CHALMERS
UNIVERSITY OF TECHNOLOGY

SecArchUnit: Extending ArchUnit to support validation of security architectural constraints

Master's Thesis in Software Engineering

MARCUS RANDEVIK
PATRIK OLSON

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2020

Master's thesis 2020:NN

SecArchUnit: Extending ArchUnit to support validation of security architectural constraints

MARCUS RANDEVIK
PATRIK OLSON

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
Gothenburg, Sweden 2020

SecArchUnit: Extending ArchUnit to support validation of security architectural constraints
MARCUS RANDEVIK
PATRIK OLSON

Supervisor: Riccardo Scandariato, Computer Science and Engineering
Examiner: Michel Chaudron, Computer Science and Engineering

Master's Thesis 2020:NN
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Todo list

SecArchUnit: Extending ArchUnit to support validation of security architectural constraints
MARCUS RANDEVIK
PATRIK OLSON
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Keywords: lorem, ipsum, dolor, sit, amet, consectetur, adipisicing, elit, sed, do.

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Name Familyname, Gothenburg, Month Year

Contents

List of Figures

List of Tables

1

Introduction

In the age of an ever more digitalized world, ensuring the security of software systems becomes an increasingly critical task. The increased importance of security is true in particular, as systems are being developed basced on the principle of permanent connectivity [1]. Services ranging from those offered by governmental agencies to that of social media are always connected to the open internet, potentially creating a large surface of attack. Although the importance of a securely designed system is widely known, developing a secure system is a challenging task. Far from all software engineers are security specialists, or for that matter, particularly educated in security aspects [2].

Introducing a weakness into a system can be done in every part of the software development lifecycle. An early introduction often leads to a more costly fix as the scope of the flaw is increased [3]. Typically, weaknesses are categorized as either implementation bugs (e.g. buffer overflow) or design flaws (e.g. client-side validation) where the later commonly has a more extensive scope and is the responsibility of the architect [4]. However, this binary division might be too simplified as it assumes a secure design is implemented correctly, meaning that there is no discrepancy between the "intended architecture" and the actual one found in the source code. Jasser [5] considers violations of the architectural security design as a third type of category with a comparable impact to that of design flaws.

Deviations from the intended architecture are not unique to the initial implementation of a system. Over time, subsequent changes made to the system, often due to new requirements, frequently lead to further disparency called software erosion [6]. Proponents of the agile methodology also somewhat worsen the problem of erosion as the reduction of upfront design severely hinders the design of the architecture [7]. Many of the tools and techniques produced by academia to try and remedy the frequent violations of architectural design have failed to gain wide adoption in the industry [6]. Of the few that have, architectural design documentation is the most prominent and widely included in numerous software processes. While the technique is well adopted, performing it formally to allow for automatic compliance monitoring is seldom done, causing developers to rely on the less scalable method of manual reviews [6].

A recently developed tool called ArchUnit[1] has provided architects with the capability of validating architectural constraints using Java unit testing frameworks. While ArchUnit is not the first tool developed for architectural conformance checking, it is possibly the first to leverage already existing testing infrastructure (unit testing) in a manner that allows for testing over time. In its current form, ArchUnit has not provided any explicit functionality to test security constraints. Thus, the objective of this thesis is to explore the feasibility of expressing and enforcing typical security architectural constraints trough ArchUnit.

1.1 Scope

This study focuses on architectural measures that aid in fulfilling a security goal. Moreover, these measures should be possible to enforce through static analysis of Java bytecode. Any measure related to the configuration (of an application or operating system), the file-system, or other run-time properties that cannot be validated through static code analysis is deemed to be out of scope.

1.2 Research questions

To fulfill the objective of the study, the following research questions have been defined:

- RQ1: What architectural security constraints can be validated using the tool?
- RQ2: What modifications can be made to the tool in order to facilitate the validation of additional constraints?

1.3 Research contribution

This study aims to show how architectural security constraints can be validated with the help of a static analysis tool. The thesis demonstrates applications of the tool to several open source systems, over time, in an evaluation of its efficacy and precision in terms of detecting violations of constraints.

1.4 Limitations

This thesis, and the modified version of ArchUnit, is not aimed at being a complete solution for all security architectural constraints. Instead, the study is performed to provide an initial evaluation of the possibility of using ArchUnit as an alternative to already existing techniques of static conformance checking.

Although the principle of ArchUnit may very well apply to programming languages

[1] https://www.archunit.org/

other than Java, the limited scope of the thesis makes it unfeasible to provide functionality to analyze source-code in additional languages.

1.5 Thesis outline

The remainder of this report is structured as follows. Outline chapters

Chapter 2: Provides a general background to the topic of software architecture and architectural security constraints. In addition, the ArchUnit framework is introduced and compared to previously developed tools for architectural conformance.

Chapter 3: design of our evaluation, how systems are chosen, the protocol for testing the tool, how to assert the ground truth, process of mapping constraints to rules

Chapter 4: composing the architectural constraints, our final constraints, what they prevent and how they are expressed

Chapter 5: expressing and enforcing constraints with the tool, the identification of missing information / tool features

Chapter 6: results from the empirical validation

Chapter 6: discussion, answer our research questions

Chapter 7: conclusion

2

Background

This chapter...

2.1 Software architecture

Although not intended to provide any contribution to the definition of software architecture, our work relies on a shared understanding of what constitutes as architecture

Initially considered by Djikstra in the organisation of systems and by Parnas [8, 9], the term was introduced by [10] later to be segmented by [11]

Today, there are mainly two defintions used where one is concernd with the structure of system components and their relations [12] whereas the other focuses on the major decision [13]

2.2 Architectural security constraints

[14], [1], [15],

Security goals, CIAA...

2.3 Static analysis

Information flow analysis [16] which we lend ideas from for our extension

2.3.1 ArchUnit

Overview of functionality

ASM, class/method visitors

Explain concept of code units, members, accesses (target, target owner)

2.4 Related Work

2.4.1 Architectural conformance monitoring

[17], [18], [19], [20], [6], [21], [22], [23],

2.4.2

3

Methodology

This chapter describes the adopted method for collecting relevant constraints, relating these to the common security-goals of CIAA, and later mapping them to functionality within ArchUnit. Second, this chapter presents the validation plan for expressing security constraints with ArchUnit, both by comparing it to industry used static analysis tools and a separate analysis focusing on constraints that could only be expressed in ArchUnit.

3.1 Identification of design constraints

Add text

3.1.1 Data collection

The relevance of the security architectural constraints included in the study was ensured by performing a review of security measures and common weaknesses and compiling the result to a list of constraints. Completeness was not the primary goal of the review, but rather to provide a set of constraints derived from previous knowledge. Presented below are the three sources used to form the final list.

CAWE catalog: The Common Architectural Weakness Enumeration catalog [24] details 224 common weaknesses in security architectures. Each entry has a description of the weakness and exemplifications of how it could manifest itself in the source code, when applicable. In some entries, there are recommendations on what techniques can be used to detect the weakness, along with mitigation strategies.

Security patterns: Similar to the usage of general design patterns made famous in [25], security patterns provide a reusable and domain-independent solution to a known problem. More specifically, this study focused on security patterns for the design phase, as defined in [2]. While the security pattern repository[1] lists over 170 security patterns, not all are provided with sufficient detail or at the appropriate level of abstraction. As a result, the report by Scandariato et al. [26] which provides a filtered list of patterns.

Security rules: Architectural security rules constrain the implementation of a system while being less solution-oriented compared to security patterns. Eden and

[1] `http://sefm.cs.utsa.edu/repository/`

Kazman differentiate architectural security rules from those defined on a level of source code based on two criteria, locality and intension/extension [27]. Architectural rules are both non-local and intensional, meaning that they affect all or several parts of the system while having “infinitely-many possible instances”. In [5], Jasser presents a catalog of architectural security rules. Although the entire catalog of 150 security rules is not yet available, the initial list of 22 included in the paper was used in our study.

3.1.2 Filtering

Starting from each of the three sources of architectural constraints described in Section 3.1.1, the first step of the process, shown in Figure 3.1, was to select the entries that could be formulated as enforceable constraint in the context of our project. The criteria for inclusion were the following:

1. The entry must be related to the architectural design of a system, i.e., non-local and intensional (as described in Section 3.1.1).
2. It must be possible to enforce the entry through static analysis. An example on a non enforceable constraint is “No two instances of a microservice are deployed on the same machine” [5], as the number of machines deployed is a dynamic property.
3. Although somewhat included in the first criterion as it is a local issue, an entry must not relate to the correctness or best practice of the implementation of an algorithm. Examples include the practice of using session tokens with time-limited validity.
4. The entry must only relate to the system under design, thus ignoring the correctness and security of any external dependencies. An example can be found in SonarQube where the usage of a version of a library with known vulnerabilities is reported as a weakness.
5. The entry must not be dependent on externally defined data. A common example is that of user permissions where the mapping between a regulated function and a users rights is performed using an access control list, external to the source code.
6. Additionally, we deemed entries defined as the absence of certain functionality as less valuable due to the increased difficulty of enforcing them [15]. An example of such an entry is “The system must not provide functionality to decrypt secured log messages”, as defined in [5].

Previous research on design notations of secure systems have shown a skew towards confidentiality and integrity while having little or no support for availability and accountability. We considered it necessary for the final list of constraints to include all of the security goals. As a consequence, once we had selected the applicable entries, we categorized them according to the security goals of CIAA, ensuring that

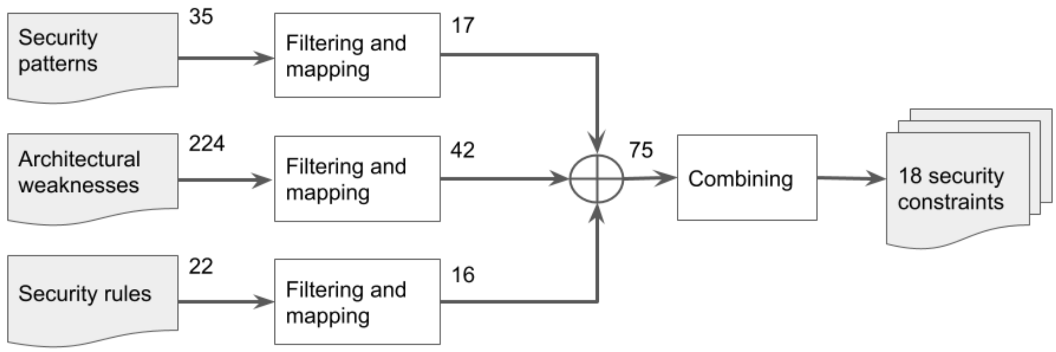

Figure 3.1: Overview of the process of mapping the three sources to constraints.

the final list of constraints covered all security goals.

The last part of compiling a list of security architectural constraints involved combining the selected entries to remove duplicates and group similar concepts. Duplication involved both a single source having several entries, such as CAWE having input validation weakness for multiple tools and technologies (e.g., SQL, LDAP) and different sources having entries for the same concept, such as the security pattern input guard and the previously mentioned input validation weaknesses. Grouping similar concepts also allowed for the constraints to be more general, thus making them applicable for a broader set of systems.

