Ensuring threat-model assumptions by using static code analyses

Johannes Geismann[1], Bastian Haverkamp[1] and Eric Bodden[1,2]

[1]Department of Computer Science, Heinz Nixdorf Institute, Paderborn University, Fürstenallee 11, 33102 Paderborn, Germany
[2]Fraunhofer IEM, Zukunftsmeile 1, 33102 Paderborn, Germany

Abstract

In the past years, the security of information systems has become more and more important. Threat modeling techniques are applied during the design phase of the development, helping to find potential threats as early as possible. However, assumptions made at this development step are often not considered in later steps or are not validated correctly, particularly not during the concrete implementation of the system. To overcome this problem, we present CARDS, a security modeling approach on the architectural level which utilizes code analyses to validate assumptions made during the threat modeling phase. CARDS helps ensure a correct implementation but also allows one to determine which effect code vulnerabilities can have on the overall architecture, as described through models. We implemented CARDS based on the Eclipse Modeling Framework, for Java-based system implementations. We evaluated CARDS based on the CoCoME case study to show its efficacy. The evaluation showed that CARDS can ease the validation of assumptions made during threat modeling and reduce the overall analysis effort.

Keywords

Threat-modeling, Security, Component-based, Static Code Analyses, Security-by-design

1. Introduction

Security is an essential property when developing modern software-intensive systems. To ensure high security, it is important to consider security not only during the implementation but already when designing the system. Especially dataflows are of high interest because confidential data resembles important assets for every information system, and also because attacker-controlled inputs need to be properly filtered before they are used. For this reason, one uses threat modeling approaches to reason about potential threats and corresponding countermeasures in early development steps [1].

Current approaches, however, are limited because of the lack of full traceability from threat model to the system artifacts. In particular, due to a missing connection of threat model artifacts and the implementation, this implementation often differs from the specifications made during threat modeling [2]. Hence, assumptions made during the design or in the threat model are not correctly implemented or not even implemented at all, which leaves the security state of the actual system unclear.

Static code analyses can help to validate these assumptions and are used to ensure that specific dataflows are prevented. For example, when paying in the super market, such an assumption on the implementation of the cash desk could be that customer credit card information is only sent to system parts that have permission to process it. Especially for large-scale systems this becomes a challenge because large code bases have to be analyzed. Additionally, such systems consist of several subsystems that are possibly developed by different parties. Distributed systems, micro-services and "serverless" architectures are just some prominent examples.

Particularly in these areas, model-based approaches are promising for threat modeling and security by design [3]. However, most approaches are either fully model-driven approaches that are quite heavy-weight and usually hardly adaptable, e.g, UMLsec [4] or SEED [5], or light-weight approaches such as STRIDE [1] that only take threat modeling into account but do not consider the connection to the implemented system. To make threat modeling more effective for distributed systems, the following challenges need to be met.

Security requirements are usually defined by several disciplines and, therefore, should be specified on the architectural or system level such that they can be discussed independently from—and in the best case already before—the implementation phase. Countermeasures defined during such a threat modeling phase are usually assumptions made about the implementation. Hence, all assumptions made on the architectural level have to be made explicit in the model and have to be correctly refined into source code [2]. Because this is a tedious and error-prone task, one must validate these assumptions on

2ND INTERNATIONAL WORKSHOP ON MODEL-DRIVEN ENGINEERING FOR SOFTWARE ARCHITECTURE (MDE4SA 2021), September 13th 2021, Virtual (originally Växjö, Sweden)
johannes.geismann@upb.de (J. Geismann); bastihav@mail.upb.de (B. Haverkamp); eric.bodden@upb.de (E. Bodden)
0000-0003-2015-2047 (J. Geismann); 0000-0002-1189-6290 (B. Haverkamp); 0000-0003-3470-3647 (E. Bodden)

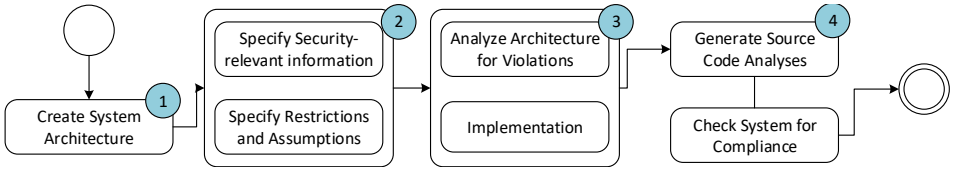

Figure 1: Overview and main process steps of CARDS.

the source code level. An additional challenge is that the implemented system is usually not completely under the control of one development team. Static code analyses are a suitable solution to this end because they validate such assumptions on the source code and can be defined for a specific subsystem regardless of who is responsible for the implementation. However, if static code analyses are used, the results are most useful if fed back to the architecture and threat model. Unfortunately, current solutions fall short in this regard.

