# Extracting and Analyzing the Implemented Security Architecture of Business Applications

Bernhard J. Berger, Karsten Sohr and Rainer Koschke
Center for Computing Technologies (TZI), Universität Bremen
Bremen, Germany
{berber|sohr|koschke}@tzi.de

***Abstract*—Security is getting more and more important for the software development process as the advent of more complex, connected and extensible software entails new risks. In particular, multi-tier business applications, e.g., based on the Service-Oriented Architecture (SOA), are vulnerable to new attacks, which may endanger the business processes of an organization. These applications consist often of legacy code, which is now exported via Web Services, although it has originally been developed for internal use only. The last years showed great progress in the area of static code analysis for the detection of common low-level security bugs, such as buffer overflows and cross-site scripting vulnerabilities. However, there is still a lack of tools that allow an analyst to assess the *implemented* security architecture of an application. In this paper, we propose a technique that automatically extracts the implemented security architecture of Java-based business applications from the source code. In addition, we carry out threat modeling on this extracted architecture to detect security flaws. We evaluate and discuss our approach with the help of two commercial real-world case studies, one taken from the e-government domain and the other one from logistics.**

***Index Terms*—reverse engineering; software security; static analysis; threat modeling**

## I. Introduction

Software security plays more and more an important role. Specifically, the increasing connectivity of IT systems is leading to higher risks. Many formerly local legacy applications are now exported via new technologies, such as the Service-Oriented Architecture (SOA). The security research community and later industry has addressed these new challenges in several ways, most notably, by employing static code analysis [1], [2], [3], [4], [5], [6]. Static code analyzers, such as Fortify SCA [7] or Coverity Prevent [8], help an analyst to detect common low-level security bugs like buffer overflows, cross-site scripting and SQL injection vulnerabilities. Recent work has applied static code analyzers also to mobile applications [9]. Although code analyzers are effective in finding these low-level bugs, they cannot detect architectural security flaws, e.g., illogical role-based access control [10].

Conversely, there are several approaches attempting to (formally) specify and validate an application's security architecture, e.g., model-driven security [11] or UMLsec [12]. Specifically, if the architecture is formalized with modeling languages such as the Unified Modeling Language (UML), model-driven development can be employed to automatically generate high-quality code [11]. However, even if this approach is widely used in practice, there are still lots of legacy systems that do not use the model-driven approach. In addition, the generated code might be modified in an ad hoc fashion due to pressing customer requirements. Consequently, it is still unclear whether the software architecture is in sync with the implementation. Also, threat modeling as introduced by Microsoft [13] does not consider the *implemented* architecture, although it is an important step of a Secure Software Development Lifecycle [1]. It still remains unclear whether the implementation reflects the specified and approved architecture.

To overcome the aforementioned shortcomings, we propose to use techniques and tools from reverse engineering to extract the implemented security architecture from the source code of Java-based business applications. For this purpose, we employ the Soot tool [14], a widely-used static analysis framework for Java, as well as Bauhaus [15], a reverse engineering tool-suite. Our approach decomposes a multi-tier business application (e.g. consisting of client, web, application and data tiers) into single components and identifies information flows between these different components. Proceeding this way, we obtain a high-level overview of the implemented software architecture of enterprise applications. The extracted architecture can be used as input to a threat-modeling process to identify security flaws [13]. Furthermore, we enrich the extracted architecture with information about actually implemented security measures and check for architectural security vulnerabilities.

If a documentation of the architecture is already available, we can additionally employ the reflexion analysis [16], [17]. The reflexion analysis automatically checks the implemented against the specified architecture and reports on divergences between both architectures. Specifically, if threat modeling has already been conducted on the modeled architecture, we need to consider only the differences between the implementation and the specified architecture.

We evaluate our approach with the help of two real-world case studies, namely, an e-government application and a logistics application.

In summary, our contributions are as follows:

1) automated extraction of the security architecture from the source code by employing static code analysis as well as reverse engineering,
2) automated analysis of the extracted security architecture with respect to architectural vulnerabilities using threat modeling,

3) consideration of architectural security weaknesses as, for example, listed in the Common Weakness Enumeration (CWE) [18] as well as mitigations for common architectural threats,
4) a visualization plugin for Eclipse to inspect the results of the automatic reengineering process and to refine the generated models,
5) evaluation of our approach with two real-world case studies.

To the best of our knowledge, this is one of the first works combining well-established software-security practices such as threat modeling with reverse engineering techniques.

The remainder of this paper is organized as follows: Section II briefly describes the background of our work, whereas Section III gives an overview of our approach of extracting the implemented security architecture from code. The evaluation of our approach is described in Section IV, followed by a discussion of limitations and further prospects. After discussing related work in Section VI, we conclude our paper.

## II. Background

Our work builds on techniques of static code analysis and threat modeling, which are described subsequently.

### A. Static Code Analysis

In our approach, we employ static code analysis to extract an implemented software security architecture of existing systems. In particular, we use Soot [14] to analyze Java bytecode and extract an abstract and framework-independent security architecture that can be visualized with the Bauhaus tool suite [15]. We now briefly discuss both tools.

*1) The Soot Tool:* Soot is a widely-used static analysis framework for Java bytecode that was developed at McGill University. It supports a large set of standard analyses, such as call-graph generation, points-to analyses and reflection handling, that are necessary to solve sophisticated analysis problems statically. Soot uses a typed 3-address intermediate representation that can be transformed into a single static assignment (SSA) form, a well-known representation from compiler construction to conduct dataflow analysis.

The Soot framework integrates different points-to analyses to support different degrees of accuracy during the call-graph construction and later analysis. In particular, Soot implements context-sensitive and context-insensitive analyses to allow the user to balance between the accuracy of the results and the runtime and memory footprint.

*2) The Bauhaus Tool-Suite:* The Bauhaus tool-suite is a mature reverse-engineering platform that focused very early on the topic of reverse-engineering software architectures. Beyond its visualization capabilities, it is able to check an existing architecture for conformance with a defined one to identify the differences between them. For this purpose, it implements the reflexion analysis [16], [17].

