Secure Data-Flow Compliance Checks between Models and Code based on Automated Mappings

Sven Peldszus*, Katja Tuma†, Daniel Strüber†, Jan Jürjens*‡, Riccardo Scandariato†
*University of Koblenz-Landau, Koblenz, Germany
†University of Gothenburg and Chalmers University of Technology, Gothenburg, Sweden
‡Fraunhofer Institute for Software and Systems Engineering ISST, Dortmund, Germany
EMail: speldszus@uni-koblenz.de, katja.tuma@cse.gu.se, danstru@chalmers.se,
juerjens@uni-koblenz.de, riccardo.scandariato@cse.gu.se

***Abstract*—During the development of security-critical software, the system implementation must capture the security properties postulated by the architectural design. This paper presents an approach to support secure data-flow compliance checks between design models and code. To iteratively guide the developer in discovering such compliance violations we introduce *automated mappings*. These mappings are created by searching for correspondences between a design-level model (Security Data Flow Diagram) and an implementation-level model (Program Model). We limit the search space by considering name similarities between model elements and code elements as well as by the use of heuristic rules for matching data-flow structures. The main contributions of this paper are three-fold. First, the automated mappings support the designer in an early discovery of implementation absence, convergence, and divergence with respect to the planned software design. Second, the mappings also support the discovery of secure data-flow compliance violations in terms of illegal asset flows in the software implementation. Third, we present our implementation of the approach as a publicly available Eclipse plugin and its evaluation on five open source Java projects (including Eclipse secure storage).**

***Index Terms*—Security-by-design, Security compliance, Data Flow Diagram (DFD), Model-to-Model Transformation (M2M)**

I. Introduction

Security threats to software systems are a growing concern in many organizations, particularly due to the recent changes in legislation (GDPR) and upcoming security standards (ISO 21434). Therefore, one needs to consider security early in the design phase, when little is known about the system. In the start of the development process, requirements are collected and use cases are defined. According to the principle of *security by design* [1], [2], the system's assets and threats already have to be defined in this phase. The system architecture is then iteratively refined and finally implemented. Before any new functionality is released, it must be checked that every security assumption made in any of the phases is met. The state of the art for this check in practice are manual code reviews by security experts. Since such reviews are expensive and error-prone, they are only performed on selected code parts, leaving a large leeway for security threats [3], [4].

In the context of software architecture design, threat analysis techniques, like Microsoft's STRIDE [5], attack trees [6], CORAS [7], and threat patterns [8] aim to identify security threats to software systems. Threat analysis is very helpful to detect security threats early and plan countermeasures to mitigate them. Yet, empirical evidence shows that existing threat analysis techniques can be manually labor intensive [9] and lack in automation [10]. Furthermore, design-level models are seldom kept in sync with the implementation, potentially resulting in architectural erosion and technical debt [11].

Threat analysis is often performed on a graphical representation of the software architecture called *Data Flow Diagram* (DFD, [12], [13]). DFD-like models are extensively used in practice, e.g., in the automotive industry [14] and at Microsoft [5] as part of their STRIDE methodology. Still, the DFD notation is informal and lacks the ability to specify security properties, which is needed to reason about security threats at the design level. To support the detection of problematic information flows at the design level, previous work extends the DFD notation with security-relevant information [15] and security semantics [16]. However, the outcomes of such detection are of limited value if the implementation does not comply with the security properties described in the DFD model.

This work aims to support the discovery of secure data-flow compliance violations between the designed and the implemented security properties in a software system. We present a technique that automatically establishes mappings between a design-level model enriched with security-relevant information (*Security Data Flow Diagram*, short: *SecDFD*) and an implementation-level model (*Program Model*). These mappings can be used to discover compliance violations of secure data-flow properties (typically, data confidentiality and data integrity properties) as follows: The designed data flow is captured in the SecDFD model. The actual data flow is obtained from implementation-level data-flow analysis tools (discussed later). These tools typically require sophisticated meta-data (e.g. an explicit tagging of security-critical data and functions) as input, which can be obtained from our mappings. Finally, our mappings also support the designer in an early discovery of implementation absence, convergence, and divergence with respect to the planned software design, including its security properties. We make the following contributions:

(i) We present an automated technique for establishing mappings between SecDFDs and program models, thereby supporting the discovery of secure data-flow compliance violations. The key idea of our technique is

twofold. First, we define a mapping between SecDFD and program-model element types, constraining how elements of a concrete system can be mapped to each other. Second, we combine similarity-based matching of element names with structural heuristics (based on data-flow properties) to automatically derive suggested mappings between the SecDFD and the program model.