We see two main concepts as essential here: Connection to the source code and making the requirements and assumptions made during threat modeling explicit. To address these challenges, we have developed CARDS (**C**omponent-based **A**ssumptions and **R**estrictions for **D**ataflow **S**pecifications), a security modeling approach for dataflows in distributed systems. It provides a new DSL which operates on a generic component model and, therefore, can be adapted for existing component-based approaches. CARDS can be used to specify security requirements for dataflows, as well as assumptions made to fulfill these restrictions on the architectural level. CARDS further illustrates how static code analyses can be used to validate the assumptions on the code level.

In particular, this paper makes the following original contributions:

- CARDS: a concept and a domain-specific language for the specification of dataflow restrictions and assumptions on the architectural level,
- an analyzer checking the system for dataflow violations,
- a concept for generating the corresponding static code analyses, and
- an implementation of these concepts based on the Eclipse modeling framework and Sirius, providing a textual as well as graphical syntax.

This paper is structured as follows: In Section 2, we provide an overview of CARDS, describe our concept for security restrictions and assumptions, explain our model analyses on and the generation of code analyses. In Section 3, we describe the implementation of the prototype and present the evaluation of CARDS in Section 4. Section 5 compares CARDS with related approaches and Section 6 concludes this paper.

The source code for our implementation can be found on https://github.com/secure-software-engineering/cards

2. CARDS: Security Modeling and Validation

Effective threat modeling requires four basic steps: (1) Finding security-relevant systems parts and functions, (2) Finding potential threats with regard to these parts, (3) Risk-assessment, i.e., prioritizing the threats, and (4) implementing appropriate countermeasures. While threat modeling in general targets all kind of threats, CARDS focuses on dataflow-specific threats. We designed CARDS in such a way that it's concepts can be applied to existing development processes. Figure 1 shows an overview of the main steps.

At first, the system designer creates a component model describing the basic *architecture* of the system (1). This step is not necessarily part of CARDS since an existing architectural model could also be adapted for the application of CARDS. Based on the component model, security and domain experts specify *security-relevant information*, e.g., confidential data. Also, *security restrictions* and *security assumptions* are specified explicitly. Security restrictions describe security requirements for specific data types of the system, e.g., data from the credit card reader are always sanitized before being sent to other components of the system. Dataflow-specific security requirements for the system can be refined to security restrictions. A security assumption makes an assumption to the implementation explicit, e.g., that confidential data will never be send to an external entity. Following this, a restriction describes global requirements the system should satisfy, an assumption contrarily describes what the designer assumes to be implemented for each component. The concept of both (1) and (2) are described in more detail in Section 2.1.

Next, the system can be analyzed whether all security restrictions are satisfied assuming that all assumptions will be implemented correctly (3). If a violated restriction is found, the security experts may add additional assumptions to mitigate this security issue and re-apply the analysis until all restrictions are satisfied. The as-

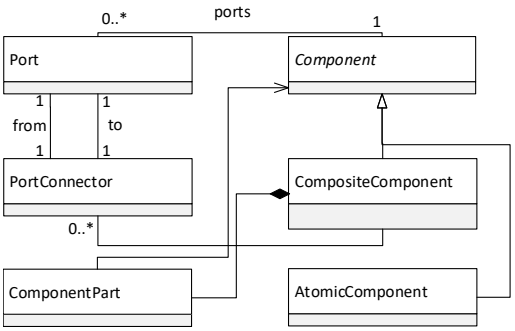

Figure 2: Overview of the used generic component model.

sumptions can be useful for the actual implementation of the system giving the developers guidelines for the implementation. Concepts for the analyses and potential use-cases in the development are explained in Section 2.2.

Finally, CARDS uses generated static code analyses to validate if all assumptions are implemented correctly (4). For this, we provide in Section 2.3 a concept for how the assumptions can be mapped to static code analyses automatically. If all generated analyses pass and no violation is found on source code, the restrictions made to the system can be seen as satisfied on code-level, too.

2.1. Specifying Restrictions and Assumptions

In this section, we explain our concepts of restrictions, assumptions, and all concepts required. We developed a DSL for specifying security-relevant information of the system, security restrictions, and security assumptions. Since it is essential to refer to the actual system model, this DSL refers to a component model. For demonstration purposes, we are using a generic component model which is described in Section 2.1.1. However, since we use a generic component model, we see our concepts not restricted to one component model but adaptable to other component models. After that, we describe in Section 2.1.2 how security-relevant information can be formalized. Finally, we explain our concept of restrictions and assumptions in more detail and describe our DSL for this step.

2.1.1. Component Model

For demonstration purposes, we are using a generic component model. We therefore expect that our concepts can be applied to most other component-based system specifications as well. Figure 2 depicts the main parts of the underlying meta-model. A component model consists of a set of components which can be either *CompositeComponents* or *AtomicComponents*. Composite components can contain further components by defining *ComponentParts* which allows for a hierarchical component model. Atomic components cannot contain further components. Components use *Ports* for communication with other components. In our component model, we assume communication to be asynchronous. Ports are connected via *PortConnectors* which are embedded into the parent composite component. For a better overview, we have omitted several parts of the meta-model that are mainly needed for technical reasons. The full meta-model can be found in our provided implementation artifacts.

