitions. An abstract overview of the whole process is given by Figure 7.2, illustrating subtasks and their particular input and output.

1. Parsing: The `.db`-file is read in line per line and is parsed into suitable datastructures, which are returned. Datastructures of interest are basically the four sets of defined entities: Local tables, input tables, output tables, and foreign keys. A rough overview of the recognized syntax for defining the different classes of tables is given in Listing 7.3.

For local tables and input tables, column definitions are parsed into column names and their corresponding types. In order to increase the chance of type compatibility without conversion during a potential wiring, recognized basic column types are given from a reduced set[3] of the types provided natively by MySQL. According to Section 4.1 and Section 5.2, both local tables and input tables have to specify exactly one `OWNER` and one key column, which is of type `KEY` for input tables and marked with the flag `PRIMARY` for local tables.

[3]Ignoring performance and/or space constraints, "superset" types are chosen from similar purpose data types – for example `DATETIME` subsumes the ranges of both `DATE` and `TIMESTAMP` (`http://dev.mysql.com/doc/refman/5.1/en/datetime.html`).

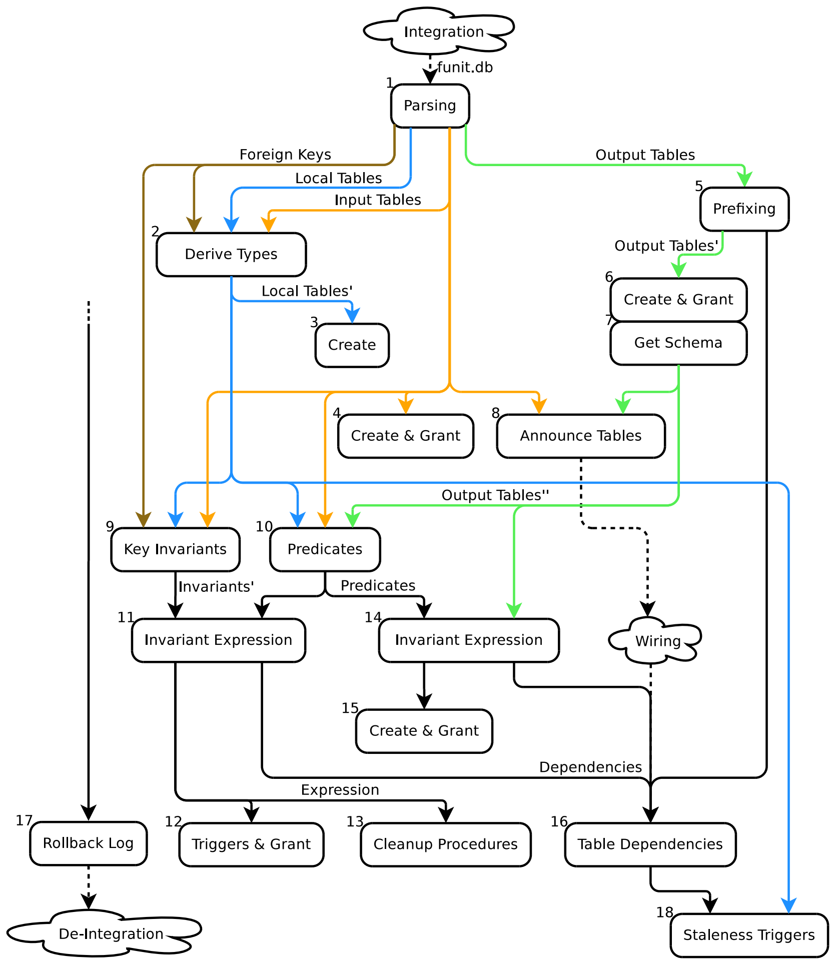


Figure 7.2: Schematic overview of f-unit database integration. From top to bottom, the major steps required for evaluating a `.db`-file are shown.

In order to flag columns of local tables to hold only unique values (besides keys), the `UNIQUE` keyword can be given both in the type definition and in one or more dedicated rows, possibly defining multiple sets of columns that are unique for themselves. Similar to lines consisting of `UNIQUE` followed by column references, an invariant as introduced in Section 6.1 can be stated for a local table by a row starting with `INVARIANT`. Furthermore, local tables may define foreign keys to column *col* of table *tbl* by a type definition of `WTYPE(`*tbl*`.`*col*`)`.

Output tables can be stated as a native MySQL `SELECT` statement. As introduced in Section 6.2, the multi-line syntax for output tables allows an `INVARIANT` expression to be given. While output tables that are available for wiring are defined using either one of the `OUTPUT TABLE` statements, it is also possible to define views for internal use only (as for example in invariants or by the presentational part) via `LOCAL VIEW`.

```
LOCAL TABLE [SINK] mytable (
  (column_name local_column_definition)+
  (INVARIANT ...)?
  (UNIQUE ...)*
)
INPUT TABLE myinput (
  (column_name input_column_definition)+
)
OUTPUT TABLE myoutput (
  SELECT ...
  (INVARIANT ...)?
)
OUTPUT TABLE myoutput = SELECT ...
LOCAL VIEW myview = SELECT ...

type = INTEGER | TEXT | DATETIME | ...
input_column_definition = type | KEY
local_column_definition = (type | USER | OWNER | WTYPE(tbl.col) | AUTO)
       (UNIQUE | PRIMARY | [NOT] NULL | DEFAULT (""|NULL|0-9+))*
```

Listing 7.3: Major components of a `.db`-file in a relaxed grammar syntax.

Upon reaching the end of the `.db`-file, the parsing returns the sets of all local tables and input tables along with their properties (i.e., their columns), all output tables (or local views), and all defined foreign keys as a table/column $\mapsto$ table/column mapping.

2. Derive Foreign Key Types: A common usability drawback of foreign keys in MySQL is that the types of both involved columns have to match, though having to be defined separately. This inconvenience can be resolved by a foreign key definition using `WTYPE` that implicitly derives and inherits the type of the referenced column.

Valid foreign key targets are `PRIMARY` or `UNIQUE` columns of other local tables or `KEY` columns of input tables inside the scope of the particular f-unit. As columns of the type `OWNER` and `USER`[4] implicitly state a foreign key, the *uid* column of the CRM-controlled table `sfw_users` is considered a valid foreign key target as well.

Using the foreign key mapping and the type information of all permissive foreign key targets, the gaps in a local table's type definitions caused by foreign keys can be recursively filled.

3. Create Local Tables: Using the complete type information for local tables, the only "real" tables can already be created in the database:

`CREATE TABLE` *tbl* `( ... ) ENGINE=InnoDB DEFAULT CHARSET=utf8;`

Apart from minor deviations and substitutions, the `LOCAL TABLE` syntax in Listing 7.3 strongly resembles the MySQL `CREATE TABLE` syntax. Consequently, stated column definitions can be basically passed through – even the line-wise `UNIQUE` statements can be taken over in an (almost) unchanged manner[5].

4. Create Input Tables: In order to prevent reference errors, input tables

[4] As with the `OWNER` type, `USER` columns hold *uid*s that might be used in explicit invariants, but without any implicit invariant checks as for `OWNER` columns.

[5] `http://dev.mysql.com/doc/refman/5.1/en/create-table.html`

with proper column signatures have to be created as well – even though there is no wiring in place and thus no data available yet. A default input table *itbl* with columns col_i does not contain any data and therefore returns an empty resultset obtained using the "dummy" table `DUAL`[6]:

```
CREATE VIEW itbl AS SELECT NULL AS `col0`, ... FROM DUAL WHERE 1=0;
GRANT SELECT ON itbl TO funituser;
```

At this point, the `SELECT` privilege can already be granted to the f-unit database user. Upon wiring, the `SELECT` statement of an input table's `VIEW` may then be replaced by a suitable statement returning the desired resultsets from one ore more output tables.