3.2 Evaluation of the proposed approach

This work followed the research design described in [28] as a "solution-seeking + sample study." The first part, solution-seeking, covers the overall idea of proposing a solution to an identified problem, namely integrating security architectural constraints within already established testing infrastructure. The second part, a sample study, aimed to achieve generalizability by composing security architectural constraints that were not tied to a specific domain and later evaluating the performance of the tool when applied to several systems.

3.2.1 Comparison to reference tools

In order to test how reliably SecArchUnit can validate the constraints, we perform a comparison with two static analysis tools widely adopted in industry: SonarQube and PMD. While these tools have a multitude of built-in rules, none of these rules can be used to enforce the architectural security constraints presented in this thesis. However, both tools are extensible, allowing a developer to define custom rules using their respective APIs.

describe limitations

An initial assessment of the tools determined that constraints 1-5 are possible to implement and validate using SonarQube and PMD. Constraints 6-7, however, track the flow of information in the system, which neither of these tools inherently support. While it should be possible to extend these tools with such analysis, doing so is outside the scope of this thesis. Hence, the first five constraints are evaluated in SecArchUnit, SonarQube and PMD, whereas the final two constraints are evaluated solely in SecArchUnit.

3.2.2 Evaluation metrics

The included tools will be evaluated in two aspects: performance and usability. Performance relates to the ability to detect violations of a constraint reliably, whereas usability relates to time required to implement the constraints and how well the tool captures the concepts used in architectural descriptions. The following sections provide further detail for each aspect.

3.2.2.1 Performance

The performance metrics that were chosen to represent how reliably the tools detect violations of security constraints are precision P and recall R. They are defined as follows:

$$P = TP/(TP + FP)$$
$$R = TP/(TP + FN)$$

A true positive (TP) is a report of a line containing a security constraint violation corresponding to an equal in the ground truth. A false positive (FP) refers to the report of a violation not listed in the ground truth. Finally, a false negative (FN) is the failure to report a violation listed in the ground truth. As the mechanism of reporting a violation differed among the included tools, some tolerance was employed regarding the line at which a violation was reported. For example, a method that receives user input could either be reported inside the method body or at its declaration.

The imbalance between the designer, who needs to ensure that every single aspect of a system is secure, and the attacker, who needs to succeed only once, influences the relative importance of the two metrics. Precision, which represents the probability of a reported violation, indeed being a violation, is relevant as the time allocated for fixes is limited. However, recall, which represents the probability of a violation being detected, is paramount as a single missed security constraint violation could be exploited. Consequently, recall was given greater weight when evaluating the results of the tools.

3.2.2.2 Usability

A significant aspect of applying new tools to a project is the time needed for training. Although SecArchUnit can integrate into existing testing infrastructure and CI/CD pipelines, the implementation of appropriate security architectural constraints is a non-trivial issue. Logging the time needed to implement the constraints included in the study was used as an estimate of the effort needed to construct new constraints. The first two constraints were considered as learning examples to reduce the bias introduced by us having more experience of using SecArchUnit.

Additional usability aspects concern how custom rules are defined in the different tools; whether the rules are specified at a proper level of abstraction for expressing security architectural constraints, and how conveniently the rules can be applied to a system.

3.2.3 Subjects of analysis

Several requirements were formed to guide the selection of projects to be included in the evaluation of our study. While some were necessary due to the languages supported by ArchUnit, others served to decrease the threats to validity. Detailed below is the final list of requirements for inclusion:

1. **The project must be open source**. The constraints defined in SecArchUnit cannot be applied to a system without access to the source code.

2. **The source code must be written in Java**. As SecArchUnit does not support any other language, a strict requirement had to be made regarding the language.

3. **The project must be previously used in literature concerning security**. Using projects already analyzed in previous literature would reduce the bias of the ground truth.

4. **The project must include some form of architectural description**. Architectural description would allow the constraints to be appropriately placed on a system in regards to the security requirements which it has been developed for.

Based on the presented criterion, three systems were selected for the evaluation: JPetStore, ATM Simulation and iTrust. All projects had previously been included in a study on secure data flow by Peldszus et al [29]. A summary of the characteristics for each system can be seen in Table 3.1. A more detailed description of each system is seen below:

JPetStore, originally designed as an example of how to use the J2EE framework, is built on top of MyBatis 3, Spring and the Stripes framework. The application is a minimal implementation of an online pet store. In addition to the inclusion of JPetStore in security litterateur, It has been used both as an industry benchmark application as well in studies on application performance [30].

Table 3.1: Characteristics of the projects used in the evaluation, adapted from [29]. **lloc** = logical lines of code.

Project	lloc	classes	methods
JPetStore	1 221	17	277
ATM Simulation	2 290	57	225
iTrust	28 133	423	3 691

ATM Simulation is, as the name suggests, a simulation of an atm machine. It features all the expected functions of an atm, such as money withdrawal, money deposit and balance checking.

iTrust is a web based electronic health records system. Originally designed as a part of a software evolution course at NSCU [31], the project has been used in the domain of software traceability [32], requirements engineering [33] and security [34, 35].

3.2.4 Ground truth

A ground truth had to be set to validate the results of applying the tools to the included projects. The ground truth consisted of: a set of classes applicable to the concepts defined in the final list of constraints, annotations to the source code, and a record of each violation found.

In the absence of any previously established ground truth, in large because we defined the constraints, a structural analysis was performed for each of the systems. Our analysis was guided by the architectural descriptions supplied with the projects and the extraction of SecDFDs by Peldszus et al. [29]. Each constraint was applied to the system by analyzing every file in the project in a separate pass. Although time-consuming, performing a complete scan of the entire system for each constraint reduced the change of wrongfully dismissing a class and ensured that violations were detected.

In order to reduce the bias of the ground truth, the analysis was performed individually, later comparing the results. In cases were the classification differed, a careful review of the reasoning behind each author's choice was performed, resulting in a combined decision.

4

Selection of Architectural Security Constraints

This chapter describes the result of compiling a list of security constraints, as described in the methodology chapter, and the selection of seven constraints that have been implemented in SecArchUnit.

4.1 Compiled list of security constraints

The compiled collection of security constraints can be seen in Table 4.1. There are in total 18 constraints. As mentioned in Section 3.1.2, each constraint was categorized according to the goals of CIAA to ensure coverage of a diverse set of security goals.

Although both the architectural rules found in Jasser [5] and the security patterns presented in Scandariato et al. [26] were at the appropriate level of design, many of the weaknesses presented in CAWE were not. Common examples include; CAWE-259 "Use of hard-coded password" where the weakness is reliant on a local change of behavior rather than the architectural structure; and CAWE-263 "Password aging with long expiration" where the weakness is introduced by a single variable most likely defined outside of the source code. As a result, a far lower percentage of entries were included from the CAWE-catalog compared to the other sources.

4.2 Selection of constraints to be implemented in SecArchUnit

As explained in Section 1.4, the aim is not to demonstrate the enforceability of as many constraints as possible, but rather to investigate the feasibility of using the tool in this manner. To that end, a subset of the full list of security constraints is selected to be implemented in SecArchUnit. In total, seven constraints have been selected, as this allows us to cover at least one constraint from each goal in the CIAA model. The selected constraints can be seen in Table 4.2. The remainder of this section presents each selected constraint in further detail.

Table 4.1: Security constraints and their related CIAA goals.

ID	Constraint	Goal
1	Exceptions shown to the client must be sent to a sanitizer	Confidentiality
2	Sensitive information must not bleed to components with lower security classification	Confidentiality
3	Sensitive information must be encrypted before transmission	Confidentiality
4	Every outbound message must be sent from a single component responsible for transmissions	Confidentiality
5	Data that passes a trust boundary must first be sent to a component responsible for hiding or removing sensitive data	Confidentiality
6	Secrets must not be exposed in log messages	Confidentiality
7	The system must not provide functionality to decrypt secured log messages	Confidentiality
8	Output passing between components must be validated against its specification	Integrity
9	Input from a user must pass through a component validating the data	Integrity
10	The session object must not be accessible to the user	Integrity
11	Components must store its state as restorable checkpoints	Availability
12	Spawning of threads must be limited or throttled	Availability
13	The system must not have multiple points of access	Accountability
14	At least one checkpoint must be initialized after successful authentication and authorization	Accountability
15	Methods related to security events must call the logger	Accountability
16	Authentication and authorization must each be enforced in a single component	Accountability
17	Security relevant log messages must be encrypted and immutable	Accountability

Table 4.2: Constraints to be implemented in SecArchUnit.
Column $\#_{4.1}$ refers to the ID of the constraint in Table 4.1.

#	$\#_{4.1}$	Constraint
1	15	Methods related to security events must call the logger
2	16	Authentication and authorization must each be enforced in a single component
3	4	Every outbound message must be sent from a single component responsible for transmissions
4	9	Input from a user must pass through a component validating the data
5	12	Spawning of threads must be limited or throttled
6	2	Sensitive information must not bleed to components with lower security classification
7	6	Secrets must not be exposed in log messages

4.2.1 Log all security events

Description: In any system, several components either directly change or process data, which represents the system's asset, or indirectly by invoking other components to act on its behalf. In either case, the request to perform a particular action originates from an actor (user or external process) who should later be held accountable. As a consequence, the system should log a security event before performing an action that could breach the specified security policies. Although the term security event has become somewhat ambiguous, the definition used in the context of this report comes from the SANS Institute: "An event is an observable occurrence in an information system that actually happened at some point in time." [1]

Typical enforcement: The usage of the *audit interceptor* forces all requests from a user to first be sent to a component responsible for logging the request and later forwarding it to the intended target.

Sources: CAWE 223/778, Jasser rule 5, Security pattern *Audit interceptor*.

Attack scenario: A typical scenario where the logging of security events increases a system's resilience to attacks is that of failed login attempts. An attacker may try and guess the credentials of a user by employing a brute-force attack. During the attack, the attacker performs several failed attempts at guessing the credentials, (hopefully) causing the system to either increase the time between repeated attempts or lock the account entirely though with the added effect of decreased availability for the intended user. Although this type of defense temporarily hinders the attacker, a log of failed attempts facilities the detection of malicious actors and enables administrators to impose more permanent measures.

4.2.2 Enforce AuthN/AuthZ at single point

Description: Any system that has more than one user needs to incorporate functionality for authentication (AuthN), as well as authorization (AuthZ) if the privileges between users differ. The difficulty in complex systems where components handle different functionality, thus receiving separate requests and creating multiple entry points, is the fact that the components may have been designed to use various mechanisms of authentication. Instead, AuthN/AuthZ should be delegated to a single component to ensure consistent behavior across all entry points.

Typical enforcement: Designing a single component responsible for AuthN/AuthZ mechanisms across several points of entry. Several third-party libraries exist that provide such features as well as language extending specifications such as Jakarta EE (formerly J2EE).

Sources: CAWE 288/420/592, Security pattern *Authentication enforcer* and *Authorization enforcer*

Attack scenario: In system where the following conditions are true:

[1] `https://www.sans.org/reading-room/whitepapers/incident/events-incidents-646`

- There are multiple points of entry;
- There are different mechanisms to provide AuthN/AuthZ, some having a greater certainty that a user is properly authenticated or authorized to perform an action
- and all points of entry share the same session object

An attacker may try and gain access to the least trusted point of entry and later use the granted authority to access services or operation normally requiring a greater level of trust.