### B. Threat Modeling

Threat Modeling was introduced by Microsoft as a process of reviewing the security of an application's architecture in order to detect security design flaws [19], [13]. A strategy to conduct threat modeling methodically from the attacker's point of view is STRIDE, which is also presented in the aforementioned publications. During the STRIDE process, the system is first decomposed into different processes, entities and dataflows. This step can be documented with dataflow diagrams (DFDs). Afterwards, a security expert writes down existing threats and assigns them to the different STRIDE categories, which are explained below in more detail. Finally, the planned mitigations are written down and assigned to the threats they address.

*1) Dataflow Diagrams:* Within the frameworks of threat modeling, the system architecture is described graphically by means of DFDs. DFDs support five different types of modeling elements, namely, processes, data stores, connections, trust boundaries as well as interactors [13]. Figure 1 gives an example of a DFD. Circles represent processes, which are running programs; arrows stand for dataflows (e.g., network connections, API calls, or remote procedure calls). Examples of data stores are database tables or files (represented as parallel lines). External entities, which are not subject to a security analysis, such as users or external systems, are depicted as rectangles, whereas dashed lines denote trust boundaries. A trust boundary divides a system into parts of different trust levels which implies that certain measures have to be undertaken to guarantee the security of the system.

Swiderski and Snyder mention the possibility to decompose systems hierarchically and to use different diagrams to show different levels of decomposition. To express the membership of nodes, each top-level node has a unique number and all nodes that are contained by that node start with the same number, followed by a dot and a unique child number.

Figure 1 shows a dataflow diagram for a typical Java-based enterprise system with a web interface. Some external clients exist, such as a web browser as well as web-service clients that communicate with the web-container process. The web-container process in turn exchanges data with the application container (named "EJB container" in Figure 1), which persists domain data to a database. Each process reads some configuration data—modeled as data stores—and the database process additionally stores data to its data store. Furthermore, we have defined two trust boundaries; the first one—with the longer dashes—stands for the machine boundary and the second one for a process boundary because the web container and the application container mostly run within the same process to improve performance.

*2) Threat Categories in STRIDE:* In addition to the DFDs, the STRIDE approach defines six threat categories. In fact, STRIDE is an acronym, which stands for the six attack categories *spoofing*, *tampering*, *repudiation*, *information disclosure*, *denial of service*, and *elevation of privilege*. These threats shall be applied to each element of a DFD during the threat

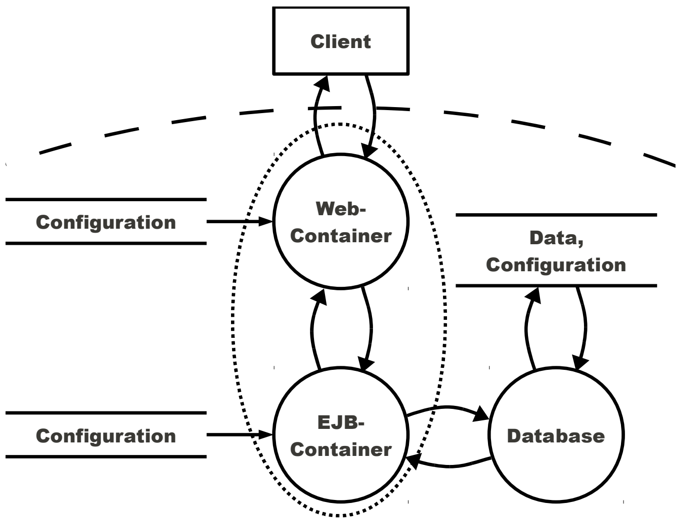

Figure 1. Exemplary dataflow diagram for a dynamic Java-based web site.

modeling process. Based on the determined threats, appropriate mitigation strategies have to be developed (e.g., authentication and authorization mechanisms).

To illustrate the aforementioned categories, we extend the example introduced in the previous section and give some possible threats for the dataflow between the client and the web container.

**Spoofing** Authentication schemes can be improperly implemented, e.g. by using IP addresses as authentication credentials (see also the CWE entry CWE-602 [20]). An attacker may forge its IP address to get access to the system.

**Tampering** An attacker may try to manipulate the data on the communication channel between the client and the web container. Digital signatures (e.g., based on XML security) are a possible solution to this threat, specifically, if additionally end-to-end security is needed.

**Repudiation** An attacker may repudiate a certain transaction. Again, digital signatures (maybe augmented with timestamps) might be a mitigation in this case.

**Information Disclosure** An attacker may try to eavesdrop on the communication between the client and the web container. A valid mitigation would be to ensure that SSL/TLS is used for transport security (see [21]).

**Denial of Service** The enterprise application may be flooded with requests, in particular, the database process may be attacked. Often, such attacks are mitigated on the level of the IT infrastructure rather than by the application itself.

**Privilege Escalation** An attacker can try to bypass client-side authorization by communicating directly with the web container (see also CWE-602 [22]). If the web container has been set up correctly, it enforces an access-control policy, which mitigates this threat.

*3) Annotated Dataflow Diagrams:* Beyond the DFDs described by Swiderski et al., Dhillon introduces *annotated* dataflow diagrams to increase their expressiveness [23]. An annotation holds additional information that cannot be modeled with traditional dataflow diagrams, such as used programming languages, the type of dataflow, encrypted or authenticated dataflows, and sensitive data. Using these annotations, a dataflow diagram is enriched with design decisions, which help an analyst performing threat modeling more effectively.

Returning to Figure 1, one can imagine useful annotations to improve the dataflow diagram. For instance, it would be helpful to mark the dataflows between the client and the web container with respect to encryption (encrypted, unencrypted) or the kind of database (e.g., SQL-based or object-based).

*4) Interaction Points:* Another practical refinement, which Dhillon introduces, is the concept of *interaction points* [23]. These are dataflows where data enters the system under investigation. The motivation behind introducing interaction points is that only those elements an attacker can interact with can be used as a basis of an attack. Consequently, this procedure saves an analyst from performing redundant work by focusing on the relevant parts of a DFD.

## III. Our Approach

Our goal is to enable software vendors to employ threat-modeling techniques to their *existing* software. Therefore, we aim to extract a security architecture from an implementation for two reasons. On the one hand, it simplifies the threat-modeling process and on the other hand, the implemented architecture differs from the documented architecture in most cases, a lesson learned from software architecture recovery [17].