2.1.2. Security-relevant Information

Based on the component model, CARDS utilizes several security-relevant pieces of information that can be specified within our DSL. In the following, we give a short overview of the supported language features and their purpose.

DataTypes are representing the security-relevant data. They are the data assets of the system because they represent the data that should be protected. We only consider data that are relevant for the analyses. DataTypes can have attributes for labels, e.g, to mark a datatype as external user input, a security level, and a type which can be interesting when mapping to the actual source code base. Listing 1 shows an excerpt of the example where three data types are defined (lines 1-5).

Data Groups are used to combine several DataTypes, e.g., all data describing parts of credit card information. DataGroups are mainly used when defining Restrictions and Assumptions. In Listing 1, the data types `CreditCardNumber` and `CreditCardPIN` are grouped (cf. line 11).

Component Groups are used similarly to combine components that have something in common, e.g., (un)trusted components.

Component Kinds can be used to categorize components, e.g., to mark components as external entities, datastores, or processes (similar to DFD threat modeling) [1].

Data Sources describe which components are the sources for a specific DataType. In Listing 1, the component `CardReader` is marked as source for the types `CredietCardPin` and `CreditCardNumber`.

Sanitzers are used to modify data making them secure for further use, e.g., escaping bad characters. At this stage, a sanitizer is only on conceptual level and can be used in the security assumptions (cf. Section 2.1.3). In the example, a

Listing 1: Example code of a CARDS-specification.

```
dataTypes {
 DataType BarCode { },
 DataType CreditCardNumber {securityLevel 3 },
 DataType CreditCardPin {securityLevel 4 }
}
components {
 AtomicComponent CardReader {
 ports { INOUTPort cardReaderPort ( )}
 sourceOf { CreditCardPin,CreditCardNumber }}
}
Groups {DataGroup CreditCardInfo {CreditCardPin,
     CreditCardNumber}}
Sanitizer {CCSanitizer}
```

sanitizer is defined that should sanitize all confidential credit card information, e.g., by replacing it with asterisks.

Security Level can be used to assign a specific level of security or trust to components.

Listing 2: Example of a restriction using CARDS-specification.

```
DataFlowRestrictions {
GloballyPREVENT CreditCardInfo {
 Comp CreditCardPin , CreditCardNumber allow CardReader ,
       Bank , CashDeskPC}}
```

Listing 3: Example code of security assumptions using CARDS.

```
DataFlowAssumptions {
  componentAssumptions {
    Component CashDesk neverOut CreditCardInfo }
  portAssumptions {
    Port pcLightDisplay neverOut CreditCardInfo
    Port pcCashBoxPort neverOut CreditCardInfo}
  sanitizersAssumptions {
    Component CashDeskPC sanitizes DataFlow
          pcCardReaderPort -> pcPrinterPort of
          CreditCardInfo using CCSanitizer}}
```

2.1.3. Dataflow Restrictions and Assumptions

In the following, we describe our concepts for security restrictions and corresponding assumptions and how CARDS supports the security engineer specifying these. Essentially, restrictions formally describe security requirements regarding the dataflow within the system. Assumptions are used to describe countermeasures that are assumed to be in place in the source code.

Specifying Restrictions Restrictions are used to formally describe security requirements for the data types specified as assets. In essence, the security engineer has to describe a security policy for each data type describing which component is allowed to access the data. Basically, there are two options: 1. Globally allow all components to access a data type and define exceptions that are not allowed to access the data type (deny-listing approach) and 2. globally prevent components from accessing the data type and define exceptions describing components that are allowed to access the data type (allow-listing approach).

Corresponding to this, we distinguish between two kinds of restrictions, so-called *Allow*-Restrictions and *Prevent*-Restrictions. For each datatype, the security engineer has to specify such a restriction. One restriction may cover more than one data type. Listing 2 shows an example of a specified restriction. In particular, we define a prevent restriction describing, that the data types `CreditCardPin` and `CreditCardNumber` should only be accessed by the components `CardReader`, `Bank`, and `CashDeskPC` by combining the prevent restriction and a component refinement. Beside component refinements, restrictions CARDS also supports refinements for component parts and component groups. Without any knowledge of the concrete behavior of the components, this restrictions could not be validated. The security engineer can therefore specify assumptions of the implemented behavior which must be met to achieve the restriction. We next explain how to specify such assumptions in CARDS.

Specifying Assumptions An assumption describes a required behavior of a component. CARDS provides different kinds of assumptions. At first, we distinguish between two major kinds of assumptions: *neverOut-assumptions* and *sanitzer-assumptions*. A neverOut-assumption specifies that a context element will never leak the given data type, e.g., that a component will never send private data to another component. A sanitizer-assumption specifies that a context element will always sanitize the data before leaking it using a specific sanitizer, e.g., replacing some digits with asterisks when sending credit card information to the printer.