4.2.3 Messages are sent from a central point

Description: Communication with external actors, whether they are a client connecting to a server, or the system sending data to a third party, is commonly performed over insecure networks using several components. Encryption is the preferred method of securing such communication against potential attackers, whereas removing secrets from the data to be sent ensures that a user only sees non-sensitive information. Having a single component responsible for all outbound communication reduces the risk of information disclosure (e.g. transmitting a sensitive message via an insecure network or disclosing implementation details through stack traces), and can prevent harmful output from reaching the client (e.g. cross-site scripting attacks from other users).

Typical enforcement: Outbound messages can be intercepted before transmission to facilitate output sanitization. Similarly, outgoing messages can be forced to pass through a single component, designated as the sending point. This sending point can handle sanitization and decide whether the sender is allowed to carry the specified message.

Sources: Jasser rules 11 and 12.

Attack scenario: A blog website may properly use a delegated component for the sanitized transmission of some data (e.g. blog entries) but fail to do so for others (e.g. comments). An attacker who posts a comment containing HTML tags may then hijack the browser session of other users visiting the site.

4.2.4 Validate user input

Description: The ability to receive and process user input is fundamental to every computer system. However, the same input is also the primary source of untrusted data as an attacker possesses full control of what the system receives. Assuming that all data passed to a system is safe to process can have severe consequences when interpreting user input as a part of a query, often referred to as injection. In order to prevent an attacker from compromising the system by injection, all user input must be validated.

Typical enforcement: Placing a component performing validation between the user's input and the component processing the data ensures that the input can be trusted. The approach is commonly referred to as the security pattern *input guard*.

Sources: CAWE 20/59/74-79/88-91/93-99/138/150/349/352/472/473/502/601/ /641/643/652/790-797/942, Security pattern *input guard*.

Attack scenario: In an application that uses user input to build a SQL query to retrieve a specific account number (as seen in Listing 4.1) an attacker may construct the request to retrieve all accounts by adding characters that break the query and introduces new parameters, such as `' or '1'='1`. The resulting operation would retrieve all customer accounts, thus exposing sensitive information.

```
String query = "SELECT * FROM accounts WHERE
    custID='" + userInput + "'";
```

Listing 4.1: Example of a vulnerable SQL query.

4.2.5 Restrict thread spawning

Description: Computers have finite resources in terms of memory, CPU time and network bandwidth. Systems should be designed with this in mind, employing measures to avoid exhausting the computer's resources. This constraint limits the number of threads that can be spawned on behalf of actors, which could otherwise lead to a malicious actor occupying all of the available CPU time.

Typical enforcement: Tasks can be dispatched to a pool of worker threads that is not allowed to grow beyond a fixed size. Moreover, various mechanisms can be employed to throttle or limit requests such that a single actor cannot occupy all of the allotted threads.

Sources: CAWE 770.

Attack scenario: An attacker may initiate many requests that are each handled by the system in a separate thread. By initiating requests at a higher rate than the server is able to process them, the resources at the server are eventually exhausted. This leads to a denial of service for any legitimate actors attempting to access the system.

4.2.6 Sensitive information must stay within trust boundary

Description: Generally, a specific set of components, which have stricter security requirements constraining their implementation, handles the sensitive data within a system. Should that information leak to less secure components, the risk of exposing secrets to the user, and a potential attacker, increases significantly. In order to prevent leakage to less secure components, sensitive information must stay within a trust boundary.

Typical enforcement: A typical approach is to manually review methods that receives or send information to other components and ensure that they do not expose any secrets. As for automated enforcement, various information flow analysis tools, like JOANA[2], can be employed to detect these types of information leaks within a system.

Sources: CAWE 488.

Attack scenario: An asset may be leaked from a component that is supposed to service a single actor, such as a session object, to a component that multiple actors have shared access to. This may subsequently lead to the asset being illegitimately accessed by a malicious actor.

4.2.7 Secrets must not be exposed in log messages

Description: Many systems handle secrets that should never touch permanent storage. A password is perhaps the most common example of such a secret. While great care can be taken on the design level to ensure that these secrets are not stored to disk, they may still be exposed unintentionally through log messages. In order to prevent such exposure, messages that are sent to the logger must not contain secrets.

Typical enforcement: Similar to the constraint described in Section 4.2.6, the typical approach is to manually review calls to the logger to ensure that no secrets are exposed, with the potential ability to use information flow analysis tools.

Sources: CAWE 359/532, Jasser rule 13.

Attack scenario: Log messages may be accessible to actors who are not otherwise granted direct access to the secrets. By exploiting an unintentional leak of secrets into the log messages, an attacker could systematically extract these without facing the intended restrictions.

[2] `https://pp.ipd.kit.edu/projects/joana/`

5
Enforcing Constraints

This chapter explains how the constraints are expressed and validated in the tool using the definitions of the rules as well as by applying the rules to a modeled example system. The constraints are divided into three distinct categories. The first category contains the constraints that are possible to express in ArchUnit as-is. The second category describes constraints that are enforceable with the help of additional information in source code. The third and final category details constraints that require an extension of ArchUnit to be possible to enforce.

5.1 Running example

We modeled a toy system, BlogWeb, and use it as a running example for this chapter. By no means is this example meant to be a realistic application, but is instead used to showcase our constraints. A UML class diagram of BlogWeb is illustrated in Figure 5.1.

BlogWeb is, as the name suggests, a blog website providing users with the ability to publish statuses to their blogs. Being a website means that the system receives actions to execute via HTTP requests. These requests are received by the RequestHandler and are forwarded to the UserService, which provides public methods for sign in and publishing a status update. The UserService depends on UserAuthorizer and UserAuthenticator which implement the logic for authorization and

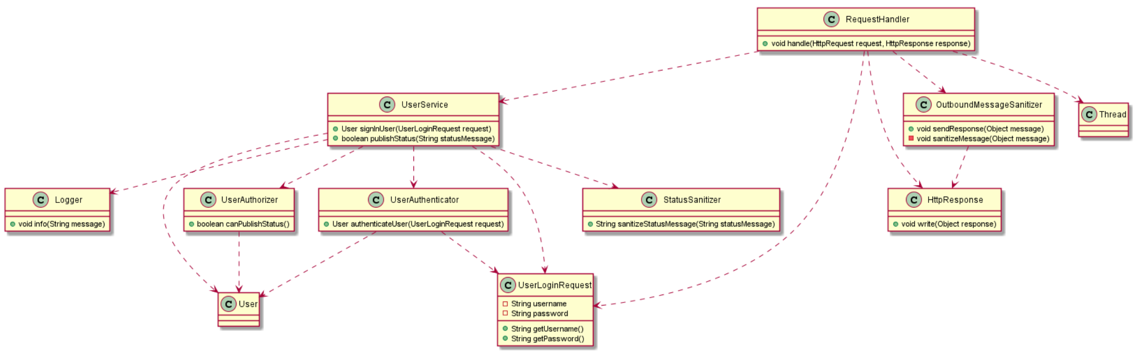

Figure 5.1: A UML class diagram of the toy system BlogWeb. Zoom in to read the labels, or see full-scale version in Appendix A.

authentication. The UserService also uses the Logger to perform the necessary logging of user requests. StatusSanitizer is used to sanitize the status updates received by users before they are stored while OutBoundMessageSanitizer is responsible for sanitizing responses. Finally, as the RequestHandler may serve several users at once, a thread is spawned for each request to make the execution concurrent.

5.2 Support in ArchUnit as-is

ArchUnit contains an extensive vocabulary for expressing architectural constraints in a sentence-like manner, as shown through an example in Listing 5.1. This rule is defined entirely using the standard vocabulary and enforces complete mediation, a security pattern which dictates that some internal classes must only be accessed through a mediator.

```
1 ArchRule rule = classes()
2     .that().resideInAPackage("..internal..")
3     .should().onlyBeAccessed().byAnyPackage("..mediator..");
```

Listing 5.1: Example of a rule that is expressed with the standard vocabulary.

Rules in ArchUnit are composed of three parts. The first part (line 1 in Listing 5.1) is the type of Java constructs that should be inspected; namely classes, fields or methods. The second part (line 2) is a predicate that is used to filter for constructs related to the constraint. The third part (line 3) describes the logical condition that must hold true for all of the selected constructs. In short, rules are generally expressed as sentences of the form “The **constructs** that match **predicate** should fulfill **condition**.”

In situations where the standard vocabulary is insufficient for expressing a constraint, there is a possibility to define custom predicates and conditions over any given construct. These can be supplied as arguments to the `that()` and `should()` methods. Custom predicates and conditions are used extensively in our implementation, as will be made apparent in the following sections.

5.2.1 Constraint 1: Log all security events

The definition of the architectural rule related to constraint 1 can be seen in Listing 5.2. This constraint is expressed with the assumption that there are services, in the form of classes, which are responsible for performing security related events. Every publicly accessible method in such a service is assumed to perform a security event and must therefore contain a call to the logging facility for accountability purposes.

```
ArchRule logSecurityEvents(
        DescribedPredicate<? super JavaClass> securityServicesDescriptor,
        Class<?> logger) {
    return methods()
        .that().haveModifier(JavaModifier.PUBLIC)
        .and().areDeclaredInClassesThat(securityServicesDescriptor)
        .should(callMethod(declaredIn(logger)));
}
```

Listing 5.2: Rule definition for constraint 1.

The predicate that selects the security services, and the class that is responsible for logging, are passed as arguments to the architectural rule. Hence, there is no need for adding information to the source code of the target system. Furthermore, by using a predicate to select the security services, the developer is left with some flexibility in how they decide to apply the constraint to a system. As opposed to a plain list of classes, a predicate can match all classes belonging to a specific package or following a set naming scheme, minimizing the need for revisiting the constraint as the system evolves. The logger, on the other hand, is described as a single class, meaning that all events must be logged with the same global logger.

In BlogWeb, illustrated in Figure 5.1, the logging facility is the class named `Logger` while the only security service is the `UserService` class. An application of the constraint on this system can be as simple as the one shown in Listing 5.3.

```
@ArchTest
ArchRule logSecurityEvents = SecArchUnit
    .logSecurityEvents(type(UserService.class), Logger.class);
```

Listing 5.3: Application of constraint 1 to BlogWeb.

5.2.2 Constraint 2: Enforce AuthN/AuthZ at single point

The definition of the second constraint is detailed in Listing 5.4. This constraint is defined in terms of two concepts: an authentication point and an authentication enforcer. Authentication is performed through a method call to the authentication enforcer, which is a class whose sole responsibility is to authenticate an actor. This call should only occur at the authentication point for the sake of ensuring a uniform authentication mechanism throughout the system. Authorization is enforced in the same manner, with the concepts of an authorization point and an authorization enforcer.

```
ArchRule enforceAuthenticationAtCentralPoint(
        Class<?> authenticationPoint,
        Class<?> authenticator) {
    return CompositeArchRule.of(
        theClass(authenticationPoint)
            .should(callMethod(declaredIn(authenticator)))
    ).and(
        methods()
            .that().areDeclaredIn(authenticator)
            .should(onlyBeAccessedBy(authenticationPoint))
    );
}

ArchRule enforceAuthorizationAtCentralPoint(
        Class<?> authorizationPoint,
        Class<?> authorizer) {
    return enforceAuthenticationAtCentralPoint(
        authorizationPoint,
        authorizer
    );
}
```

Listing 5.4: Rule definitions for constraint 2.