Our approach is divided into different stages as illustrated in Figure 2. Starting with an implemented software, we employ reverse engineering techniques to extract a security architecture. This step is specific to the software frameworks that are used within the analyzed program. More details about this process are given in Section III-B. The extracted security architecture is framework independent and based on DFDs, which have been described in Section II-B1. Furthermore, we enrich the architecture automatically with annotations, which represent security measures, found in the implementation or within configuration files.

To take advantage of the annotated security architecture, we built a knowledge base which stores well-known threats and "best practices" on how to mitigate these threats. We match an extracted security architecture with the known threat patterns from our knowledge base and add corresponding threats to the STRIDE categories. A dataflow, for instance, can be annotated to transport sensitive information, such as credentials or other user-related data. A rule within our knowledge base maps this pattern to a threat for the confidentiality of the transported data, which is mapped to the STRIDE category "information disclosure". Afterwards, the DFD is scanned for annotations that represent mitigations for these threats; these annotations are then linked to the corresponding threats. In the above mentioned example, we link annotations that represent transport or message encryption for the transport channel to the threat of information disclosure since they prevent a possible attacker from reading the transferred information. Proceeding this way, we automatically detect security vulnerabilities at the architectural level. The rules are explained in Section III-C in more detail. It is also possible to inspect the generated security architecture manually to identify additional threats, not covered by our knowledge base.

Another useful option is to refine the generated security architecture with additional knowledge that cannot be found in the implementation and is related to the environment of the software, such as existing firewalls. This enhanced architecture can also be given as input to the aforementioned automatic checks.

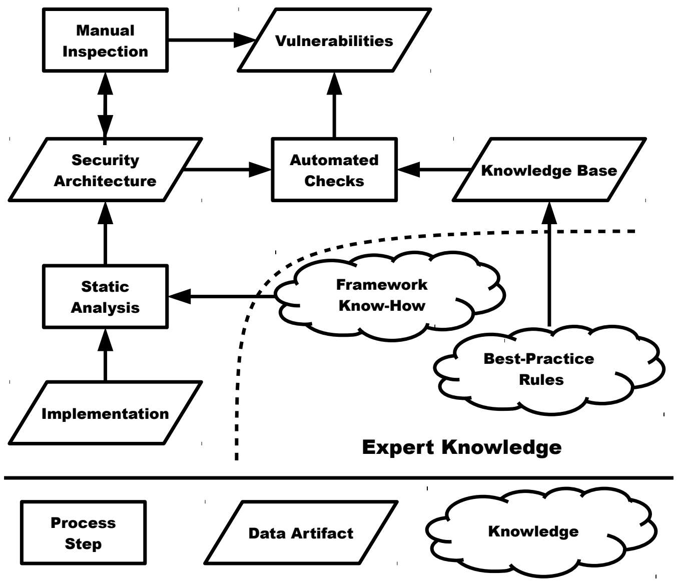

Figure 2. Process of extracting the implemented security architecture.

For didactic purposes, the security architecture as well as its representation are explained first because the following sections will refer to details of this architecture.

## A. Security Architecture

To model our security architecture, we use DFDs defined by Swiderski and Snyder [19] and Dhillon [23]. An abstract meta-model of our DFDs is depicted in Figure 3 in an UML-like notation, where we omitted all class attributes. Our model basically consists of hierarchical elements, trust boundaries, and dataflows. Each aforementioned model element can host an arbitrary number of annotations, which are in turn hierarchical as well, so we can express refinements of annotations. These refinements allow us to model threat patterns and corresponding mitigations at an abstract level, whereas the extraction phase generates low-level annotations. The hierarchy links the two levels in a form where concrete security measures are children of more abstract ones. To give an example, we can take a look at the annotation named “Encryption“, which symbolizes that a dataflow is encrypted. We split this general security measure into “Transport Encryption” and “Message Encryption”. A special kind of transport encryption is TLS over HTTP [21].

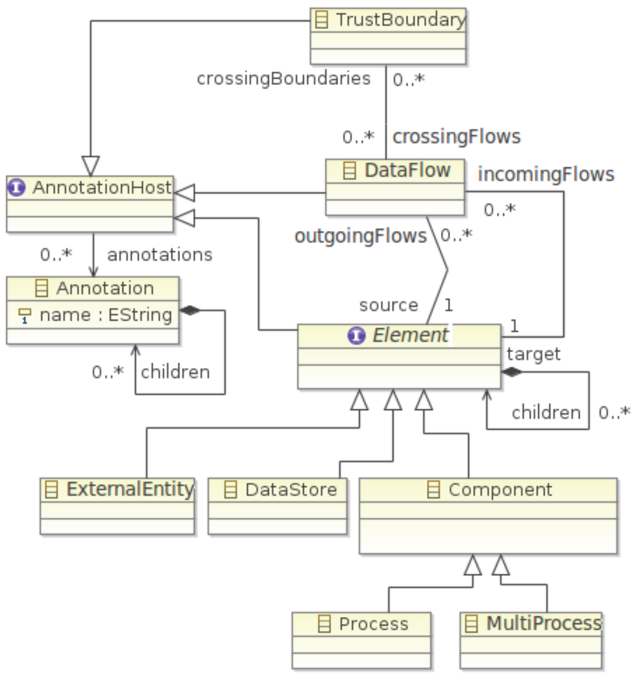

Figure 3. Meta-Model of our DFDs.

## B. Static Analysis

Reverse engineering a security architecture is a challenging task for different reasons. Firstly, it is necessary to understand and support all software frameworks used by an application. Depending on the framework, an analyst must use different strategies to gather as much information as possible from the source code. Secondly, one must find suitable mappings between implementation artifacts and the abstract architecture model.

Before stepping into the details, we give a high-level overview of the steps that constitute our analysis. Our static analysis process is divided into a sequence of five consecutive steps, depicted in Figure 4 and is also applicable to programs of other technical domains, such as Android applications. In a first step, the general procedure divides the system automatically into different components, such as processes, Web Services, Java Enterprise Beans, and dynamic web pages (also known as “Servlets”). Each such component (e.g., a web module) is transformed into a component in the DFD, which in general is hierarchical. After decomposing the application, each component is scanned for entry and exit points of data and control flow. These entry and exit points are then connected in a consecutive step, in which inter-component dataflows are determined. Based on information gathered from the different components, we derive trust boundaries during the next step, followed by the identification of already implemented security measures, which are then annotated at the architecture elements. Please note that it is nearly impossible to extract information about external entities from the source. Currently, we generate a single external “user” who communicates with all public interfaces, such as HTTP ports. It is a task of the refinement process to revise information about external entities.