We support three different context elements: components, ports, and flows within a component. Assumptions for component parts are not useful because all parts of a specific component type will have the same implementation. In the example in Listing 3, we show four different assumptions: 1. an assumption that the (composite) component `CashDesk` will never leak the credit card info (line 3). 2. an assumption that the `pcLightDisplay` port will never leak the credit card info (line 6). 3. an assumption that the `pcCashBoxPort` port will never leak the credit card info (line 7). 4. an assumption that the component CashDeskPc port will always sanitize dataflows of credit card info from `pcCardReaderPort` to `pcPrinterPort`, using the sanitizer CCSanitizer (line 10).

CARDS provides an analysis to check whether the specified restriction is satisfied on model level if all assumptions are implemented correctly. This analysis is ex-

plained in the next section. Section 2.3 describes our concept how the correct implementation of the assumptions can be validated using static code analyses.

2.2. Analysis and Reporting

CARDS provides model-based analyses checking whether all specified restrictions are satisfied and if all security assumptions have been implemented correctly. This analysis should be part of the threat modeling activity during system design and is also useful to find effects in the system's architecture when a problem in the actual implementation is found. The analysis can help security experts to find unintended dataflows and to specify requirements for the implementation of a component by creating security assumptions. Besides the analysis, CARDS also provides several reporting features to assist the security experts by exporting the analysis results in useful formats. In this section, we describe how our analysis works at first and how the results can be reported afterward.

In CARDS, we apply a two-step analysis. First, for each component, all possible paths through the model are determined. Second, for each component and component parts respectively, all data types are determined that might reach this component. For the first analysis, we treat the component model as a directed graph where components are the nodes and port connectors are the edges. Conceptually, the analysis is as a basic depth-first search. The output of the analysis is a mapping from components to all (longest) paths through the model, i.e., for each component, we store which components it could directly or indirectly communicate with. In the second analysis, for every component, a set of available data types is determined, i.e., data types that could possibly be accessed by this component. In the beginning, the set of available data types of all components that are a source for a data type are set to these data types. Next, the analysis recursively propagates data types through the system. The analysis iterates through the paths and, for each step in the path, adds all currently available data types to an output set which is again propagated to the next component in the path. In this step, we evaluate given assumptions of the component to alter the set of available data. If a sanitizer-assumption is specified for this component and datatype, we add a flag to the data type that it becomes sanitized by this component. If a neverOut-assumption is specified, the data type is removed from the output set. The output of this analysis is a mapping of components to pairs of lists of paths and data types, which are received on these paths. The advantage of this two-step analysis is that the result does not only show available data for each component but also which path is the source for a given datatype.

To find violations of restrictions, we check for each restriction if data types of the defined restriction are illegally accessible at a component. Based on the analysis results, we can compare the list of available data types and the list of data types specified in the restriction. If a violation is found, it is essential to report it to the engineers properly. For this, CARDS provides different report features, e.g., visual feedback in the graphical editors, exported HTML and JSON reports, and an export to attack-defense graphs [6].

2.3. Using Code Analyses for Validation

When all violations of dataflow restrictions are eliminated by specifying assumptions, these assumptions must also be met through correctly implemented source code. To validate this, we propose to use static code analysis (cf. Step 4 in Figure 1). We provide a general concept for creating static code analyses for the given model assumptions. Since these analyses base on a common structure, it is reasonable to generate them and, thus, automating this step. However, to generate the analysis, some manual prerequisites must be met, i.e., a connection between the model and the code base has to be created. In the following, we explain how we propose to create such a connection first and how the analyses can be generated automatically in a second step.

2.3.1. Connection to Source Code

For connecting the (secured) component model to a given code base, we propose to use a so-called *mapping model*. This mapping model is used to describe the connections between model artifacts and parts of the source code. All required mappings are shown in Table 1. All mappings have to specify the model element, class and a method.

Since creating all mappings by hand is a tedious task, we provide a source code generator that generates source code skeletons for a given composite component and also creates an appropriate mapping model containing all required mappings. As proof of concept, we implemented a generator for Java which is explained in Section 3 in more detail. Supporting the engineers in creating a mapping model for an existing code base is not in the scope of this paper but we see potential by applying semi-automatic approaches like done by Peldszus et al. [7]. However, both the mapping model itself and the generator are conceptually not restricted to one programming language but can be easily adapted for other programming languages.

2.3.2. Generating Static Analyses

After creating the mapping model, we use this information to create a suitable static code analysis. Since we tend to analyze the flow of information, we use a taint analysis to validate the flow of data through the program.

Table 1
Mappings defined in the mapping model.