(ii) We present an incremental methodology, in which the user is involved to successively discover new mappings and eventually derive an adequate mapping.

(iii) We present our implementation of the approach as a publicly available Eclipse plugin and the evaluation of its accuracy on five open source Java projects (including Eclipse secure storage [17]).

The rest of the paper is organized as follows. Section II describes the background, including the design and implementation-level models. Sections III introduces the approach, and Section IV describes the evaluation results and their discussion. The related work is presented in Section V, and the limitations of this work are discussed Section IV-C. Section VI presents the concluding remarks.

II. Background

This section describes the background on design-level models, architectural compliance checks, and implementation-level models. We consider the Eclipse secure storage [17] to illustrate the models considered in this work. The secure storage allows plugins to store and access secret data. This functionality is used, for example, by the Git extension of Eclipse to store user names and passwords [18].

A. Design-level model

Various different model kinds are used for specifying a system's data processing procedures at design time. Apart from DFDs, frequently used notations are activity diagrams [19] and business process models (BPMN, [20]). Our rationale for focusing on DFDs is twofold: First, they are widely applied in practice, specifically, in the automotive industry [14] and at Microsoft [5] as part of their STRIDE methodology. Second, they represent an essential set of concepts necessary for data-flow analysis (processes and data flows between them), which be can mapped exhaustively to activity diagrams and business processes, rendering our mapping generation technique also applicable to these model kinds. Thus, while we introduce our technique for DFDs, it can be applied to a broad range of modeling languages supporting data flow modeling. In what follows, we introduce DFDs and a security extension which allows to include security-relevant information in DFD models [21], [15], as we require for checking consistency between planned security and implemented security properties.

a) DFDs: A Data Flow Diagram (DFD) is a graphical representation of the software architecture and the information it handles [5]. It represents how the information enters, leaves, and traverses the system. The DFD consists of processes (active entities), external entities (e.g., 3rd parties), data stores (where information rests), data flows (carrying the exchanged information), and trust boundaries (signaling trust levels). Fig. 1 depicts a DFD for the Eclipse secure storage. The plugin attempts to access a secret by sending a request including path information of where to look for the secret (e.g., a password request for a user name of a Git account). The secure storage queries an internal tree-like data structure to find the corresponding node containing the secret. Next, the cache is queried for the secret value, which can be in clear text (i.e., *secret* on flow 5 in Fig. 1) or encrypted (i.e., *encr data* on flow 6). If the value is in clear text, the secret is sent to the plugin. In case of an encrypted value, a decrypt operation either fetches the root password from the operating system or prompts the user to provide it. Upon a successful decryption, the secret is sent to the plugin (flow 9 in Fig. 1). Though useful for performing architectural threat analysis [22], we do not use trust boundaries in our work.

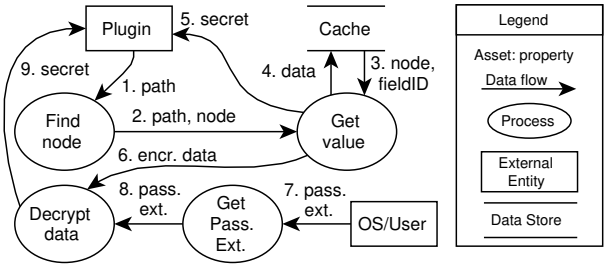

Fig. 1: A DFD for Eclipse secure storage.