The constraint is defined as two separate rules, one for authentication and one for authorization. This is purely for the sake of clarity, as their implementations are identical.

In BlogWeb, the authentication and authorization points are both situated in the `UserService` class while authentication and authorization are enforced by the classes `UserAuthenticator` and `UserAuthorizer` respectively. The application of the rule can be seen in Listing 5.5.

```
@ArchTest
ArchRule enforceAuthentication = SecArchUnit
    .enforceAuthenticationAtCentralPoint(UserService.class,
       UserAuthenticator.class);

@ArchTest
ArchRule enforceAuthorization = SecArchUnit
    .enforceAuthorizationAtCentralPoint(UserService.class,
       UserAuthorizer.class);
```

Listing 5.5: Application of constraint 2 to BlogWeb.

5.2.3 Constraint 3: Messages are sent from a central point

The rule definition of the third constraint can be seen in Listing 5.6. This constraint dictates that all outbound messages are sent from a central sending point. The intent is to have a single point that handles output sanitization or performs other safety checks on messages before they are sent.

```
ArchRule sendOutboundMessagesFromCentralPoint(
        Class<?> sendingPoint,
        DescribedPredicate<? super JavaClass> senderDescriptor) {
    return methods()
        .that().areDeclaredInClassesThat(senderDescriptor)
        .and(haveAtLeastOneParameter)
        .should(onlyBeAccessedBy(sendingPoint));
}
```

Listing 5.6: Rule definition for constraint 3.

The act of sending a message is defined as a method call to a sender with at least one argument, which is assumed to contain the message contents. The reasoning behind this distinction, and why not all accesses are constrained, is that any class should be allowed to create and pass around a sender instance without violating the constraint. Since there can be multiple sender classes in a system, e.g. one for HTTP requests and one for SMTP messages, the rule accepts a predicate that can select all these sender classes.

Listing 5.7 showcases how the constraint can be applied to BlogWeb. In this system, there is a single sender class `HttpResponse`, responsible for returning a response to a client. The central sending point is the `OutboundMessageSanitizer` class.

```
@ArchTest
ArchRule centralSendingPoint = SecArchUnit
    .sendOutboundMessagesFromCentralPoint(
        OutboundMessageSanitizer.class,
        type(HttpResponse.class)
    );
```

Listing 5.7: Application of constraint 3 to BlogWeb.

5.3 Adding information to source code

Some of our constraints require that the developer injects additional information into the source code. In some cases, this information is simply an indicator that says something about an entire class. Naming the class with a specific prefix or suffix is one approach to accomplish this. Another approach is to implement an empty interface, which is the technique used with Java's `Serializable`[1] interface.

In other cases, however, the information may be related to specific fields or methods of arbitrary signatures. For the purposes of flexibility and minimizing the obtrusiveness of our approach, any extra information is expressed in the form of annotations. These can be applied to classes, fields, methods and parameters without changing the underlying architecture of the system.

The need for additional information within the source code becomes apparent in the case where a class contains public methods with varying degrees of security

[1]https://docs.oracle.com/javase/7/docs/api/java/io/Serializable.html

requirements. A typical example is found in constraint 4, where a class is responsible for handling user input. Some methods might receive predefined values, such as in the case of checkboxes, whereas others take strings entered by the user. The former type of input is guaranteed to be safe, whereas the latter is entirely under the control of the user, thus introducing the need for sanitation. Using the broader predicate of entire classes (described in Section 5.2) would not allow the constraint to be limited to specific methods within a class (those receiving potentially malicious input), and later trace a call to a method providing sanitation. Thus, annotations provide the granularity needed to limit the scope of a constraint to the applicable code units only.

5.3.1 Constraint 4: Validate user input

The rule definition of constraint 4 can be seen in Listing 5.8. User input comes in many forms, and therefore, it is impossible to define a single algorithm to properly validate every single type. The problem grows further as queries (such as SQL) or other types of processed data (such as XML), each with its own set of grammar, are often formed using strings. As a consequence, the implemented constraint is more abstract as it checks whether a class that receives user input is said to either perform validation on its own or delegate the task another method. In total, three distinct cases conforming to the constraint were considered:

- *Method A* is annotated with both **UserInput** and **InputValidator**.
- *Method A* is annotated with **UserInput** and calls a *method B* that is annotated with **InputValidator**.
- *Method A* is annotated with **UserInput** and is only called by methods that are annotated with **InputValidator**.

```
1 ArchRule validateUserInput() {
2     return codeUnits()
3         .that().areAnnotatedWith(UserInput.class)
4         .should(performDirectOrIndirectValidation);
5 }
```

Listing 5.8: Rule definition for constraint 4.

As shown in Listing 5.8, the **UserInput** annotation is used on line 3 to limit the set of applicable code units, whereas the custom condition `performDirectOrIndirectValidation` implements the logic to ensure that at least one of the three cases outlined above is satisfied.

In BlogWeb, a user publishes a status update in the form of a string that the UserService class receives. The status is then passed to the StatusSanitizer class, where it is validated. The affected methods of each class are marked with the appropriate annotation, as shown in Figure 5.2. If the two annotations were to be added, without a call from the relevant method in the `UserService` class to the `StatusSanitizer`, the rule would fail and mark it as a violation of the constraint.

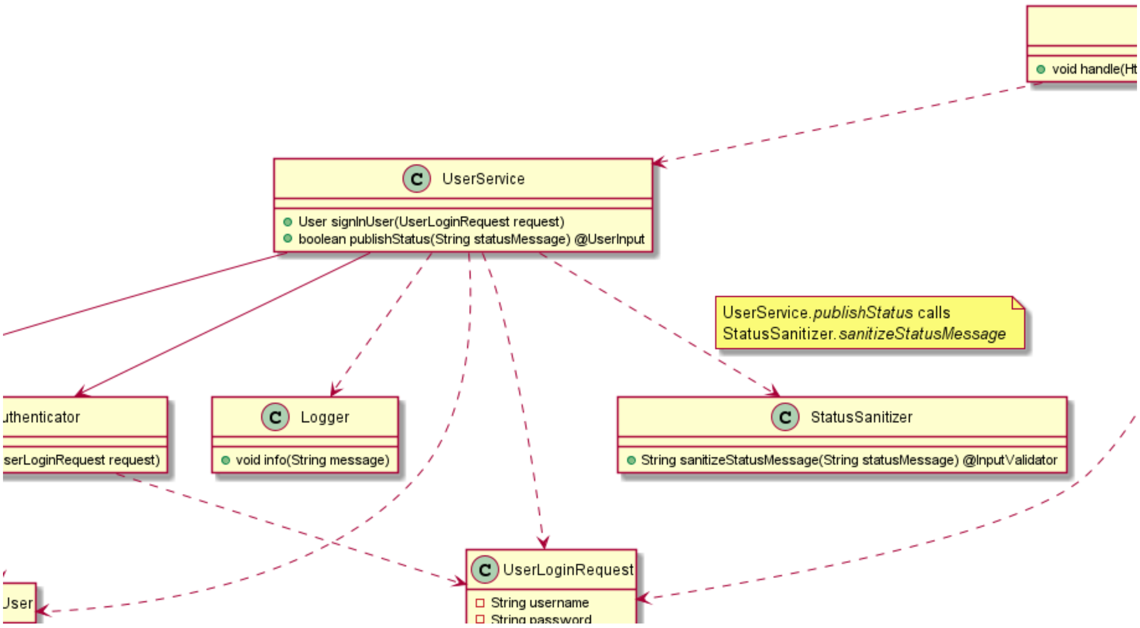

Figure 5.2: Applying constraint 4 to the model of BlogWeb using added annotations on the UserService and StatusSanitizer class.

5.3.2 Constraint 5: Restrict thread spawning

The implementation of constraint 5 can be seen in Listing 5.9. While there are many ways computer resources can be exhausted, this constraint focuses on preventing the exhaustion of CPU time and memory specifically through the creation of new threads and processes. As such, every block of code that contains a call to the `start()` method of a `Thread`[2] or any of its subclasses, must be marked as containing a resource restriction mechanism. The same rule is applied for calls to `ProcessBuilder.start()`[3] and `Runtime.exec()`[3], which lead to the creation of new processes.

```
ArchRule limitResourceAllocation() {
    return noClasses()
        .that().areNotAnnotatedWith(ResourceRestriction.class)
        .should().callMethodWhere(
            aThreadIsStartedWithoutRestriction
        ).orShould().callMethodWhere(
            aProcessIsStartedWithoutRestriction
        );
}
```

Listing 5.9: Rule definition for constraint 5.

The marking is done with the help of a **ResourceRestriction** annotation, either on the relevant method or the entire class. The decision of how the restriction mechanism is implemented is left to the developer of the system.

[2]https://docs.oracle.com/javase/7/docs/api/java/lang/Thread.html
[3]https://docs.oracle.com/javase/7/docs/api/java/lang/Process.html

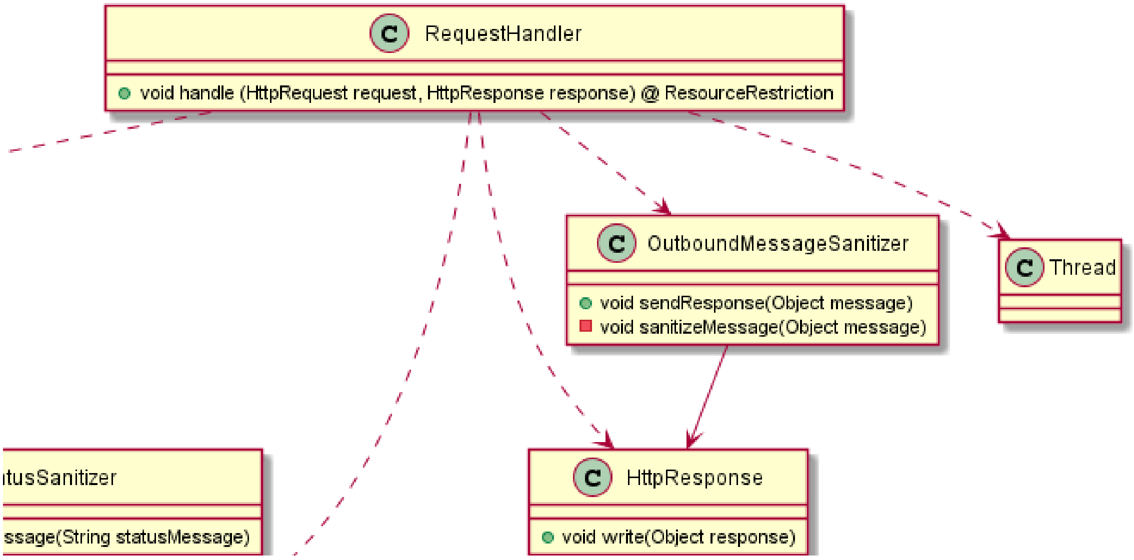

Figure 5.3: Applying constraint 5 to the model of BlogWeb using the added annotation on the RequestHandler class.

In BlogWeb, the `RequestHandler` class spawns new threads in order to handle several user requests concurrently. For the system to conform to the constraint, the **ResourceRestriction** annotation either has to be added to the entire class, or to the specific method that spawns the thread as shown in Figure 5.3. If not, the rule will report a violation, and the developer will have to ensure that the class cannot excessively spawn new threads.