5. Prefix Output Tables: SQL statements defining output tables (or local views) have to be put on a level with all other queries executable by an f-unit. Consequently and due to the query sandbox restriction (Section 4.2), every accessed table of the output table has to be prefixed with the f-unit name[7], thus enforcing the permissive scope. During the prefixing process, all accessed tables are collected as dependencies of the particular output table and are stored for later reference.

6. Create Output Tables: The prefixed representation of an output table statement has to be augmented by an appropriate `ukey` column, as introduced in Section 5.2. As the `VIEW` of an output table *otbl* may not contain a subquery in the `FROM` clause[8], the creation is splitted up into two steps: The actual output table statement is created as an intermediate view *otbl*`_orig`, while the `ukey` is explicitly added by a subsequent view holding the actual *otbl* name.

```
CREATE DEFINER=funituser SQL SECURITY DEFINER VIEW otbl_orig AS ...;
CREATE VIEW otbl AS
     SELECT *, SHA1(CONCAT('otbl_', `otbl_orig`.`key`)) AS `ukey`
       FROM `otbl_orig`;
GRANT SELECT ON otbl TO funituser;
```

When referencing *otbl* in for example invariant expressions, the `ukey` column is thus intentionally available. Due to the appended column, however, the developer has to be aware of the increased arity of the derived predicate. For local views basically being output tables not available for wiring, the intermediate view is not necessary, as there is no need for a public `ukey`.

As a view's query runs in the context of the creating user (the privileged CRM user in our case) per default[9], the explicit restriction to the f-unit database user *funituser* ensures the output table query is executed with the same permissions a regular f-unit query would be executed as well.

7. Get Output Table Schema: Currently, output tables and local views lack any column information, as they are defined using plain SQL statements without any additional information. However, column names and types are needed in the wiring process and for the evaluation of invariant predicates.

[6] `http://dev.mysql.com/doc/refman/5.1/en/select.html`

[7] While there exist numerous projects on SQL parsing routines, e.g., `http://pear.php.net/package/SQL_Parser` and `http://code.google.com/p/php-sql-parser/`, the algorithm was given during writing this thesis.

[8] `http://bugs.mysql.com/bug.php?id=16757`

[9] `http://dev.mysql.com/doc/refman/5.1/en/stored-programs-security.html`

As both output tables and local views are created by now, we can obtain the needed information using the `columns` table of the builtin metadata database `information_schema`[10]:

```
SELECT column_name, column_type FROM information_schema.columns
                                WHERE table_name='otbl'
                                ORDER BY ordinal_position ASC;
```

The output table and local view datastructures are thus augmented by the particular column information and are returned for further processing.

8. Announce Wiring Tables: Since the wiring is an independent process, the column information of all integrated input tables and output tables has to be preserved. We thus store the signatures of all tables available for wiring in a dedicated database table, in order to be able to deduce a suitable schema matching later on.

9. Foreign Key Invariants: As generic invariants will be dynamically ensured anyhow, we rely on the generic invariant mechanism for implementing foreign keys, too. Even for foreign keys between local tables, MySQL foreign key constraints would not be an option in our scenario, as cascaded deletions erroneously[11] do not activate (i.e., *after*) triggers[12], which we use for invariant delete propagation (Section 5.2). For each foreign key defined at column *ccol* of the child local table *ctbl* to column *pcol* of the parent target table *ptbl*, we thus repeatedly augment the invariant of *ctbl*:

$$inv_{ctbl} = (inv_{ctbl})\ \texttt{AND}\ \texttt{ptbl}(\ldots, *, \underbrace{\texttt{ccol}}_{\text{at position of } pcol}, *, \ldots)$$

This approach will prevent the insertion of a dataset not satisfying the foreign key invariant, i.e., of a dataset stating a value in *ccol* that does not exist in *pcol*. Both the invariant checking and the delete propagation will be implemented by triggers, which are created in an upcoming step.

10. Predicates: For the upcoming invariant expression parsing, we need the set of all valid predicates, together with their column information, i.e., their arity. Table names available for usage as predicate are those of all local tables, input tables, output tables, and local views defined by the particular f-unit. Moreover, we provide a binary `is` relation for *uid*s as cartesian product over all registered users.

11. Invariant Expression Parsing: Given the set of all predicates, the invariant string of each local table is parsed into an invariant *tree*, according to the syntax introduced in Section 6.1. As a side-effect of the invariant parsing, all used predicates are collected and provided for further processing as dependencies.

12. Create Triggers: Invariant enforcement for every local table *tbl* is ensured by triggers before `INSERT`, `UPDATE`, and `DELETE` operations:

```
CREATE TRIGGER tbl_op BEFORE op ON tbl FOR EACH ROW
                      CALL assert(tbl_op_checks);
```

[10]http://dev.mysql.com/doc/refman/5.1/en/columns-table.html
[11]http://bugs.mysql.com/bug.php?id=11472
[12]http://dev.mysql.com/doc/refman/5.1/en/innodb-foreign-key-constraints.html

The custom `assert()` procedure raises an error condition[13] and thus aborts the current operation, whenever its argument does not hold valid. Depending on the operation *op*, the checks *tbl_op_checks* to be passed consist of a subset of foreign key invariants, generic invariants, and owner invariants.

For an `INSERT` operation, *tbl_op_checks* has to contain the foreign key invariants and generic invariants as defined by the invariant expression tree. The corresponding SQL representation using `EXISTS` statements can be derived as described in Section 6.1. Furthermore, the owner invariant – i.e., the equality of the owner column *owner* value and the connected user – has to be ensured. The *tbl_op_checks* are thus `AND`-augmented with the *uid* verification using the f-unit name *funit* and a static secret *sk*, as introduced in Section 4.1:

```
verify_uid('funit', 'sk') AND (NEW.`owner`=@uid)
```

The `verify_uid()` function

```
CREATE FUNCTION verify_uid(unit VARCHAR(128), secret VARCHAR(128))
       RETURNS BOOLEAN RETURN (@uid IS NOT NULL) AND
       (@uid_h<=>SHA1(CONCAT_WS('|', @uid, LOWER(unit), secret)));
```

validates the session variable `@uid`, which the CRM initializes accordingly using `set_uid()`:

```
CREATE FUNCTION
  set_uid(uid VARCHAR(40), unit VARCHAR(128), secret VARCHAR(128))
  RETURNS BOOLEAN RETURN 'foo'<>
  @uid_h:=SHA1(CONCAT_WS('|', @uid:=uid, LOWER(unit), secret));
```

An `UPDATE` operation trigger additionally ensures that the owner column remains static, and thus adds the following expression:

```
NEW.`owner`<=>OLD.`owner`
```

For a `DELETE` operation on the other hand, no invariant but the owner invariant has to be considered inside of the assertion.

After creating the triggers that implement our access control, the permissions for the particular operations (including `SELECT`) may be granted to the f-unit user.

13. Cleanup Procedures: As introduced in Section 5.2, we have to explicitly ensure automatic deletion of invalid and thus stale entries. For each local table *tbl*, stale entries can be identified by the negation of the SQL expression *checks* that reflects the same conditions as those inside of the triggers – except for the owner invariant.

```
CREATE PROCEDURE tbl_cleanup() DELETE FROM tbl
                               WHERE set_uid(`owner`, 'funit', 'sk')
                                     AND NOT (checks)
```

By calling *tbl*`_cleanup()`, stale entries are thus identified by re-evaluating the particular table invariants and deleted accordingly.