b) Security Extension: To capture security properties at the architectural level, we propose to use Security Data Flow Diagram (SecDFD, [16]). SecDFD is an intuitive graphical notation that enriches DFD with security concepts to enable a formally grounded information-flow analysis, focusing on the confidentiality and integrity of information assets. Specifically, assets can be tagged as *high* or *low* confidentiality. Process nodes can tagged with security contracts that define how the security properties of assets change upon exiting the node. In our example, if a plugin requires permissions that are cached encrypted, the user must provide a *password* when prompted (c.f. *pass. ext.* in Fig. 1). Since the password is confidential, it should not be leaked to other plugins running in the environment. To specify this behavior, a designer would add appropriate tags to the DFD shown in Fig. 1. The specified security properties can be propagated from the SecDFD to the code using the mappings created by our approach. They can then be used as input for code-level analysis tools, thus enabling compliance checks between planned and implemented security properties (see Sect. III-E). For the concrete syntax and semantics of SecDFD we refer the reader to [16].

B. GRaViTY Program Model

To create a mapping between SecDFDs and their concrete implementation we need an easy to analyze representation of the source code. Representations such as abstract syntax trees (AST) contain every detail from the implementation, which makes it hard to analyze for security purposes. Many details

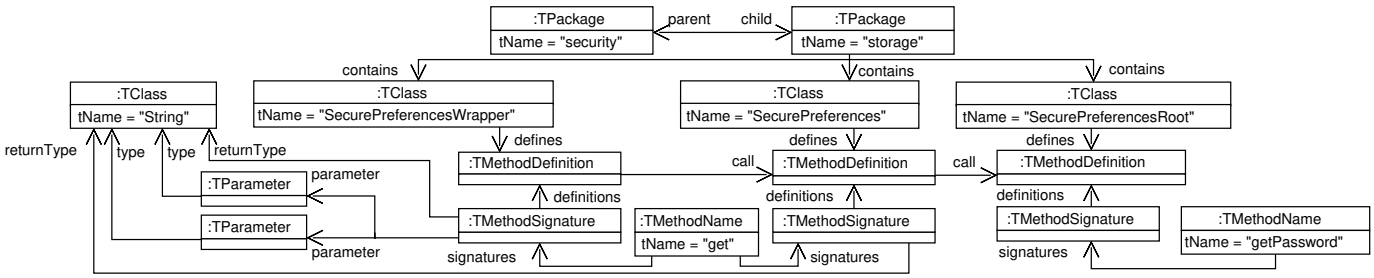

Fig. 2: Excerpt from the Program Model of the Eclipse secure storage (shown as UML object diagram).

about the implementation are not required for our approach. At the same time, important information is not always directly accessible. For example, in the source code files or an AST accesses of fields are not directly visible as access edges between the source and the accessed field, but are access statements within the source to some field with a given name. For our approach, it is only important to know that there is an access to a specific field from some source, but we don't need to know every detail about the circumstances of this access. The program model creates a more suitable abstraction for security analysis and allows easy queries, which were very useful for our approach.

A program representation which has been designed to make information about the structure of a program easily accessible and to abstract not relevant information from the statement level has been proposed by Peldszus et al. with the *GRaViTY*-framework [23], [24], [25]. The *GRaViTY*-framework has been used for the evaluation and execution of refactorings [24], for the detection of anti-patterns [26], as well as for the automated design optimization of Java applications [27].

Fig. 2 shows an excerpt of the program model created by the *GRaViTY*-framework for the Eclipse Secure Storage example. The figure shows two method calls. The first call is from the method *get(String, String)*, defined in the class *SecurePreferencesWrapper*, to the method *get(String, String, SecurePreferenesContainer)* of the class *SecurePreferences*. The second call is from the called method of the first call to the method *getPassword(String, IPreferencesContainer, boolean)* which is defined in the class *SecurePreferencesRoot*.