Model Element	Description
Component	In general, a component is mapped to a class. However, this mapping is also used to specify a method that describes the main entry point of the component, e.g., a method that executes the behavior of the component.
Port	This mapping is used to specify a method for writing to or reading from a component port. We therefore distinguish between IN-port mappings and OUT-port mappings. If an INOUT-port is used, both mappings have to be specified.
Data Source	This mapping is used to specify a method that returns a specific data type if a component is specified as a source for a data type.
Sanitizer	This mapping is used to specify a method that executes the sanitization of a data type.

Instead of generating full analyses, we use the information stored in assumptions and the mapping model to *configure* taint analyses provided by mature frameworks such as Boomerang [8, 9].

Since assumptions are always specified for a specific component, the analyses are restricted to the corresponding implementation for this component as well. In general, both the read-messages for all IN-ports of the component that receive a specific datatype and (if the component is a source) the source-method for data type are potential sources for the taint analysis. Similarly, all OUT-ports are potential sinks for the taint analysis. In the following, we describe how a taint analysis can be specified for each assumption based on our models.

We assume that the mapping model is fully specified and, therefore, provides methods for reading a data type from a IN-port, writing a data type to an OUT-port, sanitizing data types for each sanitizer, and for executing the component's behavior. The last method can be used as an entry-point for the code analyses. If not specified, all public accessible methods have to be considered as potential entry points, e.g., `public` methods in Java. Methods for ports and sanitizer are used to configure the taint analyses. Both methods for reading IN-ports of all ports that are capable of handling the data type to be analyzed, and a method if the component is a source for the data type are considered as sources for in taint analysis. Correspondingly, methods for OUT-ports are considered as sinks in the taint analysis. In the case of a flow assumption that explicitly defines a flow from one to another port, only methods for these two ports are considered. When generating the analyses, we can reduce the search space by considering the information of the component model. In particular, we only take methods for ports into account that are capable of handling the data types under investigation. For example, let us assume that the component of the card reader (cf. Listing 1) is connected to the cash desk. When analyzing the implementation of the cash desk on the flow of credit card information, it is sufficient to take the port of this connection as a source for the credit card information.

After executing the analyses, the result shows if the assumptions are correctly implemented in the given implementation. An advantage is that not all analyses have to be re-evaluated if the source code for a component changes but only the analyses that are relevant for this component. Also, the security engineer can use this information to either consider this fact in the security model, e.g., by adding additional assumptions to other components, or by contacting the developer of the components that do not comply with the assumptions.

3. Implementation

We implemented a prototype of our DSL and analyses using the Eclipse Modeling Framework (EMF). We chose to add a textual representation of the DSL using Xtext [10] and implemented a graphical editor using Sirius [11].

The source code for our implementation can be found on https://github.com/secure-software-engineering/cards

In the following, we describe all parts of our implementation shortly.

Textual and Graphical Editor The graphical editor for CARDS was implemented using Sirius. Figure 3 shows an example of the graphical editor. In addition, we provide a textual editor implemented using the Xtext framework. All changes made to the model in the graphical editor are also reflected on the underlying Xtext model. Hence, developers can switch at any time to the representation they prefer. Using the graphical editor, we can easily model systems or create representations for existing models. The diagram representation can be analyzed using Sirius' own tool to verify diagrams, which invokes our analyses, using EMF validation and are shown in the model and the Eclipse problems view.

Analyses The analysis explained in Section 2.2 is implemented as a basic depth-first search. We treat the model as a directed graph and recursively propagate data types, which a component is source for, over outgoing edges. Output of this analysis is a mapping from components to all paths through the model. The assumption analysis explained in Section 2.2 iterates through the paths determines the processed data per component. The

output of this analysis is a mapping of components to pairs of lists of paths and data types, which are received on these paths. To resolve restrictions, we check for each restriction, if data types of the defined restriction are illegally accessible at a component.

Mapping Model As explained in 2.3, we created a mapping model, which maps model parts to Java code to ease the generation of static code analyses. This mapping is implemented as a EMF model. Empty mappings for new model parts are automatically added to this model when using our graphical editor suite. Instead of providing an additional DSL for the mapping model, we provide a properties view for relevant parts of the model in our graphical editor, where mappings can be edited.

Generation of Glue Code Using the Xtend framework, we implemented a code generation, whose output can serve as glue code for Java implementations of a given model. Components are implemented as Java threads and all connections and mappings between component parts are implemented using the observer pattern. Communication is restricted to strings, but can be extended to arbitrary objects. Similar to our DSL, composite components handle the inter-component communication by instantiating connections. Additionally, all assumptions are added as documentation for the developer using Java annotations. Upon code generation, the mapping model is also created automatically.

Static Code Analyses Based on the concepts described in Section 2.3.2, we generate the configuration code for the static code analysis automatically using the Xtend framework. The generator takes the component model and the mapping model as input. All assumptions can be validated using taint analysis. Since we are focusing on Java code in our implementation, we decided to use the established analysis framework Boomerang [9, 8] for the specification and execution of the taint analyses. We generate the required taint analyses for each assumption. The generator can be adapted to any other framework that enables the specification and execution of taint analyses. This also allows one to use different languages for the implementation of the system's components.