Each local table collects the cleanup procedures of dependent tables for delete propagation in its `cleanup_t()` procedure:

[13] As MySQL does not natively provide a reliable way of aborting an operation, `assert()` simply deletes from an inexistent table: `CREATE PROCEDURE assert (IN a BOOLEAN) SQL SECURITY INVOKER IF NOT (a) THEN DELETE FROM raise_error; END IF;`

```
CREATE PROCEDURE tbl_cleanup_t() DO 0; -- NOOP
```

The contents of the `cleanup_t()` procedure and the associated triggers are set up in an upcoming step.

14. Output Table Invariants: As for invariants on local tables, the invariant expression of an output table is parsed and predicates are translated into corresponding SQL statements (Section 6.1). In addition to column names as predicate arguments, the token `@uid` is allowed, as discussed in Section 6.2. During the parsing, all referenced predicates are collected and passed along as additional dependencies of the particular output table.

15. Create Output Table Invariants: For any output table *otbl*, a third view named *otbl*`_out` is created – in addition to *otbl*`_orig` (the statement as given by the user) and *otbl* (the view providing the `ukey` for use in queries or invariants). This additional view is created for use in the wiring process, as it restricts the *otbl* view according to the generated invariant statements. If no invariant is specified for *otbl*, the default *otbl*`_out` view omits datasets that are not assigned to the current `@uid`:

```
CREATE VIEW otbl_out AS SELECT * FROM otbl WHERE `owner`=get_uid();
```

As variables such as `@uid` cannot be referred from within view definitions[14], all occurrences of `@uid` are replaced by the function `get_uid()`, which overcomes this limitation by returning the needed *uid* value.

However, the `@uid` value returned by `get_uid()` has to be validated according to `@uid_h` using `verify_uid('`*funit*`', '`*sk*`')`. As the currently connected *funit* cannot be determined inside of *otbl*`_out` for each access, the task of `@uid` verification has to be done by every input table – thereby moving the responsibility to the wiring implementation.

16. Collect Table Dependencies: The cleanup functionality of stale datasets according to foreign keys (Section 5.2) or invariants in general relies on the possibility to track dependencies between tables. At this point, all of these dependencies are centrally stored – namely, predicates used in local table invariants (including foreign key invariants), predicates used in output tables, and accessed tables in SQL statements of local views or output tables (obtained during the table prefixing step). The resulting dependency tree allows to propagate changes by calling the `cleanup()` procedures of affected tables, whose invariants are thereby re-evaluated.

While all collected dependencies are formed between tables inside the current f-unit so far, a wiring between input tables and output tables of different f-units will form a cross-f-unit path inside of the dependency tree.

17. Rollback Log: As many of the so far presented SQL statements are not usable in a single MySQL transaction[15], atomicity and rollback functionality have to be implemented "by hand". We thus collect for each executed statement a corresponding *undo* statement[16] throughout the whole integration process. This set of undo statements are used for a rollback functionality that allows

[14]`http://dev.mysql.com/doc/refman/5.1/en/create-view.html`
[15]`http://dev.mysql.com/doc/refman/5.1/en/implicit-commit.html`
[16]Corresponding statements in this sense are for example `GRANT/REVOKE`, `CREATE/DROP`, or `INSERT/DELETE`.

for leaving the database in a clean state after encountering an error during the integration. In addition, the rollback statements are stored in the database and are executed in the context of an f-unit de-integration.

18. Table Dependency Triggers: At this point, all newly created table dependencies are known, whether formed by foreign keys, wiring[17], invariant predicates, or references in output tables or views. This dependency information is required for implementing a local table's delete propagation mechanism, which basically calls the `cleanup()` procedures of all dependent local tables, whenever this might be required. As in-between output tables may be crafted arbitrarily, we have to consider all the possible triggers *after* `INSERT`, `UPDATE`, and `DELETE` operations as eventually indicating a relevant change in a local table's data[18]. For each local table *tbl*, we thus follow the dependency tree, determine the potentially affected local tables $deptbl_i$, and update the *after* triggers of *tbl* such that the $deptbl_i$`_cleanup()` procedures are called accordingly.

Starting at *tbl*, we recursively traverse the dependency paths and collect the affected tables $deptbl_i$ with their particular cleanup routines. By a trigger's nature, successful changes will continuously propagate, such that there is no need to care about transitivity – we thus terminate at the first encountered local table on each path.

As there is no way for an atomic in-place modification of the cleanup triggers, all after-triggers merely call the single *tbl*`_cleanup_t()` procedure, which finally calls the actual collected `_cleanup()` routines. We thus are able to gracefully break the delete propagation chain by first dropping the triggers, followed by dropping the *tbl*`_cleanup_t()` procedure itself.

The chain is now rebuilt backwards by first re-creating the common cleanup trigger procedure:

```
CREATE PROCEDURE tbl_cleanup_t() BEGIN
  CALL deptbl0_cleanup(); CALL deptbl1_cleanup(); CALL ...
END;
```

After bringing the triggers back in place as well, the *tbl*`_cleanup_t()` procedure is called explicitly in order to cope with the presence of inconsistent datasets, which might have accumulated in the meantime.

7.3 Wiring

The f-unit integration process provides the enumeration of all integrated input tables and output tables, together with their column information. Upon wiring, the user can pick one available table out of each type and can define a schema matching between the table columns via an interface (cf. Figure 5.5). The outcome of the wiring process is a *column mapping* that maps each column $icol_i$ of the input table *itbl* either to a constant c_i or to a column $ocol_i$ of the output

[17]As we are still in the integration phase, wiring for this particular f-unit is not possible yet.

[18]However, for foreign key dependencies between local tables only, a `DELETE` trigger would be sufficient.

table *otbl*:

$$\forall icol_i \in itbl : icol_i \mapsto m_i, \qquad m_i = \begin{cases} \texttt{'c}_\texttt{i}\texttt{'} \\ \texttt{`otbl`.`ocol}_\texttt{i}\texttt{`} \end{cases}$$

The mapping can be transformed into a single `SELECT` statement that returns the columns of *otbl* in the naming and order as defined by the columns of *itbl*:

```
SELECT m0 AS `icol0`, m1 AS `icol1`, ...
  FROM `otbl_out`
 WHERE verify_uid('funit', 'sk');
```

As introduced in Section 6.2 and accordingly in Section 7.2, the invariant-protected output table *otbl*`_out` relies on each input table to ensure the *uid* validity according to the current *funit* and the global secret *sk*. The resulting `SELECT` statement is stored in the database for this particular *otbl*/*itbl* pair. As the following process is held generic, an unwiring operation deletes the corresponding entry from the database and proceeds starting at this point.

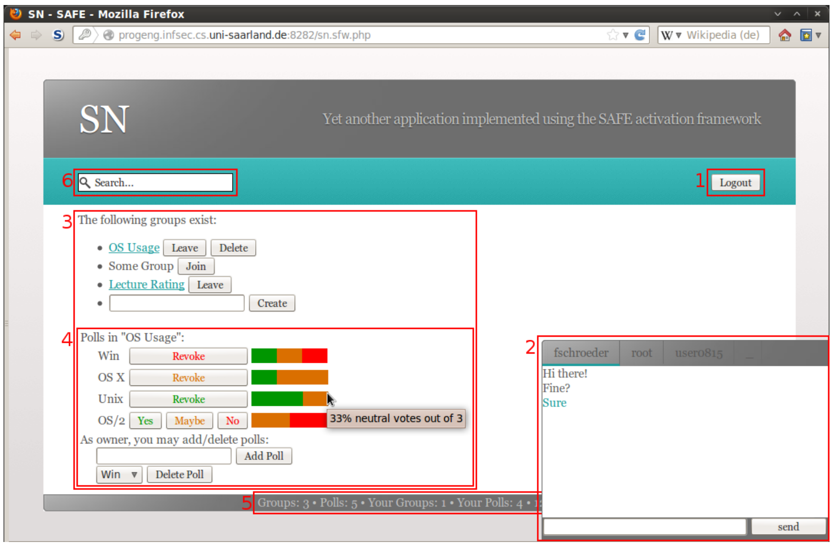
Using the stored information of all defined wirings, the view of the involved input table *itbl* can be updated. Like for the first integration, *itbl* will be replaced by a dummy view returning no data, if there is no wiring defined. In case of at least one mapping being available in the database, the view of *itbl* will be replaced by the `UNION` over all the `SELECT` statements that define the particular output table mappings.