5.4 Extending ArchUnit analysis

In the current ArchUnit API, a rule that aims to constrain method calls can only be defined in terms of the signatures of the method and its parameters. This is not an issue when the arguments passed to a method are of the same type as the parameters. However, the argument might also be a descendant of the parameter type. There is currently no support in ArchUnit to constrain the types of the objects that are actually being passed as arguments.

Consider constraint 7, which aims to ensure that no secrets are passed to the logger. Say there is a `Secret` annotation that marks all the classes whose instances must not be passed to the logger. An attempt can be made to enforce the constraint with the current ArchUnit API using a custom predicate, as seen in Listing 5.10. Given that a typical logger will accept either a plain string, or a format string along with an array of objects to be formatted, this architectural rule does little in the way of preventing secrets from being passed to such a logger.

```
ArchRule doNotLogSecrets(Class<?> logger) {
    return noClasses()
        .should().callMethodWhere(
            parameterTypeAnnotatedWith(Secret.class)
                .and(targetOwner(type(logger)))
        );
}
```

Listing 5.10: A first attempt to implement constraint 7.

For the final 2 constraints, there is a need for an extension that allows constraints to be defined against method arguments rather than parameters. There should also be hints about where these arguments have been derived from, e.g. which types that make up the components of a concatenated string. The following sections describe the extensions that have been made to ArchUnit, both in regards to its analysis and the information represented in its domain, as well as how these extensions are utilized in the definitions of the final constraints.

5.4.1 Implemented extensions

Background. ArchUnit builds its representation of the architecture using ASM[4], a Java bytecode analysis framework. ASM reads bytecode and generates callbacks to methods in its various visitor classes. The visitor of most interest to us is `MethodVisitor`[5], which is responsible for processing the contents of a method, constructor or static initializer. These are collectively named *code units* in ArchUnit's domain, i.e. anything that may contain code.

ArchUnit extends the `MethodVisitor` class in `MethodProcessor`, which primarily visits instructions related to field accesses and method invocations (see Figure 5.4). These instructions are processed into information about accesses between Java members, not entirely unlike the information provided by a static call graph.

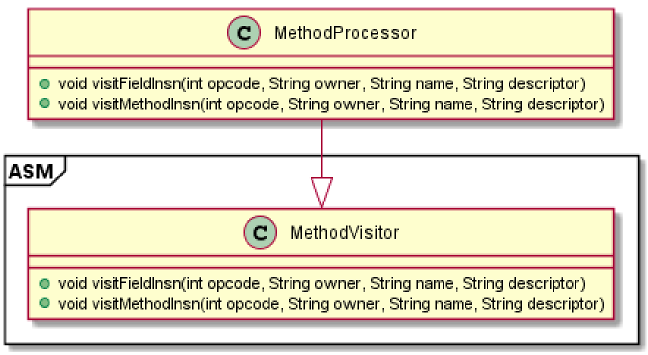

Figure 5.4: ArchUnit: The immediate context of the `MethodProcessor` class, responsible for analyzing code units.

[4] `https://asm.ow2.io/`
[5] `https://asm.ow2.io/javadoc/org/objectweb/asm/MethodVisitor.html`

Java, as a stack-oriented programming language, passes arguments to method calls and field assignments via the operand stack [16]. As such, an inspection of the operand stack at the time of a method call yields information about the arguments being passed. Conveniently, ASM provides an extension of its `MethodVisitor` class, called `AnalyzerAdapter`, which is capable of simulating the effect that each instruction has on the operand stack.

SecArchUnit. As seen in Figure 5.5, `MethodProcessor` now extends `Analyzer-Adapter` and utilizes the information about the operand stack to perform an analysis of information flow within a code unit.

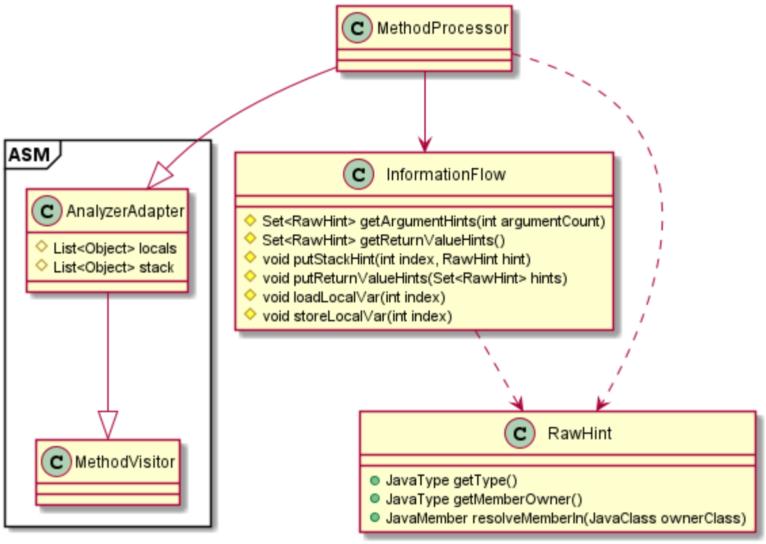

Figure 5.5: SecArchUnit: Changes made to the analysis of code units.

The information flow analysis of a code unit can be boiled down to the following key points:

- Loading a field onto the stack yields a hint in that stack position about its originating member, i.e. the field.
- Invoking a method that has a return value yields a hint in the stack position of the resulting object about its originating member, i.e. the invoked method.
- Invoking a static method transfers the hints about the arguments, if any, into the stack position of the resulting object, if the method has a return value.
- Invoking a non-static method additionally transfers the hints about the arguments into the instance the method was invoked on, and the hints about the instance into the resulting object.
- Storing an object in an array transfers the hints about the object into the array.
- Loading an object from an array transfers the hints about the array into the

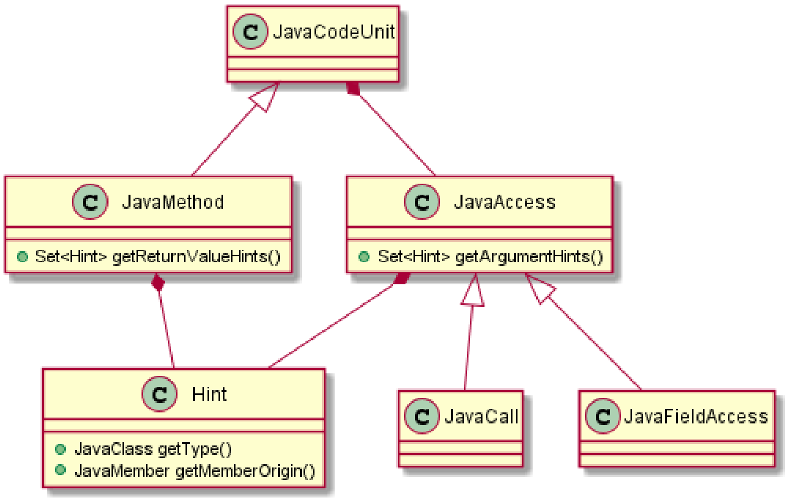

Figure 5.6: SecArchUnit: Changes made to the domain surrounding code units.

object.

- Storing or loading a local variable transfers the hints from the stack to the local variable and vice versa.
- Duplicating a reference also duplicates the collection holding the hints for that reference, such that hints that flow into the duplicate also flow into the original reference.

Once the analysis of all classes has been completed and SecArchUnit has built its representation of the architecture, the raw hints are resolved into hints referencing the actual type (`JavaClass`) and its origin (`JavaMember`), if any. As seen in Figure 5.6, `JavaAccess` now contains hints about the references that flow into the arguments of a method call or field assignment. Additionally, `JavaMethod` contains hints about the references that flow into the return value of the method. These changes to the domain aim to facilitate the definition of rules that constrain information flow.

5.4.2 Constraint 6: Sensitive information must stay within trust boundary

The rule definition for constraint 6 is presented in Listing 5.11. This constraint aims to restrict how assets that consist of sensitive information are allowed to flow between classes. The constraint deals with the two concepts of assets and asset handlers, which are expressed in the form of `Asset` and `AssetHandler` annotations in the source code. An asset is a sensitive member that should only flow to classes marked with a high security level, i.e. an asset handler.

```
ArchRule doNotBleedAssetsBetweenComponents() {
    return fields()
        .that().areAnnotatedWith(Asset.class)
        .should(notBleedToInsecureComponents);
```

Listing 5.11: Rule definition for constraint 6.

The rule itself requires no arguments, as all the necessary information is injected into the source code in the form of annotations. The custom condition, `notBleedToInsecureComponents`, ensures that the asset is only accessed directly by asset handlers. Moreover, it utilizes the information about information flow to ensure that the asset is not accessed indirectly through calls to methods where the asset flows into the return value.

The concept of asset handlers is treated as a single group of classes that have to a high security level, and are therefore granted access to all assets. No distinction is made between different assets and asset handlers in the current implementation.

In BlogWeb, the `password` field is considered an asset that must stay within the trust boundary of `UserLoginRequest` and `UserAuthenticator`. The appropriate annotations in this scenario can be seen in Figure 5.7.

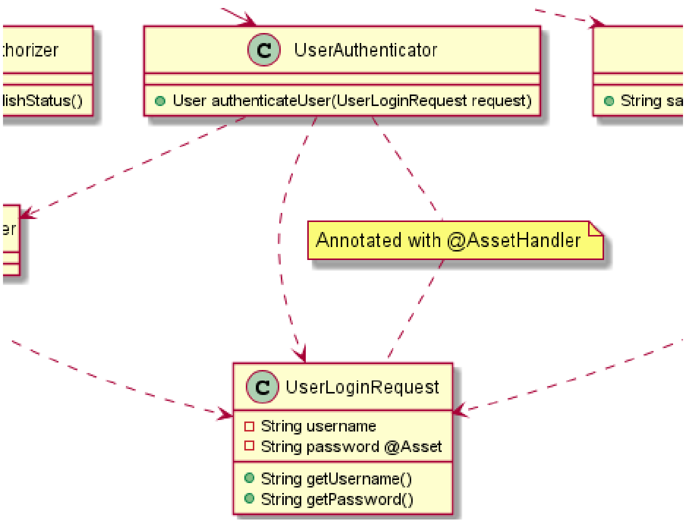

Figure 5.7: Application of constraint 6 to BlogWeb.

5.4.3 Constraint 7: Secrets must not be exposed in log messages

The rule related to constraint 7, which dictates that secrets must not be exposed in logs, is defined in Listing 5.12. There are a multitude of ways that a secret field can be exposed in a log message, e.g. through string concatenation, string formatting, or wrapping it in a different object and converting that object to its string representation. What these exposures have in common is that they issue a method call to the logger where the secret has flowed into the method arguments.

Figure 5.8: Annotation added to BlogWeb for constraint 7.

```
ArchRule doNotLogSecrets(
        DescribedPredicate<? super JavaClass> loggerDescriptor) {
    return noClasses()
        .should(passSecretArgumentTo(
            targetOwner(loggerDescriptor)
        ));
```

Listing 5.12: Rule definition for constraint 7.

Rather than a single class representing the preferred logger, as in constraint 1, this constraint should prevent exposures through all loggers present in the system. Therefore, the rule accepts a predicate that should select all such logging facilities. The custom condition `passSecretArgumentTo` inspects the argument hints of all outgoing method calls for any members marked with the `Secret` annotation. Additionally, for each originating member found in the hints, it recursively checks the hints that have flowed into that member, in an attempt to constrain information flow with intermediate steps in different code units.