4. Case Study

We evaluated CARDS using a case study based on CoCoME [12]. CoCoME is an established example for component modeling commonly used in research. The example system is a model of a store which is part of an enterprise. An enterprise consists of a server, client and several stores, each store consists of a server, client and several cash desks. A cash desk consists of a bar code scanner, a card reader, a cash box, a printer and a light display, all of which are connected to a cash desk pc, which also connects to a bank. Figure 3 shows the component model using our graphical editor. For our evaluation, we chose to base our model on CoCoME's first proposed use case, the sale. A sale is an interaction between a customer and a cashier. We model the complete cash desk, a bank and the store infrastructure. We adapted the data types provided in the reference implementation of CoCoME [13], as they are not part of the original definition. We used the case study as a proof of concept of CARDS itself. For our example, we defined a restriction that the credit card number and pin may only be accessed by the card reader, bank and cash desk pc. In the real world, the credit card number may be printed if partly replaced with asterisks, so a sanitization is a sensible approach.

Using the provided models of CoCoME, this restriction is not directly clear, as dataflows are not part of their modeling. With CARDS, we can already provide a formal restriction for this use case. Listing 2 shows the textual representation of this restriction. Upon validating the model, our analyses provide the developer with feedback that the current model violates the restriction because the credit card information may be accessed at every component, including the printer. To address this violation, we chose to define several dataflow assumptions for our model. Listing 3 shows a representation of the assumptions we made to resolve the violations. In particular, we assume that the credit card information will never be leaked to the light display, cash box and anything outside the cash desk component. Additionally, dataflows between `pcCardReaderPort` and `pcPrinterPort` of the cash desk pc component will be sanitized using the CCSanitizer. With these assumptions in place, the analysis does not show any violations for the restriction. Developers might find major security flaws in their architecture based on restriction violations, which may lead to architectural refactorings that resolve the violation. We used CARDS to generated a Java project for the cash desk application and implemented the behavior code for the relevant components based on the documentation of CoCoME. Also, the corresponding mapping model and the static code analyses were created automatically.

For the evaluation of the analyses, we created two versions of the implementation: one version violating the assumptions which should therefore lead to a report by the analysis, and one version that respects the dataflow assumptions, e.g. by preventing dataflows or using the desired sanitizer. The analyses were able to find the incorrect dataflows. However, it showed that in the current implementation false positives might get reported if one specifies different policies for data of the same port. To solve this problem, the developer needs to either adjust the implementation making sure that the data are filtered and correctly sanitized, or the result is fed back into the

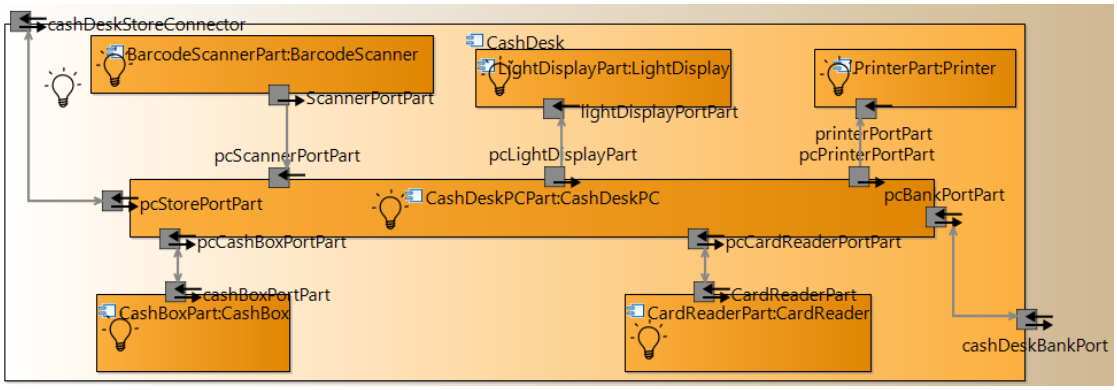

Figure 3: Component diagram based on the CoCoME case study.

component model where the security engineer can split the dataflows such that the flows are analyzed separately.

The evaluation showed one major advantage of the approach. When the source of one component changes, only the analyses for this component have to be re-evaluated instead of analyzing the whole source code again. For example, assuming that the implementation of the component CashDeskPC changes, only the analyses for this component have to be executed. If the implementation of other components changes, no re-evaluation is required. Especially for large-scale systems, this compositional approach can help to reduce the overall time for threat modeling and risk analysis.

5. Related Work

There are two major areas to which CARDS is related: Threat Modeling and Model-based security testing. Threat modeling because CARDS enables threat modeling and analyses based on the created threat model. Security testing since CARDS aims to automate validating the implemented security assumptions.