In addition to the input table's view, the wiring might affect present table dependencies – i.e., if the involved input table is a parent table in a foreign key context or is used as invariant predicate. Consequently, the delete propagation mechanism has to be updated, involving triggers on the *otbl* and `cleanup()` procedures on the *itbl* side. We thus determine all local tables tbl_j that are contributing to the output table *otbl*. As output tables may even access input tables, the recursive search can spread across f-unit scopes, until reaching the first local table *tbl* on each path. As there might exist other dependencies, the triggers of each *tbl* are fully re-created, using the table dependencies approach presented for the integration in Section 7.2. The updated wiring dependencies will thus ensure that the `cleanup()` procedure of every *itbl*-dependent local table is called by the cleanup triggers of every *otbl*-contributing local table.

7.4 Use Case Application

During writing this thesis, a showcase application was developed, for general SAFE functionality evaluation and, in particular, access control testing purposes. As depicted in Figure 7.4, the application mainly comprises six different f-units in order to provide functionality for user login, instant messaging, groups, polls, statistics, and livesearch, respectively. The documented source is attached to the electronic version of this thesis for further reference.

1. User login, logout, registering, unregistering, and password changing is implemented by `FUnitSnAuth`. As each f-unit has to authenticate at the CRM, the CRM is able to expose the corresponding interfaces for user


SN - SAFE - Mozilla Firefox

progeng.infsec.cs.uni-saarland.de:8282/sn.sfw.php

Yet another application implemented using the SAFE activation framework

The following groups exist:

Polls in "OS Usage":

33% neutral votes out of 3

As owner, you may add/delete polls:

Groups: 3 • Polls: 5 • Your Groups: 1 • Your Polls: 4 •

Hi there!
Fine?
Sure

Figure 7.4: Use case screenshot highlighting six involved f-units.

management exclusively to this particular f-unit. Upon successful login event, each f-unit is reloaded and thereby provided an environment variable, which allows to provide functionality depending on the login status.

2. `FUnitSnMessaging` enables authenticated users to compose instant messages and thereby expects the set of permissive recipients in a particular input table. As there is no f-unit that would provide a notion of friendships yet, the input table stating valid sender/recipient tuples contains the cross-product of all registered users.

3. Using `FUnitSnGroups`, any user may create a group, which can be joined by other users. Group membership is denoted by a unique group-id/user-id combination and announced by a particular output table – in addition to an output table that provides all groups together with their group-id. Upon clicking on a certain group, the group-id is used to activate the f-unit that provides the functionality encapsulated within groups – poll votings.

4. A group owner may create arbitrarily many polls that are bound to a particular group-id using a foreign key on an input table, which states all groups in their capacity of poll containers. For each poll, group members may place a vote, which involves an invariant referring to the particular input table that states group membership. The foreign key and invariant concepts ensure both polls and votes cleanup in case of group deletion or membership termination, respectively.

5. Statistics on the number of messages, groups, and polls are provided by `FUnitSnUserStats`. The corresponding statistics input table is wired to the three particular output tables, for which a fixed string denotes the source type that can be grouped by.

6. Similar to the statistics f-unit, `FUnitSnLiveSearch` relies on an input table that can be wired to output tables stating arbitrary textual content and

source type. Upon typing event, the content is inspected according to the given pattern and matching rows are shown.

We illustrate how to conveniently extend an existing application with new functionality, based on the previously introduced techniques. More specifically, we take the `SAFE` application of an interactive social network and exemplarily show how to add the incremental search functionality provided by `FUnitSnLiveSearch` (6).

As f-units are urged to state appropriate output tables for the sake of extensibility, `FUnitSnGroups` provides the public output table `all_groups` with a data and an invariant declaration:

```
OUTPUT TABLE all_groups (
      SELECT name, gid AS key, owner FROM groups
   INVARIANT is(owner, @uid) OR !is(owner, @uid)
)
```

The output table exposes the group names to the wiring process: If wired, other f-units can access the names of all groups. Per default, every user may access output table rows with a matching owner column (`is(owner, @uid)`, see Section 6.2). However, the invariant of `all_groups` replaces this default behavior by making the group information public, i.e., readable for every user, and thus for every `@uid`.

Furthermore, the instant messaging f-unit `FUnitSnMessaging` defines the output table `private_msgs` as the set of all messages (local table `conversations`) that can be associated with the current user:

```
OUTPUT TABLE private_msgs (
      SELECT msg_id AS key, msg, uid_from AS owner, uid_to AS to
        FROM conversations
   INVARIANT is(owner, @uid) OR is(to, @uid)
)
```

The invariant of `private_msgs` allows foreign f-units to access both sent and received messages of the particular user they are currently connected to.

Given both `FUnitSnGroups` and `FUnitSnMessaging`, we show how to integrate the common incremental search functionality provided by `FUnitSnLiveSearch`. By this means, `FUnitSnLiveSearch` monitors a text input field for typing events, searches all its available datasets for the input pattern, and displays matching rows. `FUnitSnLiveSearch` is equipped with an input table `data` that can be wired to output tables of other f-units. The input table has two main data fields: `text` for arbitrary textual content (e.g., chat messages, group titles, poll descriptions), and `type` for an informal description of the search source type (e.g., messages, groups, polls).

```
INPUT TABLE data (
  text  TEXT
  type  VARCHAR(20)
  key   KEY
  owner OWNER
)
```

By virtue of this input table, `FUnitSnLiveSearch` is able to search arbitrary datasets – even for data sources that are provided by f-units that were not known before, or by f-units that might come up in the future. At runtime, `FUnitSnLiveSearch` compares these data sources with the search patterns entered in its search input field:

```
<input type="text" name="search" id="searchField">
```

`FUnitSnLiveSearch` issues queries against its input table `data` for every `keyup`-event of the search field and activates corresponding instances of the helper f-unit `FUnitSnLiveSearchResults`:

```
<activate:SnLiveSearchResults
  query="SELECT text AS result, type AS info
           FROM data
          WHERE '$#search'<>'' AND LOWER(text) LIKE LOWER(CONCAT(
                  '%', REPLACE('$#search', ' ', '%'), '%'
                ))"
  refresh="searchField.keyup"
/>
```

In our social network setting, the search engine shall now include the groups of the social network in its search results.

In order to provide `FUnitSnLiveSearch` with the actual group names, the wiring of `all_groups` into `FUnitSnLiveSearch.data` maps `ukey` $\mapsto$ `key`, `name` $\mapsto$ `text`, the constant `'Group'` $\mapsto$ `type`, and `owner` $\mapsto$ `owner`. See Section 5.2 for the purpose of the implicitly added `ukey` column. Furthermore, upon integration of `FUnitSnMessaging`, the new feature of searching in both sent and received messages can be stated by the wiring of `private_msgs` into `data`: `ukey` $\mapsto$ `key`, `msg` $\mapsto$ `text`, the constant `'Message'` $\mapsto$ `type`, and `owner` $\mapsto$ `owner`.