On top of the figure we can see the package structure of the program. All packages without a parent can be taken as entry point for a search. Additionally, it is possible to iterate over all types directly. In the figure the types, in this case all are classes, are shown in the second row, each with a reference to the members defined within the type. In the figure for the three classes *SecurePreferencesWrapper*, *SecurePreferences*, and *SecurePreferencesRoot* only a single method each is shown. Methods are represented by an triple of method name, method signature and method definition. This allows an efficient search for specific methods, starting with the method name, going over signatures to concrete definitions for them. Method signatures have parameters which have a reference to the type representing the parameters type and a reference to the return type. In the program model excerpt only the parameters of the signature *get(String, String):String* are shown.

A benefit for our mapping from SecDFD to Java implementations is the possibility of an iterative search, starting only with little knowledge about the searched elements–e.g., a method name. The program model allows to start a search with such little information and to find more concrete elements by considering more information like method parameters without iterating over all method definitions defined in the source code. This makes the searches more efficient and easier.

C. Compliance

Identifying the differences and equivalences between the planned and the implemented software architecture is the goal of software architecture compliance checking. The compliance checks can be based on a static set of rules [28], dynamic monitoring of a running system [29], or a hybrid of both [11]. In our work, we statically check the compliance of design-level models to implementation-level models. Running compliance checks reveals the relations between a set of components of the first (design-level) model and a set of components of the second (implementation-level) model. As outcome, three different types of relations can be discovered.

a) Convergence: The compliance checks reveal an allowed relation between the implemented components. Convergence indicates that the implementation is compliant with the planned architecture. In this work, convergence means that the user has accepted a suggested mapping or has manually defined a mapping.

b) Divergence: The compliance checks reveal a relation between the implemented components that is not allowed. In other words, the implementation diverges and is therefore not compliant to the planned architecture. In this work, divergence means that there are flows of assets in the implementation which haven't been defined in a DFD. We look for elements that relate to existing mappings to find the relative parts of the implementation.

c) Absence: The compliance checks reveal a relation between design-level components that were not implemented. Absence indicates that the source code is not compliant with the planned architecture due to a missing implementation. In this work, divergence means that the user finished using our approach, but there are still design-level elements that have not been mapped.

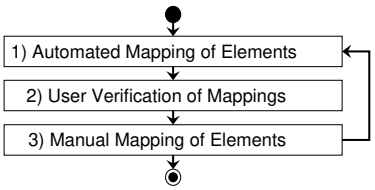

Fig. 3: Semi-automated Mapping of Implementations to DFDs

III. Enabling Compliance Checks with Automated Mapping Generation

Assuming a correct DFD, the way it is implemented can vary depending on concrete design (e.g., architectural patterns) and implementation specific decisions (e.g., programming language). Therefore, a full automatic generation of a correct and complete mapping between DFDs and code is not feasible. Yet, a manual specification of the same mapping is inefficient and error-prone. To this tend, we propose an iterative methodology for interactively guiding the user in finding an adequate mapping by combining automated mappings with user decisions as shown in Fig. 3. In step 1, mappings between DFD elements and implementation elements are calculated using a heuristic technique. In step 2, these mappings are presented to the user and manually checked by her. In step 3, the user can manually map additional elements. Afterwards the automated mapping is executed again, benefiting from the user input. The process terminates when the user cannot find any additional mapping or finds a violation.

In this section we describe the steps of our methodology, including the automated technique, in detail. In addition, we explain the use of these mappings in compliance checks. In Sect. III-A, as a basis for mappings in concrete systems, we define a mapping of DFD to element types that may correspond to each other. In Sect. III-B, we show how our automated technique in step 1 establishes concrete mappings between DFDs and their implementations by using a naming- and structure-based heuristics. In Sect. III-C and III-D, we explain the interactive steps 2 and 3 of our techniques. In Sect. III-E and III-E2, we argue how the created mappings can be used for checking general compliance and security compliance of the implementation with the DFD.

A. Corresponding Elements

As a prerequisite for mapping DFD elements to code elements, first we have to define which DFD element can correspond with which code elements.

- **Assets → types**: The assets in a DFD are the elements holding critical data. On the level of implementation, data is usually stored in fields, processed using variables and transmitted using parameters and return values. A single asset can be stored in many different locations at the same time which makes it infeasible to map an asset to every single location. The only property of an asset which only changes rarely in programs, written in an object-oriented languages, is the type of the asset.