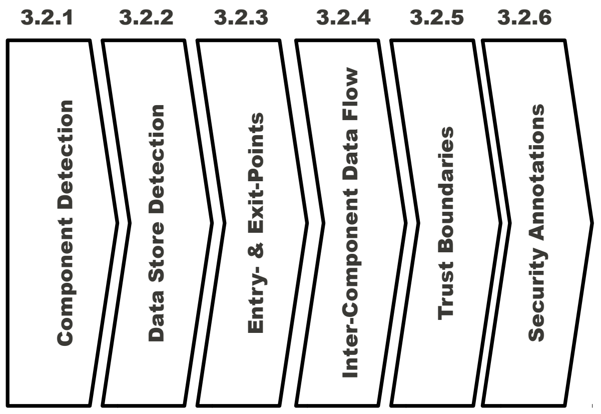

Figure 4. Static analysis process (displaying the numbers of the corresponding subsections detailing these steps at the top).

In the following subsections we describe some details of the separate steps of our static analysis and conclude with a discussion of a subset of the currently supported frameworks. We also report some challenges we faced during the case studies described in Section IV and how we were able to address them.

*1) Component Detection:* The component or process detection scans the deployed artifacts to find indicators for different processes or logical decomposition. These indicators depend on the frameworks that are used and in some cases, they can be found in additional configuration files deployed together with the compiled program. In a first step, all those checks are carried out that identify processes, and afterwards, each process is scanned for its substructure, such as web components, enterprise beans or web services. For each identified component, a corresponding DFD element is generated, which is then extended by its children.

*2) Data Store Detection:* To detect data stores, one must consider the source code as well as additional configuration files. Within these artifacts, one can identify access to different persistent objects, such as files and databases. Data-store detection is closely coupled with dataflow detection since the usage of a data store is always connected with a dataflow between a component and a data store.

*3) Entry- and Exit-Point Detection:* Entry and exit points are methods and calls that implement inter-process communication, user interaction or file access. Therefore, the entry and exit point detection searches for program points where data or control flows enter or leave the component under investigation. An example of an entry point are methods that can be called by a framework, for instance, on behalf of a web request. Such methods that are callable remotely may be attacked and therefore correspond to the interaction points, having been vaguely introduced by Dhillon. Since we cannot distinguish external entities reliably, we create one single artificial external entity that is connected to each entry point accessible for other processes.

Entry and exit points can be categorized into general and framework-specific ones. To the first group, for example, belong network communication and remote method invocation. The second group, for instance, contains web service methods, Enterprise Session Beans, and JSP files. Currently, we just support framework-specific entry and exit points since the systems in our case studies use only framework-specific entry and exit points.

*4) Extract Inter-Component Dataflow:* Up to this point, the different elements of the extracted DFD have not been connected yet. Therefore, we try to connect the entry and exit points of the different components where possible, i.e., we need to extract dataflow elements (see also Section II-B1), which is complicated in distributed systems where several processes communicate with each other. To address this problem, we use different kinds of static analyses augmented with framework know-how to find all existing dataflows within the analyzed system. More framework-specific details are given in Section III-B7.

*5) Trust Boundary Derivation:* The semantics and the implications of trust boundaries can be very different, which makes it hard to extract them automatically. A possible meaning of trust boundaries is that two processes rely on the proper functionality of each other to work correctly. Another meaning may be that all processes within this boundary run on the same trusted machine and therefore it is not necessary to use encryption or carry out additional access control checks.

Some of these trust boundaries can be derived from the deployment information that is stored along the application. However, it may be a quite rough estimation of the actual trust boundaries. Therefore, it is necessary to revise the trust boundaries manually after the extraction process.

*6) Security Annotations:* To represent the security rules and framework-specific security features, we created a set of predefined security annotations that can be extended during the extraction step as well as during the manual inspection. This allows the user to add additional information during the manual refinement step. As long as the annotations are associated with already known security concepts, they are considered during the automatic analysis steps.

*7) Supported Frameworks:* In the following subsections, we describe some of the frameworks we support and the DFD artifacts that we generated. Furthermore, we give some details of the used static analysis techniques.

*a) Java Applications:* Normal Java applications start with methods with the signature `public static void main (String [] args)`. To identify these methods, we can easily scan the type information for all classes belonging to the system. At the moment, we generate a process node for each matching method. Currently, we do not subdivide these processes, for instance into single threads. In the next step, we generate a context-(in)sensitive call graph by means of Soot for each main method to identify the part of the implementation that is used within the application. All reachable methods are then scanned for further elements of interest.

*b) Java Enterprise Edition:* Enterprise applications, implemented with the help of the *Java platform, Enterprise Edition* (JEE), are divided into different modules, each having its own task. There are modules for web frontends, fat clients, business logic, and libraries. All participating modules are listed in specific XML files, which are called deployment descriptors. For each enterprise application, a process node is generated, which contains sub-nodes for each defined module within the application.

Within the business logic modules, programmers use Enterprise Java Beans to expose the business logic to the front-end classes. Depending on the JEE version, Enterprise Java Beans are declared either with the help of configuration files or with the help of Java annotations. In some cases, Enterprise Java Beans are accessible for other processes, a fact that we note in the DFD by generating dataflows from an external user to the business logic module.

Until the version 1.4 of the JEE specification, Entity Beans are means to persist data to a database. If we find Entity Beans in the implementation, we read the according configuration files to determine the used database and create a corresponding data-store node.

For web modules, we search for existing Servlets, such as JavaServer Pages, which are Java classes that dynamically generate content for web sites. Since Servlets are exposed via HTTP, web modules are entry points to the system, an information we note in the DFD. A special kind of Servlet are web services, which are discussed below.