The wiring of `all_groups` and `private_msgs` into `data` results in a safe setting that reflects our modularity and extensibility paradigms. The implementation of `FUnitSnLiveSearch` benefits from various features and concepts that are offered by the described extensions of SAFE. For instance, the result set of *LiveSearch* can be arbitrarily augmented at "run-time", and the wiring allows for easy integration of new functionality into an existing application, without affecting already established f-units. Collaboration across f-units thus only relies on a sufficiently generic interface of all involved f-units, formed by input tables and output tables. Furthermore, `FUnitSnLiveSearchResults` – or any other involved f-unit – can be replaced by means of extensibility with respect to both presentation and functionality, allowing for augmenting the application in unforeseen directions.

In addition, even though `FUnitSnMessaging` publishes privacy-sensitive data, the f-unit itself is able to bind datasets to appropriate invariants and thus has full control over which data might possibly be presented to other users. Consequently, the impact of an malicious f-unit on the overall system's security is limited to the abuse of the malicious f-unit's very own or received datasets.

Chapter 8

Evaluation

In Section 3.3, we introduced an access control model that comprises multiple principal dimensions and weakens the resulting data separation by means of sharing. Furthermore, in Chapter 4, Chapter 5, and Chapter 6, we proposed approaches that implement access control in the context of SAFE, namely, owner invariants, query sandboxing, and wiring.

In this section, we present an instantiation of the access control model introduced in Section 3.3. In particular, we consider the SAFE environment, which instantiates the principal universe

$$\mathcal{P}^2 = \langle \mathcal{U}, \mathcal{F} \rangle$$

to consist of the set of all users and the set of all f-units that are present in the system. On this basis, this section formalizes the semantics of our implemented access control approach and shows that the derived semantics resembles the formal model stated in advance.

8.1 Model Instantiation

Tables and Affected F-units. As we consider all data entities of $\mathcal{D}$ being rows of a particular database table, data entities can be grouped according to a set of database tables $\mathcal{T}$. The set of all affected tables $\mathit{aff}_{\mathcal{T}}$ according to a particular request $r \in \mathcal{R}$ and the set of all data entities held by a particular table $t \in \mathcal{T}$ hence establish a relation between requests $\mathcal{R}$ and data entities $\mathcal{D}$:

$$\begin{aligned} \mathit{aff}_{\mathcal{T}} &\colon \mathcal{R} \to 2^{\mathcal{T}} \\ \mathit{data} &\colon \mathcal{T} \to 2^{\mathcal{D}} \end{aligned}$$

By this means, all data entities of all affected tables form an over-approximation over all actually affected data entities $\mathit{aff}_{\mathcal{D}}(r)$ for some request $r \in \mathcal{R}$ – in other words, the datasets of affected tables include the actually affected datasets:

$$d \in \mathit{aff}_{\mathcal{D}}(r) \Rightarrow \exists t \in \mathit{aff}_{\mathcal{T}}(r) \colon d \in \mathit{data}(t) \tag{8.1}$$

The set of all affected tables can be obtained using the table prefixing algorithm as introduced by the query sandbox in Section 4.2 and accordingly in Section 7.2.

Furthermore, we have to introduce the notion of input tables. Besides local tables

$$lt\colon \mathcal{F} \to 2^{\mathcal{T}}$$

that hold the actual data of the assigned f-unit, input tables subsume data entities of the same schema that can be explicitly provided by other f-units by means of *sharing* – implemented using wired output tables, as introduced in Chapter 5. Access control policies might impose restrictions on sharing the actual data of an input table according to the user who is accessing the data (cf. Section 6.2). Input tables are therefore instantiated for a particular user id:

$$it\colon \mathcal{F} \times \mathcal{U} \to 2^{\mathcal{T}}$$

As the overall content of an input table $t \in it(\cdot,\cdot)$ may be provided by multiple f-units, the providing f-unit of a particular retrieved data entity $d \in data(t)$, i.e., the source

$$src\colon \mathcal{D} \to \mathcal{F}$$

constitutes the affected f-unit for access on an input table:

$$\forall t \in it(\cdot,\cdot), d \in data(t)\colon\ aff_{\mathcal{F}}(d) = src(d) \tag{8.2}$$

For access on local tables, the affected f-unit is the associated f-unit $f \in \mathcal{F}$ itself:

$$\forall t \in lt(f), d \in data(t)\colon\ aff_{\mathcal{F}}(d) = f \tag{8.3}$$

Owners and Affected Users. In order to determine the affected user of any access on any data entity $d \in \mathcal{D}$, we require the presence of an *owner* mapping

$$own\colon \mathcal{D} \to \mathcal{U}$$

to be maintained in the database. The retrieval of owner information relies on the *owner* column for local tables and input tables, as introduced in Section 4.1 and Chapter 5, respectively. We instantiate the affected user accordingly:

$$\forall d \in \mathcal{D}\colon\ aff_{\mathcal{U}}(d) = own(d) \tag{8.4}$$

Sharing. As discussed in Section 6.2, f-units may access arbitrary datasets of their own local tables[1]. Access control responsibility for local tables is thus delegated to the dimension $\mathcal{F}$ of f-units by a virtual cross-user sharing:

$$\forall t \in lt(\cdot), d \in data(t), u \in \mathcal{U} \setminus \{own(d)\}\colon\ sh_{\mathcal{U}}(own(d), u, d) \tag{8.5}$$

In other words, any user $u \in \mathcal{U}$ that uses some f-unit $f \in \mathcal{F}$ has to rely on f in which data f presents to other users.

Similarly, a user has to rely on the output table access control (Section 6.2) of the providing f-unit in which datasets might eventually end up in an input

[1]Only for read operations, though.

table. Thus, given any user $u \in \mathcal{U}$ and f-unit $f \in \mathcal{F}$, the contents of an input table are assumed to be shared across the dimension $\mathcal{U}$ of users:

$$\forall t \in it(f,u), d \in data(t), u \in \mathcal{U} \setminus \{own(d)\}\colon sh_{\mathcal{U}}(own(d), u, d) \tag{8.6}$$

By Equation 8.5 and Equation 8.6, we basically incapacitate the user by shifting sharing responsibility solely to the f-unit dimension – in contrast to the requirement that both dimensions have to agree on a sharing of a particular data entity. However, an f-unit can only share datasets that it was explicitly provided with by either the dataset's owner or another sharing f-unit. We can regard both cases as the implicit affirmation of the user based on his personal trust assessment for potential sharing in a manner the f-unit may specify on own behalf.

By the definition of an input table t, all data entities $d \in data(t)$ are intentionally shared on behalf of the providing f-unit $src(d)$. Hence, for all input tables of f-unit $f \in \mathcal{F}$ running in the scope of user $u \in \mathcal{U}$, we assume

$$\forall t \in it(f,u), d \in data(t)\colon sh_{\mathcal{F}}(src(d), f, d) \tag{8.7}$$

that is, an intended sharing in the principal dimension of f-units $\mathcal{F}$.

Model Instantiation. Using the previously introduced predicates, we propose an instantiation of the formal model that we call *sandbox.* The sandbox *sb* discriminates a request $r \in \mathcal{R}$ according to a common operation-set in relational database systems:

$$op\colon \mathcal{R} \to \{\texttt{SEL}, \texttt{INS}, \texttt{UPD}, \texttt{DEL}\}$$

Here, *op* is defined to restrict a single request to a single operation, while this is not necessarily the case in practice, e.g., an update operation `UPD` might incorporate a select operation `SEL` when updating data that was previously read or evaluated according to some condition. However, we assume $op(r)$ to be well-defined for all $r \in \mathcal{R}$ – if necessary, r has to be split up into sub-requests.