In BlogWeb, the constraint can be applied as seen in Listing 5.13. The only secret in this system is the password field, annotated in Figure 5.8.

```
@ArchTest
ArchRule doNotLogSecrets = SecArchUnit
    .doNotLogSecrets(type(Logger.class));
```

Listing 5.13: Application of constraint 7 to BlogWeb.

6

Evaluation

This chapter presents an evaluation of SecArchUnit in terms of performance and usability when compared to static analysis tools used in industry. The results are discussed and related to the research questions. Finally, a discussion is held regarding the validity threats of the evaluation.

fine tune

6.1 Results

This section presents the results of the evaluation in two steps: first the evaluation relating to how effectively SecArchUnit and similar tools validate the constraints, and then an evaluation of differences between the tools in terms of their usability.

6.1.1 Performance

The performance evaluation aims to evaluate how well SecArchUnit can validate the 7 constraints and compare this to the performance of industrial-grade tools SonarQube and PMD. Due to the fact that not all constraints could be expressed in the tools used for comparison, the evaluation was divided into two stages: a comparison of all tools using the first 5 constraints, and a review of the performance of SecArchUnit regarding constraints 6 and 7.

For both stages, the tools were evaluated according to the performance metrics. The true positives (TP) refer to violations of constraints that are reported by the tool and coincide with the ground truth. False positives (FP) are violations reported by the tool that are not included in the ground truth. False negatives (FN) are violations that exist in the ground truth but do not get reported by the tool.

As seen from Table 6.1, all three tools performed well in regards to both precision and recall. Since each tool's rules were evaluated against the same collection of classes, the set of detected violations is the same for all tools. Overall, very few false positives and false negatives were found, making up for less than 5% of the total amount of reported violations when counting unique cases across all tools and systems.

As depicted in Figure 6.1, constraint 1 accounts for the majority of all violations found in the three projects. Violations of constraint 5, on the other hand, were

Table 6.1: Results from validating constraints 1-5 using SecArchUnit, SonarQube and PMD.

Project	Tool	TP	FP	FN	Precision	Recall
ATMSim	SecArchUnit	19	1	0	0.95	1
	SonarQube Plugin	19	1	0	0.95	1
	PMD Plugin	15	1	4	0.94	0.79
JPetStore	SecArchUnit	51	0	0	1	1
	SonarQube Plugin	51	0	0	1	1
	PMD Plugin	47	0	4	1	0.92
ITrust	SecArchUnit	216	3	0	0.99	1
	SonarQube Plugin	216	0	0	1	1
	PMD Plugin	216	0	0	1	1

not found in any project; thus, we can not draw any conclusions regarding the tools' reliability of detecting that particular violation. Additionally, violations of constraints 2 and 3 were sparse and not consequently found in all systems.

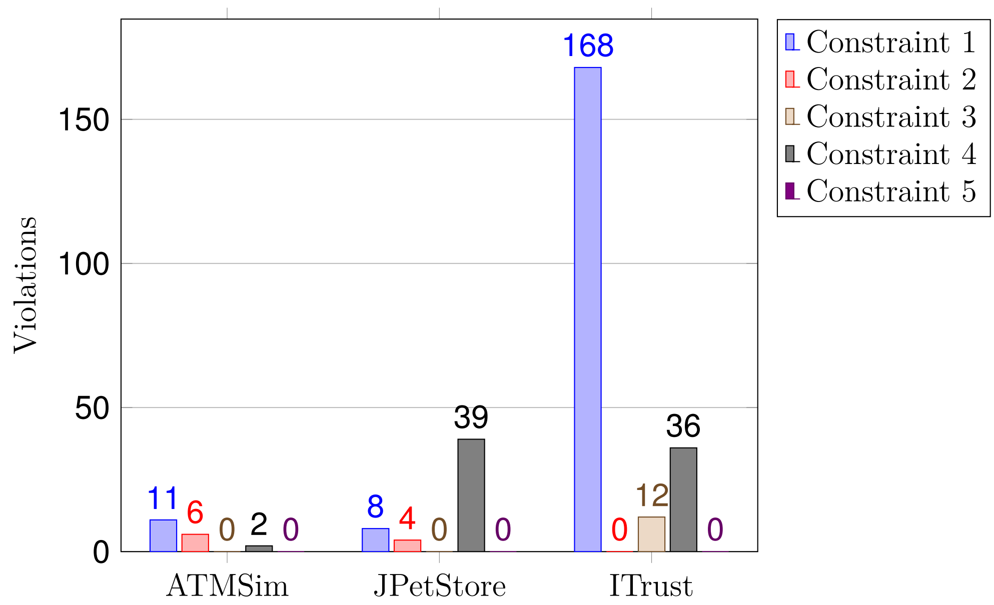

Figure 6.1: The constraint violations found in the ground truth of each system.

In the second stage, constraints 6-7 were applied to iTrust and validated solely using SecArchUnit. The performance metrics resulting from this evaluation are presented in Table 6.2. Similarly to the results of the first stage, SecArchUnit reliably detects violations of both constraints without reporting any false negatives, resulting in a high recall. Additionally, no false positives were found, leading to a high degree of precision.

Looking at the distribution of constraint violations for constraint 6 and 7 in iTrust, found in Figure 6.2, we can see that both constraints had enough violation to allow for an evaluation.

Table 6.2: Results from validating constraints 6-7 on iTrust using SecArchUnit.

Project	Tool	TP	FP	FN	Precision	Recall
iTrust	SecArchUnit	61	0	0	1	1

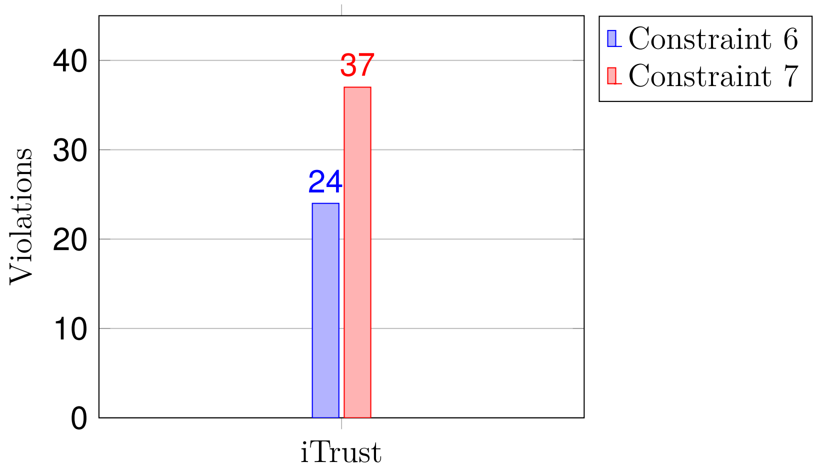

Figure 6.2: The violations of constraint 6 and 7 found in iTrust when asserting ground truth.

Finally, the performance of SecArchUnit for each constraint across all systems can be seen in Table 6.3. We can infer from data that SecArchUnit reliably detects violations of all constraints except for constraint 5 as it was not present in any of the systems.

Table 6.3: The performance of SecArchUnit for each constraint across all test systems.

Constraint	TP	FP	FN	Precision	Recall
1	167	3	0	0.98	1
2	10	0	0	1	1
3	12	1	0	0.92	1
4	87	0	0	1	1
5	0	0	0	-	-
6	24	0	0	1	1
7	37	0	0	1	1
Mean				0.98	1

6.1.2 Usability

The tools were additionally evaluated in terms of their usability. Firstly, the effort required to express the constraints in the different tools was considered. This effort was quantified as the amount of time spent implementing the rule pertaining to each constraint. These results can be seen in Figure 6.3. From the figure, and the fact that the constraints were implemented sequentially, we can infer that the time required to implement new constraints decreases following increased knowledge of

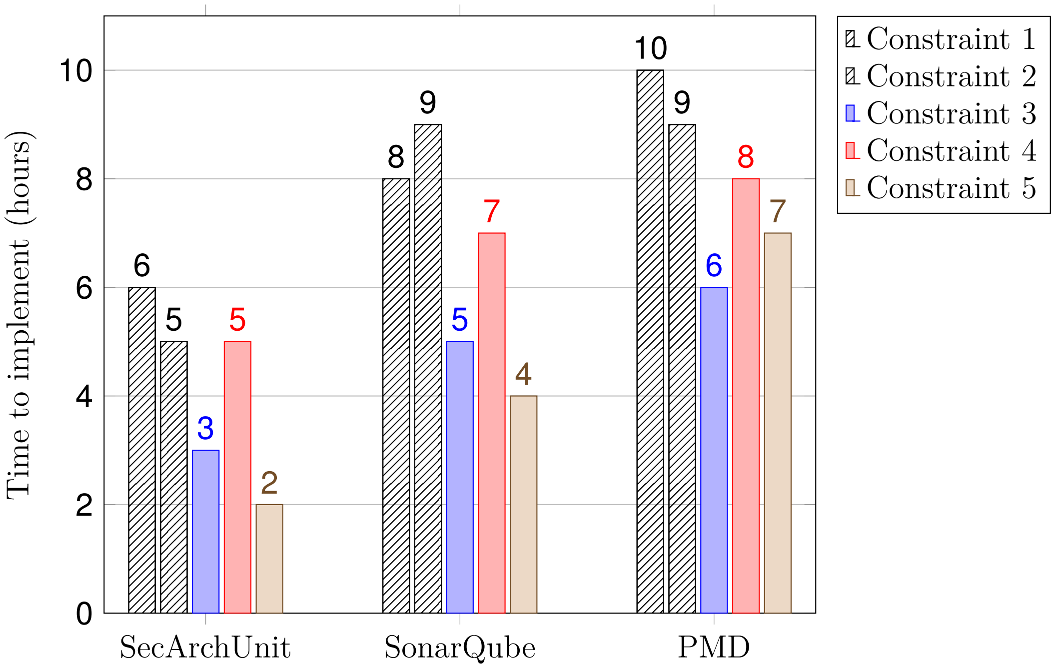

Figure 6.3: Time to implement each constraint.

the tools. As constraint 4 included additional functionality, in the form of inspecting annotations, an increase in time can be seen between constraint 3 and 4. For both SecArchUnit and SonarQube, the time to implement is significantly reduced when later implementing constraint 5. PMD, on the other hand, does not see the same decrease. The absence of decreased time was attributed to the fact that there was additional work involved with extracting and checking annotations in PMD, which we distributed equally between the two constraints.

Secondly, the time required to validate the constraints, i.e. the running time of each tool, was considered. The tools were executed three times on each system to improve the accuracy of the measurement. These results are presented in Table 6.4. SecArchUnit and PMD perform similarly on the two smaller systems, with both finishing in approximately 1 second. In contrast, the time required for SonarQube to finish is increased by a factor of ten. When validating iTrust, there is an apparent disparity between SecArchUnit, which completes its validation in 2 seconds, whereas PMD and SonarQube complete their validation in 7 and 39 seconds, respectively.

6.2 Discussion of performance results

Both in regards to precision, as well as recall, the tools performed equally. However, the causes of failure differed noticeably in cases where the results of the tools varied. The examples are described below:

- In ATM simulator, the same false positive occurred in all three tools. This was in relation to constraint 3, where a subclass of the sending point contained a method call to the sender. Additionally, PMD had 4 false negatives which

Table 6.4: Run time benchmarks from validating the constraints in each tool and system, measured in 3 consecutive runs.