Beside this structural information, JEE offers a number of security-related services. Container-based authentication makes sure that each client of a web module is authenticated before it receives any result. Furthermore, container-based authorization is supported— based on role-based access control —and provides two different kinds of information. On the one hand we can derive the fact that authorization is used and on the other hand, we can determine different external entities, one for each role. Last but not least, JEE supports encryption of data sent to other hosts.

*c) Web Services:* Web services, which are implemented according to the Java API for XML web Services (JAX-WS), are software components that can be called remotely. For each web service, a DFD process element is generated. The interface of a web service is represented by WSDL and XML Schema files, describing the available operations, or by runtime-visible annotations in the Java source. Using these annotations, one can find the calls between clients and the web service. Based on these calls, dataflow edges are added to the DFD.

The web-service description can be extended with policies that enforce security measures on message level, such as message encryption, message signing, or a timestamp. These policies are translated into corresponding DFD annotations.

*d) File Access:* File access is available through a predefined set of Java classes, located in the `java.io` package. By means of Soot, we search for uses of these classes and extract the names of the files that are accessed. Therefore, it is necessary to conduct a flow-sensitive and inter-procedural constant propagation and to use points-to information to identify the file names (see Carini et al. [24]). Nevertheless, not all file names can be determined statically, for instance, if the file name is entered by the user, which leads to a number of unnamed files. These unnamed files symbolize every existing file that can be accessed on the machine that runs the program.

For each identified file, a data store element is generated with a file name (if it can be determined). A dataflow between the component that accesses the file and the data store element is also added to the DFD.

*e) Java Persistence API:* Based on Java annotations and the configuration file `persistence.xml`, we search for the use of the Java Persistence API, a standard Java API to persist objects to SQL-based databases. For each identified database, a data-store element is created and dataflow edges are added, depending on the usage of the database.

*f) Java Messaging Service:* The Java Messaging Service is a framework that supports means to asynchronously communicate with an arbitrary number of clients. Messages are sent using specific topic or queue names that every client needs to know. Therefore, we use inter-procedural constant propagation to identify the topics and queues a program is connected to. In general, this is not possible for all programs, since the names of the queues and topics might be constructed at runtime but this was not the case in our case study. Afterwards, we track all operations that are executed on the queues and topics to extract whether they are used for receiving or sending messages. Proceeding this way, we can find dataflows between applications, using the Java Messaging Service.

## C. Best Practices

To create our knowledge base containing well known threats as well as possible mitigations, we inspected the CWE [18], which lists a number of security problems, their consequences as well as potential mitigations (as CWE entries). Sometimes, the mitigations refer to the architecture and design phase. If possible, we translated the threats to DFD patterns as well as to annotation-based mitigations. During this process, we created a set of annotations that provide information about the security properties of DFD elements as well as mitigations to the security threats. We then defined rules that derive required security mitigations based on the security properties of the elements. These rules are formulated using the Object Constraint Language (OCL, see [25]) standardized by the Object Management Group (OMG) [26]. OCL allows one to specify constraints for a model instance on the level of the model's meta-model. In our case, the rules refer to the elements depicted in Figure 3.

A simple example rule is given in Listing 1. The rule checks that if there is a dataflow with the *Data.Is_Confidential* annotation, then there must also be the annotation *DataFlow.Is_-Encrypted*. With the help of this simple rule, we can test for the CWE-5 *J2EE Misconfiguration: Data Transmission Without Encryption* entry [27].

Listing 1. An OCL rule, checking CWE-5.

```
context DataFlow inv EncryptedChannel:
self.annotations->
  exists(d|d.name = 'Data.Is_Confidential')
implies self.annotations->
  exists(d|d.name = 'DataFlow.Is_Encrypted')
```

### D. Implementation Aspects

We implemented our prototype by means of different frameworks, namely Bauhaus, Eclipse, and Soot. Bauhaus gives us a solid foundation for architecture reengineering and allows us to create visualizations of the architecture. Soot is used to implement all necessary static analyses to extract the DFDs from the application's bytecode. Finally, we used the modeling framework made available by Eclipse [28] for our DFD model.

Depending on the analysis problem, we employ different kinds of static analysis that are implemented in the Soot framework. Starting with simple type information for some tasks, we also use, where necessary, context-sensitive call graph generation. This gives us accurate information. Based on the generated call graph, we employ inter-procedural dataflow analysis, such as inter-procedural constant propagation, which includes evaluating statically the effect of operations on constant values.

The data gained from the static analysis is stored in the DFDs, which we modeled with the help of the EMF metamodel Ecore, which is depicted in Figure 3. One advantage of using Ecore is the possibility to use Eclipse OCL, an OCL implementation for ecore-based models. Furthermore, we started to implement a visualization using the Eclipse Graphiti project; a screenshot is depicted in Figure 5.

## IV. Case Studies

We conducted two case studies with commercial applications to answer a set of predefined questions. The applications are implemented in Java and make use of JEE. To better assess our findings, we discussed the result of our evaluation with developers as well as a quality assurance representative responsible for the software under investigation.

The questions we wanted to answer are the following:

**RQ1** Is the extracted DFD similar to the one that was created by a security-aware developer?

**RQ2** Which part of the architecture can be extracted automatically?

**RQ3** Can we find security issues with the proposed approach?

In this section, we first describe the applications that we analyzed, and thereafter summarize the results we obtained. At the end, we give a short summary of our findings.

### A. Case Study Set-up

Our first case study was selected from the e-government domain and is based on the Web Service technology. It is implemented using SOA and consists of web services, traditional JEE applications, and normal Java applications. During our case study, we focused on a subsystem that is responsible for creating qualified digital signatures[1] for arbitrary documents. In particular, this application makes available a batch-signature mechanism for signing many similar documents with the same key at the same time. This software contains all the aforementioned component types and uses different kinds of inter-process communication means. Due to the application's task, security is a crucial aspect.

The second case study is a commercial business application from the logistics domain that helps companies to declare goods electronically for import and export and is used by hundreds of customers every day. It is implemented following the JEE specification in version 1.4, and its source code comprises 600k LoC and more than 1000 dynamic web pages (JSP files). The software is offered to customers on a software-as-a-service basis, which implies different security requirements, such as confidentiality and non-repudiation.

### B. Results

We automatically extracted a security architecture for both case study objects and applied our knowledge base to automatically identify possible threats and framework-based security means. We inspected the results and discussed them with application experts.