Threat Modeling For threat modeling, often dataflow diagram based approaches are applied because of the simplicity and technology-agnostic modeling [2]. Most prominent examples are the STRIDE approach [14] or LINDDUNN[15] for privacy-focused threat modeling. CARDS is related to these approaches since it also utilizes an architectural description of the system. However, in contrast, CARDS focuses on seamless threat modeling by combining threat modeling and analyses on the actual implementation. Currently, CARDS does support finding known threats automatically but we plan to implement this in future work.

Several approaches enhanced the use of data-flow diagrams to improve threat modeling and risk analysis. Extended dataflow diagrams Berger et al. present an approach using extended DFDs [16] which are a more formal version of classical dataflow diagrams. Since these DFDs allow for formal analyses and hierarchical system specification, it allows for more precise threat modeling. In contrast, we base our threat modeling approach on established modeling artifacts enabling the integration of our concepts into existing approaches. Peldszus et al. [17] providing an approach that aims at the connection from dataflow diagrams to source code and is therefore also highly related to our approach. This approach enables more precise threat modeling because the actual implementation is respected in the threat model. In contrast, CARDS focuses on a top-down approach enabling early analyses without a code-base.

Also, model-driven and model-based security grew to a large research area in the last years [18]. An overview of approaches in general can be found in the mapping study by Nguyen et al. [19]. Several approaches integrate security modeling into existing modeling approaches, e.g. SEED [5] or UMLsec [4]. SEED [5] is an approach that aims at building a bridge between embedded system experts and security experts. In SEED, security experts can define security solutions that can be used during the system design and to validate the system based on the integrated security solutions. In contrast, CARDS focuses on the definition of assumptions at design time and the validation on source code level instead of defining concrete security solutions that are integrated into the system design. UMLsec [4] provides a UML profile providing modeling concepts and analyses for security-relevant system properties. In contrast to UMLsec, CARDS focuses on the connection of design-time assumptions and the source code implementation, leaving model-driven concepts like concrete behavior modeling out.

Model-based Security Testing Following the classifications discussed in a survey by Felderer et. al [20], for security testing two principal approaches are distinguished in general: Testing to find vulnerabilities and unknown threats in the system and testing if the security mechanisms are implemented correctly [21]. The first category does not fit to CARDS since we are using threat modeling techniques to define security requirements and threats in the initial steps but CARDS does not contribute to finding new threats or vulnerabilities by itself.

Following Schieferdecker et al. [22], models that are used for model-based security testing can be categorized into three major categories: First, *Architectural and functional models* which "are concerned with system requirements regarding the general behavior and setup of a software-based system" [22]. Second, *Threat, fault and risk models* that "focus on what can go wrong" [22] and are used to determine potential threats, corresponding risk factors, and their relationships, e.g., STRIDE [1]. Third, *Weakness and vulnerabilities models* describing "the weakness and vulnerabilities itself" [22], e.g., models referring to CVE or CWE but also catalogs for generating threat lists like in the Microsoft Threat Modeling Tool [1]. CARDS provides a combination of the approaches of the first and second category because it utilizes architectural models for describing a secure system architecture but also concepts and analyses for reasoning about dataflow threats in the system. In contrast to existing approaches CARDS combines a light-weighted threat modeling approach on abstract design models with concrete analyses on the implemented system and, therefore, enables seamless threat modeling of a system. Providing vulnerability and attack catalogs or the integration of CVEs is currently not supported and left for future work.

6. Conclusion

Modern information systems require development techniques that ensure security-by-design. Especially, dataflows within a system are of high interest since data is often a sensitive asset of the system. The early creation of a threat model but also the seamless integration of the threat model into all development steps of the system are essential to this extent. In this paper, we have presented CARDS, a model-based threat modeling approach for dataflows in distributed systems. We discussed our concepts based on a generic component model. CARDS allows to formally specify security requirements for sensitive data of the system and to validate these requirements on architectural level by defining assumptions for the system's components that need to be fulfilled in the implementation. For this, we provide a DSL that allows defining both requirements and assumptions for a component-based system specification. Using this systematic approach helps designers identify required dataflow rules for the implementation at early development steps. These rules (assumptions) can be useful in different ways: On the one hand, when implementing a new system, they can be used as requirements for the later implementation. On the other hand, they can be used to validate if an already implemented system does comply with the security assumptions.

Furthermore, we provide a concept of how these assumptions can be expressed by static code analyses, allowing to automatically validate the assumptions on a given implementation. The advantage of this modular approach compared to approaches that validate security requirements is that assumptions are defined component-wise and, therefore, only the code for affected components has to be analyzed. This is especially important if the source code for only one component changes and the requirements has to be re-evaluated. Also, connecting a threat model on the architectural level with concrete analyses on the source code level helps feed back analysis results into the threat model. This simplifies reasoning about the effects of the analysis results.