Project	Tool	Mean time (s)	sd
ATMSim	SecArchUnit	0.957	0.072
	PMD Plugin	1.3	0.078
	SonarQube Plugin	13.7	1.7
JPetStore	SecArchUnit	0.837	0.035
	PMD Plugin	1.46	0.21
	SonarQube Plugin	12.4	0.50
ITrust	SecArchUnit	2.24	0.12
	PMD Plugin	7.37	0.23
	SonarQube Plugin	39.4	3.25

occurred because it was unable to determine the classes that these method calls targeted.

- In JPetStore, PMD reported 4 false negatives, again because it was unable to determine the target class of these method calls.

- In iTrust, a security service contained both an inner interface and an inner static class whose methods did not perform any security events. In both PMD and SonarQube, these methods were determined to be declared in the inner class by traversing the AST from the method to its first parent class or interface declaration. In comparison, SecArchUnit considers the members of the inner class to be declared in both the inner and outer class. This improperly marks the inner methods as violating the constraint, resulting in 3 false positives in SecArchUnit.

As shown in Figure 6.1, the tools were evaluated using imbalanced data. Constraint 1 accounted for a majority of all violations found throughout all three systems, while no system violated constraint 5. Additionally, iTrust was the only system to contain a violation of constraint 3.

To answer the research question regarding which architectural security constraints can be validated using ArchUnit (**RQ1**), the evaluation has shown that ArchUnit can validate constraints 1-4, which relate to the security goals of confidentiality, integrity and accountability. Its performance, in terms of precision and recall, is comparable to tools used in the industry. As the evaluation found no violation of constraint 5, it remains unclear whether the tool can validate constraints related to the security goal of availability.

transition to RQ2

The system, iTrust, initially contained no violations of constraint 7. Therefore, violations were injected by systematically marking all identifier fields (e.g. `patientId`, `personnelId`) in the model and base-action packages as secrets. We chose to mark these identifiers because they were commonly sent to the logger as a way to describe the patient or personnel involved in a transaction. Hence, the 37 violations of constraint 7, as seen in Table 6.2, are artificially injected.

Moreover, iTrust is built with a mix of Java and Java Server Pages (JSP) files whereas ArchUnit can only analyze plain Java. The classes in the action package, from which the logger is called, are all instantiated in the JSP files outside the view of our analysis. As such, the types of information flow that are analyzed and included in the ground truth are rather rudimentary. Out of the 37 violations of constraint 7, 1 was found without recursion (direct access to secret field) and the remaining 36 were found using a single recursion step (access to getter method of a secret field).

To answer the research question regarding what modifications can be made to ArchUnit (**RQ2**), the evaluation suggests that the tool can be extended with an analysis of information flow, and that this can be leveraged to validate additional constraints relating to the security goal of confidentiality.

Discuss examples of things that the tools won't catch

6.3 Discussion of qualitative differences

While SecArchUnit, SonarQube and PMD are all static analysis tools that support evaluation of custom rules, they differ considerably in how their rules are defined and evaluated.

SecArchUnit builds a representation of the entirety of the analyzed system, which is available during the evaluation of the rules. As shown throughout Chapter 5, rules have access to information about both incoming and outgoing accesses to all members, within and between classes, making it convenient to specify architectural constraints at an appropriate level of abstraction.

Regarding SonarQube and PMD, both of these tools evaluate rules by traversing an Abstract Syntax Tree (AST) where the root node is the Java class currently being analyzed. As a consequence, a rule can only inspect one class at a time; it can audit outgoing accesses from the current class, but it does not know anything about incoming accesses from other classes. In a constraint where incoming accesses to a specific class need to be restricted, these tools instead inspect all classes one by one and look for outgoing accesses to the concerned class, as exemplified in Listing 6.1.

```
public void visitNode(MethodInvocation node) {
    if (target method is declared in a sender) {
        // Get class in which method invocation takes place
        ClassTree classTree = node.parent() until type is ClassTree

        if (classTree is not the sending point) {
            reportIssue(...);
        }
    }
}
```

Listing 6.1: Excerpt of constraint 3 in SonarQube, simplified using pseudo code. See full rule definition in Appendix B and the related PMD rule in Appendix C.

As a consequence, these tools cannot enforce constraints where both incoming and outgoing accesses need to be inspected in unison. An example of this is constraint 4, where the constraint allows user input to be validated either by calling a validator or by only being called by validators. The rules in SonarQube and PMD can only look for outgoing calls to validators, not taking incoming accesses into account, and therefore risk reporting false positives. However, no such case was found in the evaluation.

When inspecting a method invocation node in the AST, SonarQube exposes information about the target method, including its signature, annotations, and information about the class in which it is declared. In PMD, however, the node that corresponds to method invocations only contains information about the return type of the method. In order to resolve the target class of a method invocation, we examined the immediately preceding expression. If the expression had a return type, this was assumed to be the target owner. Otherwise, the expression was further examined for references to local variables and fields, the declarations of which contained the necessary type information. This lack of type information not only made the constraints complicated to express, but it also resulted in an unreliable resolution of method targets, as reflected in the number of false negatives reported by the tool.

Moreover, PMD contains no information about the annotations that are present on the target method. As a workaround, a separate rule was created which visited all class and method declarations, dumping their annotations into a text file. This text file was then fed into the rules related to constraints 4 and 5.

To summarize, while it is possible for SonarQube and PMD to enforce some security architectural constraints, these rules are not expressed at the proper level of abstraction. This was made especially apparent in PMD, where several workarounds had to be devised in order to express the constraints.

6.4 Threats to validity

This section presents the validity threats of the thesis.

6.4.1 Construct validity

The primary threat to construct validity is whether the constraints considered in the study increases the security of a system. While we do not consider the enforcement of our constraints to provide a holistic approach to security, the relevance of providing a more secure system has been asserted by gathering established security measures.

6.4.2 Internal validity

In the absence of any preexisting ground truth, the threat of misinterpreting the architectural design of a system, and consequently, establishing an invalid ground truth is inherent. We reduced the threat of misinterpretation by having each author independently establish a ground truth, later comparing the results and carefully discussing the differences. Additionally, in cases where both of us were uncertain, our supervisors were consulted for further knowledge.

For our seventh constraint, “secrets must not be sent to a logger”, we found no violations as the developers of iTrust had ensured that the logger only received the appropriate information. To still be able to test the reliability of the extensions made to ArchUnit, we used data that was seen to be sent to the logger, although the data was non-sensitive. However, the functionality of SecArchUnit to support information flow constraints was still tested.

As the constraints were implemented sequentially, the time required to implement each constraint in a tool is affected by a carry-over effect as we learn how to use the tool. Similarly, our knowledge of SecArchUnit, compared to the tools used for comparison, was greater, resulting in a biased result. The ramifications of both are minimized by excluding the first two constraints as a learning experience, and then considering the total time required to implement the remaining constraints.

6.4.3 External validity

We formed the constraints used in the study to be generally applicable and not dependent on any specific domain or framework. Additionally, the reference to previously established security patterns, weaknesses, or constraints should provide some assurance of the validity of the constraint to projects other than those included in the study. However, as no previous study has applied them to a broad set of projects, we cannot draw any conclusion on the general validity.

Our choice of sample systems used in the evaluation represents another threat to external validity. We weighted several factors when deciding on which projects to use, such as the availability of architectural documentation, the size of the system, and whether they had been used in previous literature. Many open-source projects were discarded based on the fact that they did not provide any documentation for which we could establish the ground truth. Naturally, the size of a system profoundly affected the time needed for the architectural analysis, further reducing the set of systems that we could include. Thus, we tried to strike a balance between including project large enough to provide a realistic context, while at the same time being sufficiently small to allow for a rigorous analysis. We believe that while all three sample systems had initially been developed in the context of teaching, their implementation represents a realistic system. Additionally, the size of iTrust is large enough to warrant further validity.

7

Conclusion

Future research

Productivity when applying the tool to a new project

Decreased architectural violations over time

Bibliography

[1] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner, "Security Testing," in *Advances in Computers*. Elsevier, 2016, vol. 101, pp. 1–51. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0065245815000649

[2] N. Yoshioka, H. Washizaki, and K. Maruyama, "A survey on security patterns," *Progress in Informatics*, no. 5, p. 35, Mar. 2008. [Online]. Available: http://www.nii.ac.jp/pi/n5/5_35.html

[3] G. Mcgraw, "Software security," *IEEE Security & Privacy Magazine*, vol. 2, no. 2, pp. 80–83, Mar. 2004. [Online]. Available: http://ieeexplore.ieee.org/document/1281254/

[4] I. Arce, K. Clark-Fisher, N. Daswani, J. DelGrosso, D. Dhillon, C. Kern, T. Kohno, C. Landwehr, G. McGraw, B. Schoenfield, and others, "Avoiding the top 10 software security design flaws," *IEEE Computer Society*, 2014.

[5] S. Jasser, "Constraining the Implementation Through Architectural Security Rules: An Expert Study," in *Product-Focused Software Process Improvement*, X. Franch, T. Männistö, and S. Martínez-Fernández, Eds. Cham: Springer International Publishing, 2019, vol. 11915, pp. 203–219. [Online]. Available: http://link.springer.com/10.1007/978-3-030-35333-9_15

[6] L. de Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, Jan. 2012. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121211002044

[7] J. van Gurp and J. Bosch, "Design erosion: problems and causes," *Journal of Systems and Software*, vol. 61, no. 2, pp. 105–119, Mar. 2002. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0164121201001522

[8] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modules," in *Pioneers and Their Contributions to Software Engineering*, M. Broy and E. Denert, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1972, pp. 479–498. [Online]. Available: http://link.springer.com/10.1007/978-3-642-48354-7_20

[9] D. Parnas, “On the Design and Development of Program Families,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 1, pp. 1–9, Mar. 1976. [Online]. Available: http://ieeexplore.ieee.org/document/1702332/

[10] D. E. Perry and A. L. Wolf, “Foundations for the study of software architecture,” *ACM SIGSOFT Software Engineering Notes*, vol. 17, no. 4, pp. 40–52, Oct. 1992. [Online]. Available: http://portal.acm.org/citation.cfm?doid=141874.141884

[11] D. Garlan and M. Shaw, “AN INTRODUCTION TO SOFTWARE ARCHITECTURE,” in *Series on Software Engineering and Knowledge Engineering*. WORLD SCIENTIFIC, Dec. 1993, vol. 2, pp. 1–39. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/9789812798039_0001

[12] L. Bass, P. Clements, and R. Kazman, *Software architecture in practice*, 3rd ed., ser. SEI series in software engineering. Upper Saddle River, NJ: Addison-Wesley, 2013.

[13] A. Jansen and J. Bosch, “Software Architecture as a Set of Architectural Design Decisions,” in *5th Working IEEE/IFIP Conference on Software Architecture (WICSA’05)*. Pittsburgh, PA, USA: IEEE, 2005, pp. 109–120. [Online]. Available: http://ieeexplore.ieee.org/document/1620096/

[14] M. Broy, J. Grünbauer, and C. A. R. Hoare, Eds., *Software system reliability and security*, ser. NATO security through science series. Amsterdam ; Washington, DC: IOS Press, 2007, no. v. 9, oCLC: ocn127107624.