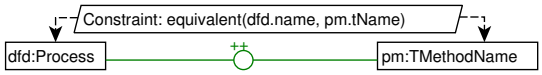

Fig. 4: Rule describing the name matching for methods

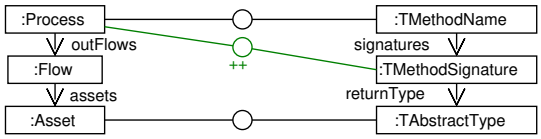

Fig. 5: Rule for extending name matches based on return types

- **Data stores → types & methods**: If we think about data stores like the cache in the example DFD, it is quite obvious that this could be a field in some class. But it could also be implemented by an operation which, e.g., requests the cached values from an external server by creating HTTP requests. The thing in common between these two variants is the type used to store the data in. The field has a type which provides getters and setters for using the data store, and the method used to get data from a remote server is implemented in a type. Therefore, we map data stores to types as well as to the methods used for accessing the stored data.
- **Processes → method(-names)**: Processes in DFDs describe functionalities which process data, like methods in implementations do. Obviously, these two elements correspond with each other. While a concrete method definition in an implementation contains all details describing the functionality of this method, the processes only have a name describing the functionality. We assume that a developer implementing a process will chose a similar name for the methods implementing this process. This leads us to a correspondence between the names of processes and the names of methods.
- **Processes + Assets → method parameters**: Between processes in a DFD, data can be exchanged using flows, where the exchanged data are represented by assets on the flows. In the methods implementing these processes the same data have to be exchanged. Data between methods in implementations are usually exchanged using parameters and return values. Therefore, we can combine the name mappings between processes and methods with the assets flowing into and out of a process to method parameters giving us the according method signatures.

B. Semi-automated Mapping

In what follows we discuss the steps of our automatic generation of mappings in detail.

1) Automated Mapping of Elements: The automated generation of mappings is based on name matchings and structural heuristics, which are sequentially executed and complement each other. For illustration, we formalize two of our mappings using graph rules, using a notation inspired by algebraic graph transformation [30] (explained below). The other mappings can be formalized in a similar way.

a) Name matching: First, the names of elements from a DFD are mapped to the according names in the implementation. Asset and data store names are mapped to the names of types and process names are mapped to the names of methods. Fig. 4 shows a rule for mapping processes from a DFD to method names from a program model. A correspondence (visualized as circle connecting the corresponding elements) between a process and a method name is created (denoted by ++) if the constraint at the top of the rule holds. In this case the names of the two elements on the left and right of the rule have to be equivalent. The precise definition of this equivalence is described in what follows.

Names, both in a DFD and in a Java implementation, are usually build by concatenating multiple words. For example, a Java method name *getPassword* consists of the word *get* and *password*. These words can vary slightly in the names of the corresponding DFD processes (e.g., in plural form, *passwords* instead of *password*). In addition the style of word concatenation can differ. In Java usually the camel case (*getPassword*) is used, whereas in DFDs this is not a prescribed style, so underscores may also be used (*Get_Passwords*).

To deal with these issues, first, we split the strings at frequently used delimiters and upper-case characters. This gives us for the example the sets of words *[get, Password]* and [Get, Passwords]. Then we compare the lower-case versions of the words with each other using a fuzzy compare based on the Levenshtein distance [31]. The Levenshtein distance is a measure of the minimal amount of characters which have to be removed, added or flipped to change one word into the other one. For the given example this distance is zero and one as the first word is already identical and only the character *s* has to be added to change *password* into *passwords*. We accept different distances between words for considering them as identical according to the length of the words to be compared.

Finally, a DFD process is usually implemented in multiple methods, typically having slightly more concrete names. For example besides the method *getPassword*, there might also be an additional method *internalGetPasswod* involved in the implementation of the process *Get_Passwords*. But the DFD process name might also contain additional information – e.g. the process *get_Passwords_External* of the DFD in Fig. 1. To address this challenge, we compare all words from the two names with each other and count the similar words. If this number reaches an threshold of more than half the number of the average words of the compared names, we consider the names sufficiently equal.