We define the sandbox

$$sb\colon \mathcal{R} \times \mathcal{U} \times \mathcal{F} \to \{0,1\}$$

according to the following semantics:

$$sb(r,u,f) \mapsto \begin{cases} \overbrace{\forall t \in \mathit{aff}_{\mathcal{T}}(r)\colon \underbrace{t \in lt(f)}_{\text{(B)}} \vee \underbrace{t \in it(f,u)}_{\text{(C)}}}^{\text{(A)}} & , op(r) \in \{\texttt{SEL}\} \\ \overbrace{\forall t \in \mathit{aff}_{\mathcal{T}}(r)\colon t \in lt(f)}^{\text{(D)}} \wedge & \\ \quad \underbrace{\forall d \in \mathit{aff}_{\mathcal{D}}(r)\colon own(d) = u}_{\text{(E)}} & , \text{else} \end{cases} \tag{8.8}$$

Intuitively, a read-only request is considered permissive (A), if it operates only on own local tables (B) or on input tables (C) – in other words, an f-unit may read data from own local tables and from own input tables that are wired according to the currently connected user. This scenario is covered by our query sandbox implementation (Section 4.2), which ensures that only tables associated to a particular f-unit are accessible and thus possibly affected.

On the other hand, a modification request is considered permissive, if it operates only on own local tables (D) and if all affected datasets are owned by the currently connected user (E). As only local tables can be modified[2], the owner invariant (Section 4.1) restricts authorized changes to the particular dataset owner and thereby implements the *sb* semantics for modification requests.

8.2 Soundness

In this section, we investigate whether our sandbox semantics presented in Equation 8.8 is a valid instantiation of the formal access control model as defined in Equation 3.4.

We thus claim that

$$\forall r \in \mathcal{R}, u \in \mathcal{U}, f \in \mathcal{F} \colon sb(r,u,f) \Rightarrow \mathit{req\_valid}(r,u,f)$$

i.e., that the sandbox is at least as restrictive as the presented formal model. By the introduced instantiation, we can derive for the particular parts of *sb*:

(B) The affected f-unit for any local table dataset is the associated f-unit itself:

$$\begin{aligned} &t \in \mathit{lt}(f) \\ \Rightarrow_{(8.3)}\ &\forall d \in \mathit{data}(t) \colon \mathit{aff}_{\mathcal{F}}(d) = f \end{aligned}$$

(B) The affected user for any local table dataset omits any restriction but relies on the particular f-unit:

$$\begin{aligned} &t \in \mathit{lt}(f) \\ \Rightarrow_{(8.5)}\ &\forall d \in \mathit{data}(t) \colon \mathit{own}(d) = u \vee \mathit{sh}_{\mathcal{U}}(\mathit{own}(d), u, d) \\ \Rightarrow_{(8.4)}\ &\forall d \in \mathit{data}(t) \colon \mathit{aff}_{\mathcal{U}}(d) = u \vee \mathit{sh}_{\mathcal{U}}(\mathit{aff}_{\mathcal{U}}(d), u, d) \end{aligned}$$

(B) Both the above implications of (B) yield together:

$$\forall d \in \mathit{data}(t) \colon \mathit{aff}_{\mathcal{F}}(d) = f \ \wedge\ (\mathit{aff}_{\mathcal{U}}(d) = u \vee \mathit{sh}_{\mathcal{U}}(\mathit{aff}_{\mathcal{U}}(d), u, d))$$

(C) Input table datasets are shared by the providing f-unit:

$$\begin{aligned} &t \in \mathit{it}(f,u) \\ \Rightarrow_{(8.7)}\ &\forall d \in \mathit{data}(t) \colon \mathit{sh}_{\mathcal{F}}(\mathit{src}(d), f, d) \\ \Rightarrow_{(8.2)}\ &\forall d \in \mathit{data}(t) \colon \mathit{sh}_{\mathcal{F}}(\mathit{aff}_{\mathcal{F}}(d), f, d) \end{aligned}$$

(C) The affected user for any input table dataset omits any restriction but relies on the providing f-unit and its output table invariants:

$$\begin{aligned} &t \in \mathit{it}(f,u) \\ \Rightarrow_{(8.6)}\ &\forall d \in \mathit{data}(t) \colon \mathit{own}(d) = u \vee \mathit{sh}_{\mathcal{U}}(\mathit{own}(d), u, d) \\ \Rightarrow_{(8.4)}\ &\forall d \in \mathit{data}(t) \colon \mathit{aff}_{\mathcal{U}}(d) = u \vee \mathit{sh}_{\mathcal{U}}(\mathit{aff}_{\mathcal{U}}(d), u, d) \end{aligned}$$

[2]Local tables are the only entities for which write permissions are granted.

(C) Both the above implications of (C) yield together:

$$\forall d \in \mathit{data}(t)\colon \mathit{sh}_{\mathcal{F}}(\mathit{aff}_{\mathcal{F}}(d), f, d) \ \wedge\ (\mathit{aff}_{\mathcal{U}}(d) = u \vee \mathit{sh}_{\mathcal{U}}(\mathit{aff}_{\mathcal{U}}(d), u, d))$$

(D) If only local tables of the same f-unit f are affected, the overall affected f-unit is f itself:

$$\begin{aligned} & \forall t \in \mathit{aff}_{\mathcal{T}}(r)\colon t \in \mathit{lt}(f) \\ \Rightarrow_{(8.3)}\ & \forall t \in \mathit{aff}_{\mathcal{T}}(r)\colon \forall d \in \mathit{data}(t)\colon \mathit{aff}_{\mathcal{F}}(d) = f \\ \Rightarrow_{(8.1)}\ & \forall d \in \mathit{aff}_{\mathcal{D}}(r)\colon \mathit{aff}_{\mathcal{F}}(d) = f \end{aligned}$$

(E) If all affected datasets are owned by user u, the overall affected user is u itself:

$$\begin{aligned} & \forall d \in \mathit{aff}_{\mathcal{D}}(r)\colon \mathit{own}(d) = u \\ \Rightarrow_{(8.4)}\ & \forall d \in \mathit{aff}_{\mathcal{D}}(r)\colon \mathit{aff}_{\mathcal{U}}(d) = u \end{aligned}$$

By the implications of (B) and (C), we can derive the overall semantics of (A), i.e., of a SEL operation:

$$\begin{array}{lll} & \text{(A)} & \\ \Leftrightarrow & \forall t \in \mathit{aff}_{\mathcal{T}}(r)\colon & \text{(B)} \vee \text{(C)} \\ \Rightarrow_{(8.1)} & \forall d \in \mathit{aff}_{\mathcal{D}}(r)\colon & (\mathit{aff}_{\mathcal{F}}(d) = f \vee \mathit{sh}_{\mathcal{F}}(\mathit{aff}_{\mathcal{F}}(d), f, d)) \wedge \\ & & (\mathit{aff}_{\mathcal{U}}(d) = u \vee \mathit{sh}_{\mathcal{U}}(\mathit{aff}_{\mathcal{U}}(d), u, d)) \end{array} \tag{8.9}$$

Furthermore, for a request being an INS, UPD, or DEL operation, we have:

$$\begin{array}{ll} & \text{(D)} \ \wedge\ \text{(E)} \\ \Leftrightarrow & \forall d \in \mathit{aff}_{\mathcal{D}}(r)\colon \mathit{aff}_{\mathcal{F}}(d) = f \ \wedge\ \mathit{aff}_{\mathcal{U}}(d) = u \end{array} \tag{8.10}$$

If we reconsider the formal model as in Equation 3.4, the instantiation to the $\langle \mathcal{U}, \mathcal{F} \rangle$ universe gives:

$$\begin{aligned} \mathit{req\_valid}(r, u, f) \mapsto \forall d \in \mathit{aff}_{\mathcal{D}}(r)\colon \quad & (\mathit{aff}_{\mathcal{U}}(d) = u \vee \mathit{sh}_{\mathcal{U}}(\mathit{aff}_{\mathcal{U}}(d), u, d)) \wedge \\ & (\mathit{aff}_{\mathcal{F}}(d) = f \vee \mathit{sh}_{\mathcal{F}}(\mathit{aff}_{\mathcal{F}}(d), f, d)) \end{aligned}$$

Both branches of our implemented sandbox, Equation 8.9 and Equation 8.10, reflect that for every dimension, either own or explicitly shared datasets are affected – and thus satisfy the formal model. □

Chapter 9

Future Work

This chapter revises particular technical aspects that are considered problematic or worth further investigation. Moreover, suitable existing approaches are presented as related work.