[15] C. Haley, R. Laney, J. Moffett, and B. Nuseibeh, “Security Requirements Engineering: A Framework for Representation and Analysis,” *IEEE Transactions on Software Engineering*, vol. 34, no. 1, pp. 133–153, Jan. 2008. [Online]. Available: http://ieeexplore.ieee.org/document/4359475/

[16] S. Genaim and F. Spoto, “Information Flow Analysis for Java Bytecode,” in *Verification, Model Checking, and Abstract Interpretation*, D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, and R. Cousot, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, vol. 3385, pp. 346–362. [Online]. Available: http://link.springer.com/10.1007/978-3-540-30579-8_23

[17] J. Aldrich, C. Chambers, and D. Notkin, “ArchJava: connecting software architecture to implementation,” in *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*. Orlando, FL, USA: ACM, 2002, pp. 187–197. [Online]. Available: http://ieeexplore.ieee.org/document/1007967/

[18] M. Abi-Antoun and J. M. Barnes, “Analyzing Security Architectures,” in *Proceedings of the IEEE/ACM International Conference on Automated*

Software Engineering, ser. ASE '10. New York, NY, USA: ACM, 2010, pp. 3–12, event-place: Antwerp, Belgium. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859001

[19] D. Luckham and J. Vera, "An event-based architecture definition language," *IEEE Transactions on Software Engineering*, vol. 21, no. 9, pp. 717–734, Sep. 1995. [Online]. Available: http://ieeexplore.ieee.org/document/464548/

[20] M. Abi-Antoun and J. Aldrich, "Static extraction and conformance analysis of hierarchical runtime architectural structure using annotations," in *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications - OOPSLA 09*. Orlando, Florida, USA: ACM Press, 2009, p. 321. [Online]. Available: http://dl.acm.org/citation.cfm?doid=1640089.1640113

[21] J. Knodel and D. Popescu, "A Comparison of Static Architecture Compliance Checking Approaches," in *2007 Working IEEE/IFIP Conference on Software Architecture (WICSA'07)*. Mumbai, India: IEEE, Jan. 2007, pp. 12–12. [Online]. Available: http://ieeexplore.ieee.org/document/4077029/

[22] A. Jansen, J. Bosch, and P. Avgeriou, "Documenting after the fact: Recovering architectural design decisions," *Journal of Systems and Software*, vol. 81, no. 4, pp. 536–557, Apr. 2008. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016412120700194X

[23] Hong Yan, D. Garlan, B. Schmerl, J. Aldrich, and R. Kazman, "DiscoTect: a system for discovering architectures from running systems," in *Proceedings. 26th International Conference on Software Engineering*. Edinburgh, UK: IEEE Comput. Soc, 2004, pp. 470–479. [Online]. Available: http://ieeexplore.ieee.org/document/1317469/

[24] J. C. S. Santos, K. Tarrit, and M. Mirakhorli, "A Catalog of Security Architecture Weaknesses," in *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. Gothenburg, Sweden: IEEE, Apr. 2017, pp. 220–223. [Online]. Available: http://ieeexplore.ieee.org/document/7958491/

[25] E. Gamma, Ed., *Design patterns: elements of reusable object-oriented software*, ser. Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley, 1995.

[26] R. Scandariato, K. Yskout, T. Heyman, and W. Joosen, "A system of security patterns," Katholieke Universiteit Leuven, Belgium, Tech. Rep. CW 469, Dec. 2006.

[27] A. Eden and R. Kazman, "Architecture, design, implementation," in *25th International Conference on Software Engineering, 2003. Proceedings.* Portland, OR, USA: IEEE, 2003, pp. 149–159. [Online]. Available: http://ieeexplore.ieee.org/document/1201196/

[28] K.-J. Stol and B. Fitzgerald, “The ABC of Software Engineering Research,” *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 3, pp. 1–51, Sep. 2018. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3276753.3241743

[29] S. Peldszus, K. Tuma, D. Struber, J. Jurjens, and R. Scandariato, “Secure Data-Flow Compliance Checks between Models and Code Based on Automated Mappings,” in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Munich, Germany: IEEE, Sep. 2019, pp. 23–33. [Online]. Available: https://ieeexplore.ieee.org/document/8906984/

[30] Q. Luo, A. Nair, M. Grechanik, and D. Poshyvanyk, “FOREPOST: finding performance problems automatically with feedback-directed learning software testing,” *Empirical Software Engineering*, vol. 22, no. 1, pp. 6–56, Feb. 2017. [Online]. Available: http://link.springer.com/10.1007/s10664-015-9413-5

[31] S. Heckman, K. Stolee, and C. Parnin, “10+ Years of Teaching Software Engineering with iTrust: The Good, the Bad, and the Ugly,” in *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 2018, pp. 1–4.

[32] J. Cleland-Huang, O. Gotel, and A. Zisman, Eds., *Software and Systems Traceability*. London: Springer London, 2012. [Online]. Available: http://link.springer.com/10.1007/978-1-4471-2239-5

[33] A. K. Massey, P. N. Otto, and A. I. Antón, “Aligning Requirements with HIPAA in the iTrust System,” in *2008 16th IEEE International Requirements Engineering Conference*. Barcelona, Spain: IEEE, Sep. 2008, pp. 335–336. [Online]. Available: http://ieeexplore.ieee.org/document/4685700/

[34] J. Bürger, D. Strüber, S. Gärtner, T. Ruhroth, J. Jürjens, and K. Schneider, “A framework for semi-automated co-evolution of security knowledge and system models,” *Journal of Systems and Software*, vol. 139, pp. 142–160, May 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S016412121830027X

[35] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie, “Automated extraction of security policies from natural-language software documents,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE ’12*. Cary, North Carolina: ACM Press, 2012, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2393596.2393608

A

BlogWeb

See full-scale UML diagram on the next page.

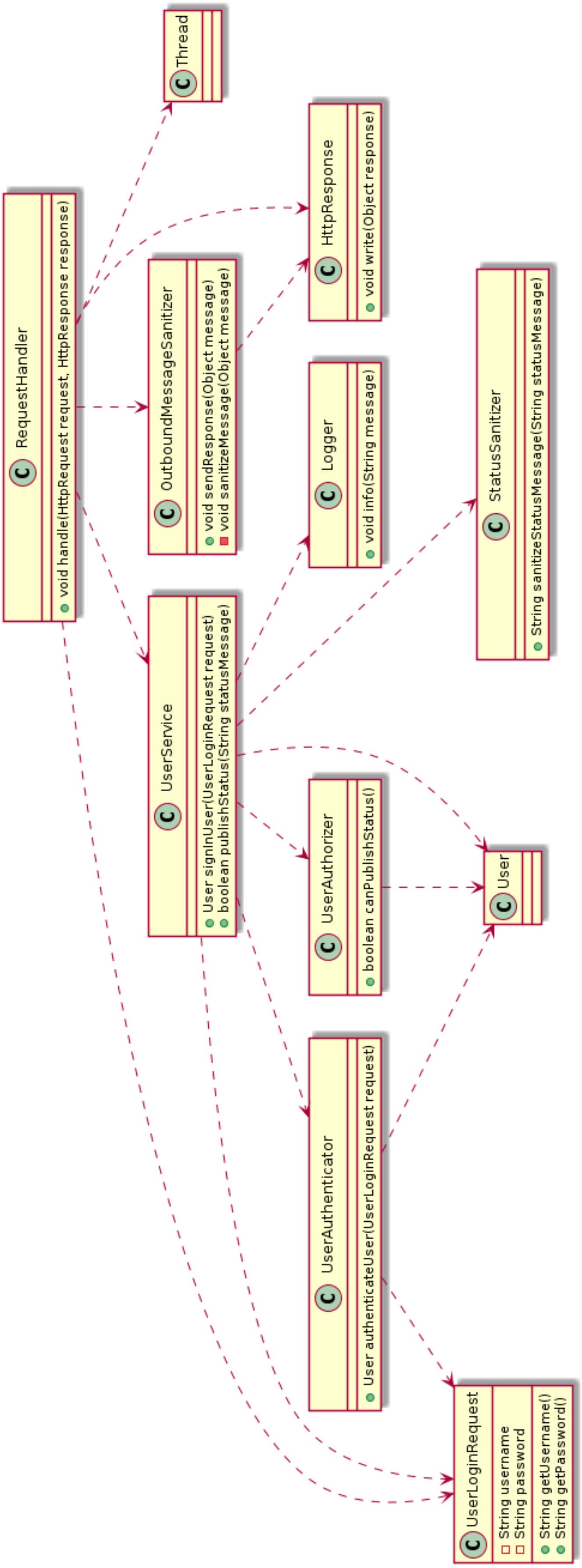

Figure A.1: A UML class diagram of the toy system BlogWeb.

B

SonarQube Rule Definition

```java
class CentralMessageRule extends IssuableSubscriptionVisitor {
    private static final String SENDING_POINT = "Transaction";
    private static final MethodMatchers SENDERS = MethodMatchers
        .create()
        .ofTypes("atm.physical.NetworkToBank")
        .anyName()
        .addParametersMatcher(parameters -> !parameters.isEmpty())
        .build();

    @Override
    public List<Tree.Kind> nodesToVisit() {
        return Collections.singletonList(Tree.Kind.METHOD_INVOCATION);
    }

    @Override
    public void visitNode(Tree tree) {
        MethodInvocationTree methodInvocation =
            (MethodInvocationTree) tree;

        if (SENDERS.matches(methodInvocation)) {
            // Get class where method invocation takes place
            Tree parent = methodInvocation.parent();
            while (!parent.is(Tree.Kind.CLASS)) {
                parent = parent.parent();
            }
            ClassTree classTree = (ClassTree) parent;

            // Compare class with sending point
            String sendingClassName = classTree.symbol().name();
            if (!SENDING_POINT.equals(sendingClassName)) {
                reportIssue(methodInvocation, "Messages must only be sent from the sending point");
            }
        }
    }
}
```

Listing B.1: SonarQube custom rule that enforces constraint 3.

C

PMD Rule Definition

```java
class CentralSendingRule extends AbstractJavaRule {
    private static final String SENDING_POINT =
        "atm.transaction.Transaction";
    private static final Collection<String> SENDERS =
        Arrays.asList(
            "atm.physical.NetworkToBank"
        );

    public CentralSendingRule() {
        super();

        // Types of nodes to visit
        addRuleChainVisit(
            ASTClassOrInterfaceBodyDeclaration.class
        );
    }

    @Override
    public Object visit(
            ASTClassOrInterfaceBodyDeclaration body,
            Object data) {
        ASTClassOrInterfaceDeclaration owningClass =
            body.getFirstParentOfType(
                ASTClassOrInterfaceDeclaration.class
            );
        boolean isSendingPoint = owningClass != null &&
            SENDING_POINT.equals(owningClass.getBinaryName());
        if (isSendingPoint) {
            // Method defined in sending point; skip this method
            return data;
        }

        Util.getMethodCallsFrom(body).stream()
            .filter(call -> SENDERS.contains(call.targetOwner)
                && call.argumentCount > 0)
            .forEach(offendingCall -> addViolation(data,
               offendingCall.source));

        return data;
    }
}
```

Listing C.1: PMD custom rule that enforces constraint 3.