For the example DFD in Fig. 1 and the program model excerpt in Fig. 2 we get a name match between the *Get_Value* process and the two method names *get* and *getPassword* as well as a match between the process *Get_Passwords_External* and the method name *getPassword*. While two of this matches are expected, the match between *Get_Value* and *getPassword* is unexpected and should be dropped in the following steps.

b) Extending Name Matches to Method Signatures: For every method name, multiple signatures may exist. Even if our name matches were always perfectly correct, this would not imply that all signatures with this name are the ones corresponding to the according process. For example, besides the relevant signature *getPassword(String, IPreferencesContainer, boolean):PasswordExt*, there might be a second signature *getPassword():char[]* defined in the Java standard library which is never used in the implementation. To identify the actually relevant signatures, we use data-flow information about assets flowing into and out of a process. Information flowing into a process has to be passed to the implementation of the process, for example, as a parameter value. Likewise, information leaving a process can leave it over return values and parameters. Accordingly, we can use the mapped assets to identify relevant signatures. For every signature, we count how many mapped assets are compatible with the parameters and return types of the existing signatures. If we have at least one match we consider this signature for further mappings.

A rule for extending a process mapping based on an asset flowing out of a process is shown in Fig. 5. On top of the rule we can see an existing mapping between a process and a method name, as e.g. created by the rule shown in Fig. 4. A mapping to one of the signatures having this name is created if there is an mapping between an asset flowing out of the process and a type which is the return type of the signature.

If we look at the return type of the signature *get(String, String):String* and assume that the *secret* asset from Fig. 1 has been mapped to the class *java.lang.String* we'll accept this signature as corresponding with the process *Get_Value*. The other method name corresponding with this process was *getPassword*. The return type of this method signature is *PasswordExt* and also no parameter type is matching to an asset. Accordingly, we don't create a correspondence.

c) Finding Implementations of Signatures: The last step is to find concrete implementations of a signature corresponding with the process. For every signature there might be several concrete implementations, all of which do not necessarily correspond to the process. We make use of the flows between different processes to find the concrete definitions.

If there is a flow from one process to another, this does not only mean that there has to be a signature which has the capability to return or receive the according asset. There also has to be a definition of this signature which is called from a definition in the other process. Therefore, we search for two kinds of data flows between the concrete definitions of the signatures found before.

1) Parameters passed by a call from the source of a flow to to the target of the flow.
2) Return values returned along a call from the target of a flow to the source of the flow.

The flow between two such definitions is not necessarily a single direct call between the two definitions. There can also be multiple definitions in between forwarding data. For example we can see in Fig. 2 a call between the methods *get(String, String, SecurePreferenesContainer):String* and *getPassword(String, IPreferencesContainer, boolean):PasswordExt* but in the DFD in Fig. 1 there is no flow between the processes *Get_Value* and

Get_Passwords_External, they have been mapped to. In the implementation the *get* method forwards the return value of *getPassword* to a call of method *decrypt* which has been mapped to the process *Decrypt_data*. Matching this intermediate to one of the two involved processes is non-trivial. However, if we found such a flow, we can definitely assume that we found two definitions implementing at least parts of the two processes.

The intermediate definitions can be partly mapped to one of the two processes by considering the internal coupling in a process. For every pair of signatures mapped to the same process, we look for pairs of definitions calling each other. For example, this is the case for the definition of the signature *internalGetPassword*, which is called by *getPassword(String, IPreferencesContainer, boolean):PasswordExt*.

d) Cleanup: After matching assets and processes we have to decide which matches are most likely to be correct and, therefore, should be presented to the user. For that reason, we introduce a certainty score for our mappings. This score is calculated with respect to the quality of the underlying name matching as well as the coupling of matched elements with other matched elements. For every DFD element we only present mappings whose score is higher or equal to the median score of all mappings for this element.