*1) Case Study: E-Government Application:* The DFD that we extracted for the first case study needed only a few refinements to satisfy the needs of the security expert. It is depicted in Figure 5 where the names of the elements are changed for the sake of confidentiality. Our process detected a set of nine top-level components as well as a set of subcomponents that implement the provided functionality. Four of the top-level components are not depicted in Figure 5 because they were test programs that are related to third-party libraries that were not relevant for the expert.

Besides the structure, we also identified security measures, such as authentication based on client-certificates and the use of WS-Security [29] to enable message-level security and time stamping. We were not able to identify automatically vulnerabilities in the extracted security architecture because all identified threats were mitigated. This result is not surprising as the application has been evaluated according to the Common Criteria (CC), a standard for the security evaluation of IT products several times and therefore different kinds of security reviews have already been carried out.

There were several differences between the manually created DFD and the automatically extracted one. We found several processes that, according to the security expert, do not belong to the system (the aforementioned test programs related to third-party libraries). Nevertheless, these programs could be started by employees and therefore interact with the system. Furthermore, our technique was not able to detect the trust boundaries manually added to the diagram.

Nevertheless, the extracted security architecture helped us to understand and manually assess the security aspects of the

[1]Qualified digital signatures are legally binding in Germany and are hence the counterpart of hand-written signatures. Consequently, special measures must be put in place to secure the process of signature generation.

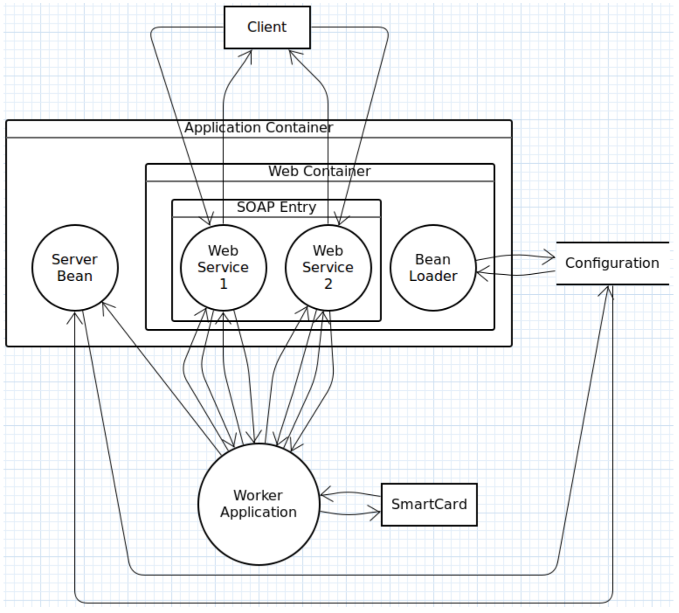

Figure 5. Extracted data-flow diagram of the e-government application.

system leading us to security-relevant components. Based on the extracted diagram, we were able to manually identify an authorization vulnerability allowing every authenticated client to use all smart cards to sign any document.

*2) Case Study: Customs Application:* From the second case study, we extracted a DFD with three top-level elements. There is an external entity—a web browser—that communicates with the application container using HTTP. The container exchanges data with a data store, namely, an SQL database. The overall architecture is similar to that depicted in Figure 1.

Based on the deployment information the application-container process is subdivided into a web frontend and a business logic part. We were able to refine the dataflow described earlier to the point that the external entity communicates with the web frontend. The web frontend in turn communicates with the business logic as well as with the database. The business logic itself is also connected via dataflow edges to the database.

We detected security measures that were used within the software, such as authentication, and we added this information to the DFD. Employing the resulting DFD, we applied relevant threats and mitigations, which are defined in our knowledge base, and detected several architectural vulnerabilities. We explain two of them in more detail now.

Our extraction reveals that the dataflow between the client and the web frontend transports authentication information over HTTP. Based on that fact, our analysis added the threat "An attacker captures authentication information by sniffing network traffic" to the *Spoofing* category of STRIDE. A valid countermeasure would be to use transport- or message-level encryption, but we could not find any hint concerning transport-level security by means of our analysis process. Inspecting the implementation, we found that the programmers failed to configure the framework correctly to ensure that TLS over HTTP is used. In particular, this problem corresponds to the entry CWE-5 *J2EE Misconfiguration: Data Transmission Without Encryption* (see above).

Furthermore, we refined the existing external entities because the developers told us that there were different kinds of users, normal users and administrators. This refinement led us to the threat "A client may bypass the authorization measures taken place in the client by directly communicating with the server." (CWE-602), which belongs to the *Privilege Escalation* category. To mitigate this threat, the server must perform the authorization itself, but our analysis did correctly not find any authorization check on the server side.

## V. Discussion and Threats to Validity

We automatically extracted two DFDs for commercial JEE systems and discussed two examples of architectural vulnerabilities as well as their underlying threats and possible mitigations. All this information was extracted from the source code of the JEE applications based on the approach described in Section III. The extracted DFDs helped us to understand the structure, the security requirements, and the security measures of the software although we did not know the internals of the program. Furthermore, we found that the extracted DFDs were similar to the ones that were created by the security experts (see **RQ 1**). On the one hand, we were able to identify different processes, data stores and the communication channels between these objects. On the other hand, we were not able to extract the trust boundaries and external entities accurately (see **RQ2**). This is not surprising because external entities, on the one hand, are not within the analysis scope. On the other hand, Hernan et al. state that trust boundaries are very subjective concepts [13]. In the second case study, we identified security vulnerabilities automatically employing our knowledge base (see **RQ3**).

Several threats to the validity of our results exist that can be attributed to the static analysis techniques as well as the environment, which we cannot analyze. There are some limitations concerning the threats and mitigations that can be found, too.

First of all, static analyses tend to be inaccurate based on the pessimistic assumptions they have to make to guarantee the soundness of the analysis. To improve the precision of the employed analysis, we use deployment and configuration information to reduce the number of false positives. Therefore, the results are only valid for the analyzed setup. Depending on the required accuracy of the static analysis, time and memory consumption are factors that must not be neglected. This problem increases with the number of used frameworks within the analyzed system as well as the size of the systems. Secondly, our approach detects only those security and inter-component communication means that belong to the supported frameworks. Consequently, this entails a large amount of implementation and engineering work.