9.1 Advanced Owner Management

The extensions of SAFE do not offer the possibility for handling *delegation*, i.e., users cannot temporarily or permanently transfer their ownerships on a certain set of data entities: The owner invariant (Section 4.1) maintains the owner information throughout the lifetime of a particular row. However, there are scenarios where delegation could be a desirable feature, even regardless of a potential loss of accountability properties. Moreover, one wants to account for a group of users that are jointly responsible for a common set of data in the scope of *collaboration*.

In addition to the problem of multiple or deviant owners per dataset, the owner invariant has to be maintained even for output tables in the presence of malicious f-units: So far, an f-unit may arbitrarily choose the returned owner for a particular dataset. However, some information flow approach on output tables might be needed in order to track returned rows down to the set of actually affected users.

9.2 F-unit Sandboxing

As required by the sandbox assumption *SB* (Section 3.1 and, in particular, Section 3.2.1), f-unit information flow shall be limited to the channel between f-unit, CRM, and database. In other words, an f-unit must not be able to communicate with other f-units or even foreign entities.

Connections in *Flash*-based applications are restricted to their originating domain per default [1]. Similar functionality for preventing outbound connections in Web applications is provided by the *RequestPolicy* browser extension [22]. In contrast to the W3C *Cross-Origin Resource Sharing* [26] that can be used to relax a browser's same origin policy, the opposite approach that would additionally limit the connectivity of delivered content seems to be still missing. However, there are projects that might be worth further evaluation in this context – the *Google Caja* project [6] for example claims to introduce a fine-grained security model for a "safe" embedding of third-party JavaScript code. In general, static or dynamic JavaScript code analysis [7] is an option that is worth being mentioned, too.

A short-term solution for preventing leakage across f-units while neglecting the possibility of sending data to foreign servers could consist of *iframes* that hold f-units hosted on distinct subdomains: Every f-unit is hosted on its own subdomain and instead of a `<div>` tag as container, the f-unit is displayed as a standalone page inside an iframe. The same origin policy, which is implemented by all recent and common browsers [27, 5], will thus prevent any attempts of DOM accesses across f-units. However, in contrast to `divs`, `iframes` do not support an automatic sizing according to its content and would thus only allow a fixed layout. For this reason, a dedicated communication channel, e.g., by using the fragment identifier [13], would be needed that allows to announce the children's inner dimensions[1]. The parent iframe may then resize its child appropriately.

9.3 Securing the Session ID

The management of users and sessions presented in Section 7.1 follows the basic and well-known approach of cookies carrying a session id. This is possible because of our strong assumptions on the behavior of an f-unit. In particular, the sandbox assumption may be considered too strong to be satisfiable – preventing a malicious f-unit from leaking sensitive information (the *sid*) is still a problem to be solved. If we weaken or even drop the sandbox assumption, a reasonable approach could be to limit the consequences of a possible *sid* leakage: If a *sid* is bound to and only valid for a particular f-unit, an adversary can abuse a stolen session id only for queries that would be possible directly for the malicious f-unit anyhow. Additionally, the *sid* could be bound to a secret that is only available in the current browser instance and hidden even from the f-units themselves.

However, when binding a cookie to an IP address (Section 7.1), there are several problematic technical scenarios that are worth to mention:

NAT If the attacker and the client reside in the same network that is using *network address translation*, our IP-binding signature will render useless.

Proxy The same holds if both parties are using an anonymous Web proxy that is not transmitting the `X-Forwarded-For`-header.

IP-Spoofing In an environment allowing for *IP address spoofing*, an adversary could abuse the cookie as well.

[1]`http://stackoverflow.com/questions/153152/resizing-an-iframe-based-on-content/362564#362564`

DHCP Our approach will break sessions for clients using dynamic addresses that can legitimately change in between.

A common proposal to overcome these limitations is to use the SSL session key instead of the IP address for binding the *sid* to a browser instance [11]. However, this approach might suffer from the possibility of renegotiation handshakes during the session[2].

While binding the *sid* to the client IP address provides protection against remote abuse of a leaked *sid* in principle, we suggest several techniques that are considered suitable for implementing the semantics of a per-f-unit or per-browser-instance *sid* in the following. While we focus on the problem of the cookie residing in the same browser environment, more generic approaches on the topic of securing cookies are introduced in existing work [11, 14, 17, 28].

In order to prevent leakage of session information, it is reasonable to restrict its validity to a corresponding secret available only in the client's browser environment. Upon visit, client-side JavaScript code could generate a random *nonce*, which is announced to the CRM using a secure and authenticated channel and thereby bound to the session. As long as this *nonce* is kept hidden in memory on the browser side, its value can be used to sign queries upon request of an f-unit before passing the queries to the CRM. In addition, this approach can easily be extended to an additional set of per-f-unit nonces that are handed out to each f-unit's object upon creation. This approach ensures that all leaked nonces will render useless without knowing the secret that is only locally available in the browser.

The previous approach requires an initial confidential channel between the client and the CRM. Meeting this confidentiality requirement could be technically feasible, whereas this could be undesirable. Hash-chains as introduced by Lamport [9] use a one-way function $H(\cdot)$, which is initially applied up to n times on a chosen secret s_0:

$$s_i = \underbrace{H(\dots H(}_{i \text{ times}} s_0)\dots) = H^i(s_0) = H(s_{i-1}) \qquad , 0 < i \leq n$$

A client-side script could now announce its computed value of s_n together with the session id to the CRM, keeping all other parts secret – the root image s_0 (and n), in particular. For an upcoming jth query, the f-unit could then request a signature for its query, which is based on s_{n-j} that is recalled or recomputed using s_0 in the client's browser. While it is considered infeasible to compute the pre-image s_{n-j} on the basis of a seen s_{n-j+1} only, the validity of a new token is publicly verifiable by comparing $H(s_{n-j})$ with the previously received s_{n-j+1}.

A second approach that does not rely on a confidential exchange of information is inspired by the accountability technique for randomized systems called *CSAR* [2]. Adapted to our scenario with weakened requirements, one could extract the possible usage of low-exponent RSA as one-way trapdoor function. When picking n as the product of two suitable primes p and q, one would define the low-exponent RSA $f_n(\cdot)$ as:

$$f_n(x) = x^3 \mod n$$

[2]`http://support.microsoft.com/kb/257591/EN-US`

Assume a client-side script chooses an initial seed value s_0 that is publicly (but authentic) announced along with the RSA modulus n, while p and q are kept secret in the browser environment. Then, the client is able to compute tokens of the sequence

$$s_i = f_n^{-1}(s_{i-1})$$

while this is assumed to be infeasible for parties not aware of the factorization of n into p and q. However, the origin of each produced s_i is easily verifiable using:

$$f_n(s_i) = s_{i-1}$$

This provides another possibility of binding the session of a user to its current browser environment – in addition to the session information, each query reaching the CRM has to carry an s_i that was certainly requested from the client browser by the f-unit. Furthermore, this approach is considered extensible with respect to supporting per-f-unit tokens as well. For implementing $f(\cdot)$, other trapdoor functions may as well be suitable.