We provide a prototypical implementation of CARDS containing a graphical and textual editor for component model and our DSL for describing assumptions and restrictions and evaluated our concepts based on a use case of the CoCoME case study. To ease the process of connecting threat model and code, we provide a generator to Java code that automatically creates a mapping model describing the connections from model elements to dedicated Java methods. For existing system implementations, the approach is currently limited in efficacy because the mapping model that connects the component model used for threat modeling and the source code has to be created manually. However, we see potential to automate this step in future work. We also plan to extend the approach by taking the kind and security level of data types and components into account when analyzing the model. This would enable the security engineers to apply concepts of DFD-threat modeling (like in STRIDE) on the component model and to search for required restrictions and corresponding assumptions automatically.

We see CARDS as a promising combination of light-weighted threat modeling and concrete security analyses on source code which can help system developers to create more secure large-scaled distributed systems.

References

[1] A. Shostack, Threat Modeling: Designing for Security, John Wiley and Sons, Indianapolis, USA, 2014.

[2] L. Sion, K. Yskout, D. Van Landuyt, A. van den Berghe, W. Joosen, Security threat modeling: Are data flow diagrams enough?, in: IEEE/ACM 42nd

International Conference on Software Engineering-Workshops, IEEE/ACM, 2020.

[3] P. H. Nguyen, M. Kramer, J. Klein, Y. Le, An extensive systematic review on the Model-Driven Development of secure systems, Information and Software Technology 68 (2015) 62–81. doi:10.1016/j.infsof.2015.08.006.

[4] J. Jürjens, Umlsec: Extending uml for secure systems development, in: J.-M. Jézéquel, H. Hussmann, S. Cook (Eds.), UML 2002 — The Unified Modeling Language, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 412–425.

[5] M. Vasilevskaya, L. A. Gunawan, S. Nadjm-Tehrani, P. Herrmann, Integrating security mechanisms into embedded systems by domain-specific modelling, Security and Communication Networks 7 (2014) 2815–2832.

[6] B. Kordy, L. Piètre-Cambacédès, P. Schweitzer, DAG-based attack and defense modeling: Don't miss the forest for the attack trees, Computer Science Review 13-14 (2014) 1–38. doi:10.1016/j.cosrev.2014.07.001. arXiv:arXiv:1303.7397v1.

[7] S. Peldszus, K. Tuma, D. Strüber, J. Jürjens, R. Scandariato, Secure Data-Flow Compliance Checks between Models and Code based on Automated Mappings, in: MODELS2019, i, 2019.

[8] J. Späth, K. Ali, E. Bodden, Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems, Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages 3 (2019) 48:1–48:29. doi:10.1145/3290361.

[9] J. Späth, L. N. Q. Do, K. Ali, E. Bodden, Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java, in: European Conference on Object-Oriented Programming (ECOOP), 2016.

[10] Xtext, https://www.eclipse.org/Xtext/, [Online; accessed Dec-2020].

[11] Sirius, https://www.eclipse.org/sirius/, [Online; accessed Dec-2020].

[12] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, et al., Cocome-the common component modeling example, in: The Common Component Modeling Example, Springer, 2008, pp. 16–53.

[13] Cocome - the common component modelling example webpage, https://cocome.org, [Online; accessed Dec-2020].

[14] A. Shostack, Threat modeling : designing for security, Indianapolis, Ind. : Wiley, 2014.

[15] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, W. Joosen, A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements, Requirements Engineering 16 (2011) 3–32.

[16] B. J. Berger, K. Sohr, R. Koschke, Automatically extracting threats from extended data flow diagrams, in: J. Caballero, E. Bodden, E. Athanasopoulos (Eds.), Engineering Secure Software and Systems, Springer International Publishing, Cham, 2016, pp. 56–71.

[17] S. Peldszus, D. Strüber, J. Jürjens, Model-based security analysis of feature-oriented software product lines, in: Proceedings of the 17th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, 2018, pp. 93–106.

[18] A. V. Uzunov, E. B. Fernandez, K. Falkner, Engineering security into distributed systems: A survey of methodologies., J. UCS 18 (2012) 2920–3006.

[19] P. H. Nguyen, S. Ali, T. Yue, Model-based security engineering for cyber-physical systems: A systematic mapping study, Information and Software Technology 83 (2017) 116–135.

[20] M. Felderer, P. Zech, R. Breu, M. Büchler, A. Pretschner, Model-based security testing: a taxonomy and systematic classification, Software Testing, Verification and Reliability 26 (2016) 119–148.

[21] G. Tian-yang, S. Yin-Sheng, F. You-yuan, Research on software security testing, World Academy of science, engineering and Technology 70 (2010) 647–651.

[22] I. Schieferdecker, J. Grossmann, M. A. Schneider, Model-based security testing, in: A. K. Petrenko, H. Schlingloff (Eds.), Proceedings 7th Workshop on Model-Based Testing, MBT 2012, Tallinn, Estonia, 25 March 2012, volume 80 of *EPTCS*, ETAPS, Tallinn, Estonia, 2012, pp. 1–12. doi:10.4204/EPTCS.80.1.