A problem of our approach is to extract information about the environment of the system under investigation since it is not part of the implementation that we analyze. There is no possibility for us to detect different kinds of external entities, existing security measures (such as network firewalls and application-layer firewalls), and the distribution of the software to different machines is also hard or even impossible to extract. Nevertheless, the possibility to refine the extracted DFD allows an analyst to add this external information and take it into account for the analysis process.

At the moment, our knowledge base with architecture-related security flaws is still preliminary, extracted from CWE entries. An incomplete knowledge base may overlook possible threats.

In addition, not all kinds of threats and mitigations can be identified automatically. For instance, mitigations to the STRIDE category *Denial of Service* are very hard to implement at the application level because the process must handle all requests made by an attacker to determine whether the request is valid or not.

Another aspect is to apply our approach during the evaluation of software according to the CC security standard. Discussions with CC evaluators revealed that they often do not have the time and expertise to understand the implemented security architecture such that they must trust in the vendor's statements concerning software security [30]. Our approach helps one to pinpoint security-critical regions in the code and eases the work of an evaluator by identifying common architectural weaknesses.

## VI. Related Work

Static security analysis of software has evolved into an active research area over the years. There are several works on static checking for software security [2], [6], [4], [5], [31]. Important research prototypes from static security analysis are e.g. MOPS [5], Eau Claire [4], and LAPSE [2]. MOPS uses temporal logic as formalism and model checking to discover issues such as race conditions in C programs. The tool xg++ by Ashcraft and Engler was used to detect vulnerabilities in the Linux Kernel [6]. Eau Claire can detect general security problems, such as buffer overflows and race conditions, based on static type checking. Moreover, there is work by Livshits and Lam who present a tool to detect common low-level vulnerabilities, such as SQL injections and Cross-site-scripting vulnerabilities [2]. Felmetsger et al. employ the Daikon tool [32] to dynamically infer security specifications for web applications. Thereafter, they use a model checker to detect application logic vulnerabilities violating the specifications [31]. All the aforementioned approaches focus on finding common low-level security bugs. Our approach is complementary to all those works because we analyze the implemented architecture.

Other works deal with the topic of detecting covert channels in applications, e.g., based on non-interference properties [33]. For example, Myers et al. introduced the JFlow language, an annotation-based extension of Java, which allows a developer to define security labels on variables. Proceeding this way, hidden information flows, e.g., induced by an application's control flow, can be detected. This approach, however, assumes that a developer uses the annotation language and hence does not work with legacy code.

Another approach to software security is model-driven development, most notably, based on the concepts of UML as well as its constraint language OCL. For example, Basin et al. coined the term "model-driven security" (MDS) by introducing the security specification language SecureUML [11]. In particular, SecureUML focuses on RBAC and allows a developer to automatically generate an application's access control infrastructure via MDS. In parallel work, Jürjens et al. introduced the UMLSec language, also a security extension of UML, with a slightly broader application area [12]. Beyond access control, this specification language allows a developer to define confidentiality and integrity policies. As stated by McGraw, both languages tend to focus on security functionality and do not consider an attacker's view on software security [1]. In addition, MDS does not consider legacy code, and even if MDS were adopted by industry to a large scale, often the implemented architecture may erode over time from the specified architecture due to pressing ad-hoc customer requirements.

Jung et al. describe their SiSOA approach in several publications [34], [35], [36]. They extract security artifacts from SOA systems that are implemented in Java using Apache Tuscany[2]. The security artifacts are detected based on syntactic source-code elements, such as Java annotations and configuration elements. Afterwards, the artifacts are aggregated into security tags with the help of a knowledge base. Security tags are abstract security measures which ensure that certain security goals are fulfilled. Their approach is currently limited to systems using the Tuscany framework and is just evaluated by simple examples.

Abi-Antoun and Barnes [37] annotate the source code of an application to extract Ownership Object-Graphs statically. Ownership Object-Graphs represent a hierarchical runtime-architecture of the objects within a system. Furthermore, they compare the extracted graph with a given DFD to find data flows that are not allowed. The approach is currently tested for small and non-distributed systems.

Threat modeling takes an attacker's viewpoint in matching possible attacks to DFD elements [13] (see also Section II-B). However, again one cannot be sure that the implementation is in sync with the architecture developed by the threat modeling process. For example, based on the experience which was gained by practical projects at the EMC corp., Dhillon proposes to employ testing to detect inconsistencies between the modeled security architecture and its implementation as future work [23]. We, however, take the position of employing static security analysis to regain and analyze the implemented architecture. We gave a preliminary outline of our method in a position paper [38]. The position paper does not describe the technical details and has no evaluation.

In summary, our analysis method, which checks the implemented against the specified architecture, is complementary to well-established approaches to the static security analysis of software, such as bug finding, MDS, language-based security, and threat modeling. Our focus lies on combining techniques

[2]http://tuscany.apache.org/

from reverse engineering with architectural security analyses.

## VII. Conclusion and Outlook

In this paper, we described a method to automatically extract the security architecture from the source code of Java-based business applications with the help of static analysis. On this extracted architecture, we automatically carried out analyses to detect architectural vulnerabilities employing threat modeling. These security analyses are based on a knowledge base that we partly extracted from the Common Weakness Enumeration and that contains possible mitigations. To inspect and refine the extracted security architecture, we implemented a visualization plugin for Eclipse. Finally, we evaluated our approach with the help of two real-world case studies.

In future work, we will extend our framework to support other kinds of platforms and frameworks, such as Android. Along this way, we need to extend the knowledge base to support a larger set of threats and mitigations. Also, it would be useful to apply methods from artificial intelligence to find a better representation of the knowledge in form of rules.

## Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) under the grant 01IS10015B.

## References

[1] G. McGraw, *Software Security: Building Security In*. Addison-Wesley, 2006.

[2] B. Livshits and M. Lam, “Finding Security Vulnerabilities in Java Applications Using Static Analysis,” in *Proceedings of the 14th USENIX Security Symposium*, Aug. 2005.

[3] B. Chess and J. West, *Secure Programming with Static Analysis*. Addison-Wesley, 2007.