The presented approaches are intended to allow for a weakened sandbox assumption. However, the resulting assumption can only be slightly weakened: Only the leakage of the session id is tolerated, while an assumption on the secrecy of private data in client-side scripts has to be added. Furthermore, the leakage of other user-related data, the possibility of accessing another f-unit's DOM, and all the other possible sandbox-related threats remain unaffected. Another main problem consists in using the browser instance as additional authenticity factor to the cookie. This disqualifies the approaches for usage in an initial or static queries context (Section 7.1), as code executed at PHP-runtime (before being delivered to the client) cannot use any JavaScript functionality. Other problems of technical nature are the storage of secrets beyond page reloads and the implementation of session-resets if the CRM misses some s_i in between. For these reasons, further evaluation or even implementation of one of the presented techniques are left for future work.

9.4 Mitigating Cross-Site Request Forgery

Even in scenarios that reliably bind a cookie to a browser instance and thus to a particular user, an adversary might be able to trick the user into performing unintended actions by a *Cross-Site Request Forgery.*

In the scope of a CSRF, an adversary can exploit various techniques that result in self-crafted requests issued by the victim's browser to the target application. As cookies are seamlessly included in any request to the particular domain in general, the potential malicious request seems to originate from the authenticated source, as indicated by the cookie.

Amongst others, a promising mitigation technique could be the inclusion of a verifiable challenge or the cookie value itself in hidden form fields [24] – basically preventing an adversary from crafting valid form-submitting requests. As in SAFE, forms are parsed and automatically augmented anyhow, this approach could be adapted with manageable effort. Furthermore, advanced security methods such as link signing or encryption could be supported, allowing for prevention of parameter tampering, e.g., by means of injection, in general.

Chapter 10

Summary & Conclusions

In this thesis, we have payed particular attention to an access control abstraction that complies with the demands of contemporary RIAs. The SAFE framework addresses the common pattern of mashup Web applications, which can be composed out of functional self-contained f-units. Additionally, the design of SAFE comprises the presence of a CRM that is considered well-suited for central database query monitoring and policy enforcing.

By this means, we have derived an attacker model and a generic access control abstraction, yielding a clear notion of data separation between and across both users and f-units. On this basis, we have extended SAFE by an ownership-based access control approach that maintains well-defined scopes on the database layer and thereby meets the derived data separation requirement. On top of this, by automatically maintained database table invariants, the developer of an f-unit may shift the responsibility of additional access control policies to the database layer as well.

Moreover, we have addressed the possibility of f-unit collaboration and runtime extensibility by introducing the concept of wiring, which allows f-units to receive and extend data through particular interface tables. In particular, we have implemented foreign key functionality with dependency tracking and seamless cleanup even on input tables, allowing f-units to extend arbitrary foreign datasets according to their own functionality. We have shown that, in addition to our basic access control approach, our wiring approach resembles the notion of data sharing in the scope of our previously introduced access control model.

While we consider our overall approach as well-suited to the particular SAFE scenario, we ultimately had to rely on a user's perception of the trustworthiness of an f-unit. In particular, we avoided to base our approach on information flow or code analysis tools, i.e., for HTML, JavaScript, or SQL – thereby leaving still room for further improvements.

List of Figures

List of Listings

Bibliography

[1] Adobe Systems Inc. Cross-domain policy for Flash movies. `http://kb2.adobe.com/cps/142/tn_14213.html`.

[2] Michael Backes, Peter Druschel, Andreas Haeberlen, and Dominique Unruh. CSAR: A Practical and Provable Technique to Make Randomized Systems Accountable. In *Proceedings of the Network and Distributed System Security Symposium*. The Internet Society, 2009.

[3] Philip A. Bernstein, Jayant Madhavan, and Erhard Rahm. Generic Schema Matching, Ten Years Later. *PVLDB*, 4(11):695–701, 2011.

[4] Silvana Castano, Maria Grazia Fugini, Giancarlo Martella, and Pierangela Samarati. *Database Security*. Addison-Wesley & ACM Press, 1995.

[5] Google Inc. Browser Security Handbook. `http://code.google.com/p/browsersec/wiki/Part2#Same-origin_policy`.

[6] Google Project Hosting. google-caja – Compiler for making third-party HTML, CSS and JavaScript safe for embedding. `http://code.google.com/p/google-caja/`.

[7] Seth Just, Alan Cleary, Brandon Shirley, and Christian Hammer. Information Flow Analysis for JavaScript. In *Proceedings of the 1st ACM SIGPLAN international workshop on Programming language and systems technologies for internet clients*, PLASTIC '11, pages 9–18. ACM, 2011.

[8] D. Kristol and L. Montulli. HTTP State Management Mechanism. RFC 2109 (Proposed Standard), February 1997. Obsoleted by RFC 2965.

[9] Leslie Lamport. Password Authentication with Insecure Communication. *Commun. ACM*, 24(11):770–772, November 1981.

[10] Maurizio Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the Twenty-first ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 233–246. ACM, 2002.

[11] Alex X. Liu, Jason M. Kovacs, Chin-Tser Huang, and Mohamed G. Gouda. A Secure Cookie Protocol. In *Proceedings of the 14th IEEE International Conference on Computer Communications and Networks*, pages 333–338, October 2005.

[12] Microsoft Developer Network Library. Mitigating Cross-site Scripting With HTTP-only Cookies. `http://msdn.microsoft.com/library/en-us/ms533046.aspx`.

[13] Microsoft Developer Network Library. Secure Cross-Domain Communication in the Browser. `http://msdn.microsoft.com/en-us/library/bb735305.aspx`.

[14] Steven J. Murdoch. Hardened Stateless Session Cookies. In *Security Protocols XVI - 16th International Workshop*, pages 93–101. Springer, 2008.

[15] Qun Ni and Elisa Bertino. xfACL: An Extensible Functional Language for Access Control. In *Proceedings of the 16th ACM symposium on Access control models and technologies*, SACMAT '11, pages 61–72. ACM, 2011.

[16] Oracle. MySQL 5.1 Reference Manual. `http://dev.mysql.com/doc/refman/5.1/en/`.

[17] Joon S. Park and Ravi Sandhu. Secure Cookies on the Web. *IEEE Internet Computing*, 4(4):36–44, 2000.

[18] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems (3rd ed.)*. McGraw-Hill Publ. Comp., 2002.

[19] Raphael M. Reischuk. The SAFE Activation Framework for Extensibility: Official SAFE User Manual. `http://safe-activation.org/publications/safe-manual.pdf`.

[20] Raphael M. Reischuk, Michael Backes, and Johannes Gehrke. SAFE Extensibility for Data-Driven Web Applications. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 799–808. ACM, 2012.

[21] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 551–562. ACM, 2004.

[22] Justin Samuel. Firefox addon for privacy and security – RequestPolicy. `https://www.requestpolicy.com/`.

[23] Bruce Schneier. *Secrets and Lies – Digital Security in a Networked World.* Wiley Publishing, Inc., 2004.

[24] The Open Web Application Security Project (OWASP). Cross-Site Request Forgery (CSRF) Prevention Cheat Sheet. `https://www.owasp.org/index.php/Cross-Site_Request_Forgery_%28CSRF%29_Prevention_Cheat_Sheet`.

[25] Dobromir Todorov. *Mechanics of User Identification and Authentication: Fundamentals of Identity Management.* Taylor & Francis, 2007.

[26] World Wide Web Consortium. Cross-Origin Resource Sharing. `http://www.w3.org/TR/cors/`.

[27] World Wide Web Consortium. Same Origin Policy. `http://www.w3.org/Security/wiki/Same_Origin_Policy`.

[28] Donghua Xu, Chenghuai Lu, and Andre Dos Santos. Protecting Web Usage of Credit Cards Using One-Time Pad Cookie Encryption. In *Proceedings of the 18th Annual Computer Security Applications Conference*, ACSAC '02, pages 51–. IEEE Computer Society, 2002.