[4] B. Chess, “Improving Computer Security Using Extended Static Checking,” in *IEEE Symposium on Security and Privacy*, 2002, pp. 160–173.

[5] H. Chen and D. Wagner, “MOPS: an infrastructure for examining security properties of software,” in *Proceedings of the ACM Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2002, pp. 235–244.

[6] K. Ashcraft and D. Engler, “Using Programmer-Written Compiler Extensions to Catch Security Holes,” in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE Computer Society, 2002, p. 143.

[7] Fortify Software, “Fortify Source Code Analyser,” 2012, http://www.fortify.com/products.

[8] Coverity, “Coverity Prevent,” 2012, http://www.coverity.com.

[9] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, “A Study of Android Application Security,” in *Proceedings of the 14th USENIX Security Symposium*, Aug. 2011.

[10] J. H. Allen, S. Barnum, R. J. Ellison, G. McGraw, and N. R. Mead, *Software Security Engineering: A Guide for Project Managers (The SEI Series in Software Engineering)*. Addison-Wesley Professional, 2008.

[11] D. A. Basin, J. Doser, and T. Lodderstedt, “Model Driven Security: From UML Models to Access Control Infrastructures,” *ACM Transactions on Software Engineering Methodology*, vol. 15, no. 1, pp. 39–91, 2006.

[12] J. Jürjens and P. Shabalin, “Automated Verification of UMLsec Models for Security Requirements,” in *Proceedings of UML 2004 - The Unified Modelling Language: Modelling Languages and Applications*, ser. LNCS, vol. 3273. Springer, 2004, pp. 365–379.

[13] S. Hernan, S. Lambert, T. Ostwald, and A. Shostack, “Uncover Security Design Flaws Using the STRIDE Approach,” *MSDN Magazine*, Nov. 2006. [Online]. Available: http://msdn.microsoft.com/en-us/magazine/cc163519.aspx

[14] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot – a Java Bytecode Optimization Framework,” in *Proceedings of Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 1999, p. 13.

[15] A. Raza, G. Vogel, and E. Plödereder, “Bauhaus—A Tool Suite for Program Analysis and Reverse Engineering,” in *Ada-Europe*, ser. Lecture Notes in Computer Science, vol. 4006. Springer, 2006, pp. 71–82.

[16] G. C. Murphy, D. Notkin, and K. J. Sullivan, “Software Reflexion Models: Bridging the Gap between Design and Implementation,” *IEEE Transactions on Software Engineering*, vol. 27, no. 4, pp. 364–380, Apr. 2001.

[17] R. Koschke, “Incremental Reflexion Analysis,” in *European Conference on Software Maintenance and Reengineering*. IEEE Computer Society Press, Mar. 2010.

[18] MITRE Corporation, “The Common Weakness Enumeration (CWE) Initiative,” 2012, http://cwe.mitre.org/.

[19] F. Swiderski and W. Snyder, *Threat Modeling*. Redmond, WA, USA: Microsoft Press, 2004.

[20] MITRE Corporation, “The Common Weakness Enumeration (CWE) Initiative — CWE-290: Authentication Bypass by Spoofing,” 2012, http://cwe.mitre.org/data/definitions/290.html.

[21] E. Rescorla, “HTTP over TLS,” may 2000, http://tools.ietf.org/html/rfc2818.

[22] MITRE Corporation, “The Common Weakness Enumeration (CWE) Initiative — CWE-602: Client-Side Enforcement of Server-Side Security,” 2012, http://cwe.mitre.org/data/definitions/602.html.

[23] D. Dhillon, “Developer-Driven Threat Modeling: Lessons Learned in the Trenches,” *IEEE Security and Privacy*, vol. 9, no. 4, 2011.

[24] P. R. Carini and M. Hind, “Flow-sensitive Interprocedural Constant Propagation,” in *Proceedings ”of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation”*, ser. PLDI ’95. New York, NY, USA: ACM, 1995, pp. 23–31.

[25] Object Management Group, “OMG Object Constraint Language,” 2012, http://www.omg.org/spec/OCL/2.3.1/PDF/.

[26] ——, “Object Management Group,” 2012, http://www.omg.org/.

[27] MITRE Corporation, “The Common Weakness Enumeration (CWE) Initiative — CWE-5: J2EE Misconfiguration: Data Transmission Without Encryption,” 2012, http://cwe.mitre.org/data/definitions/5.html.

[28] The Eclipse Foundation, “Eclipse Modeling Framework,” 2012, http://www.eclipse.org/modeling/emf/.

[29] OASIS, “Web Services Security: SOAP Message Security 1.1,” 2006, https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf.

[30] S. Maseberg, “Personal communication,” 2011.

[31] V. Felmetsger, L. Cavedon, C. Kruegel, and G. Vigna, “Toward Automated Detection of Logic Vulnerabilities in Web Applications,” in *USENIX Security Symposium*. USENIX Association, 2010, pp. 143–160.

[32] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The Daikon System for Dynamic Detection of Likely Invariants,” *Sci. Comput. Program.*, vol. 69, pp. 35–45, December 2007.

[33] A. C. Myers, “JFlow: Practical mostly-static Information Flow Control,” in *POPL ’99: Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999, pp. 228–241.

[34] P. Antonino, S. Duszynski, C. Jung, and M. Rudolph, “Indicator-based Architecture-level Security Evaluation in a Service-oriented Environment,” in *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*, ser. ECSA ’10. New York, NY, USA: ACM, 2010, pp. 221–228.

[35] C. Jung, M. Rudolph, and R. Schwarz, “Security Evaluation of Service-Oriented Systems Using the SiSOA Method,” *International Journal of Secure Software Engineering*, vol. 2, no. 4, pp. 19–33, 2011.

[36] ——, “Security Evaluation of Service-oriented Systems with an Extensible Knowledge Base,” in *Proceedings of the Sixth International Conference on Availability, Reliability and Security (ARES)*. IEEE, 2011, pp. 698–703.

[37] 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. [Online]. Available: http://doi.acm.org/10.1145/1858996.1859001

[38] K. Sohr and B. Berger, “Idea: Towards Architecture-Centric Security Analysis of Software,” in *Engineering Secure Software and Systems*. Springer-Verlag, 2010.