The mappings sorted out in this step are not presented to the user, but may be discovered later again in the interactive process – based on future matches, which might have a coupling to the elements that are now discarded.

C. User Verification of Mappings

The mappings created in the previous step are now presented to and verified by the user. For every asset-, data store-type and process-definition mapping the user can preform three actions:

1) **Accept:** The user can accept the mapping. From then, the mapping cannot be discarded by the optimization step of the automated mapping approach anymore, and all mappings coupled to this mapping obtain a higher certainty score.
2) **Reject:** The user can reject the mapping. From then, this mapping is never presented to the user again and it is not considered anymore for extending it to other mappings. All other mappings to which the rejected mapping has been extended will be removed, too, but might be presented to the user again.
3) **Tolerate:** The user can choose to ignore some suggested mappings. Mappings that are not explicitly accepted or rejected are suggested again and can be re-assessed in future iterations.

Mappings accepted or rejected by the user allow the heuristic to automatically discard related mappings that have only been found by following up the rejected mapping. This is how the search space is reduced in the next automated iteration. Conversely, manually accepting mappings can lead to the score of related mappings being increased and, for this reason, allow to propose new mappings which haven not been considered as correct ones before. Anyhow, a limitation of our heuristic is that they cannot detect mappings which are outside of the search space created by the initial name mappings. We are overcoming this limitation in our approach by including user feedback as described in what follows.

D. Manual Mapping of Elements

To increase the search space, an additional user step is conducted after the user manually verified the automatically created mappings (or at least a part of them). In this step, the user has to add at least one new mapping to give additional input to the automated mapping algorithm. The selection of this manually mapped element can have a large impact on the efficiency of the following automated steps.

E. Compliance Checks

The created mapping can be used to perform different kinds of compliance checks. At first it can be checked if the implementation corresponds with the specification in the DFD. Afterwards the mappings can be used to perform more sophisticated security analyses on the code using security information from the SecDFDs.

1) Compliance of Models and Code: The correspondence checks take place while the mappings are created. Using the proposed approach, we check for the three kinds of correspondences introduced in section II-C:

a) Convergence: All DFD elements which have been mapped to implementation elements and have not been rejected are allowed to be mapped. Following the definition of convergence, the convergences between the DFDs and the code are described by the set of all allowed mappings.

b) Divergence: Elements present in the code, but not specified in the DFD represent a divergence between the DFD and code. To help the user discovering divergences, it is possible to show all flows from members mapped to one process to other members not mapped to this process. If the target of such a flow has not been mapped to any process, there seems to be a divergence. But, a divergence also arises if there is a flow between two processes in the code that has not been specified on the DFD. If an critical asset is communicated along such a flow this is not only a divergence from the intended design but a security violation.

c) Absence: If we are neither able to map a DFD element to the code automatically and the user is not able to map the same element when asked, we discover an absence of specified functionality in the code. Assuming correctness of DFD models, we only have to consider this one direction of absence (concerning the opposite direction, see *divergence*).

Using these checks, a developer or code reviewer can detect convergences, divergences and absences between an DFD and the implementation at hand. However, regarding security, these checks are not precise enough: They might not reveal flows of confidential assets into parts of the program that are not supposed to take place – e.g., if a developer uses a full representation of an object, instead of a stripped one. To this end, we can perform more sophisticated security checks, as described in what follows.

2) Security Compliance Checks: The created mappings cannot only be used for compliance checks of SecDFDs and code, in terms of convergence, divergence and absence, but they can also be used for checking the security of the code. For this purpose multiple checks have been developed. A first example are security metrics [32], [33] which, e.g., calculate relations between security-critical and non-security-critical properties and allow an coarse estimation of the systems security level. Additionally, there are sophisticated path and flow analyses which can be used to detect concrete security violations [34], [35]. Such metrics and analyses are not directly applicable to the source code of the subjects which should be analyzed, since they require an explicit specification of sources and sinks of security-relevant data.

In the SecDFDs this information is explicitly given and can be fed into analysis tools as described before. Every asset in a SecDFD is labeled with the security properties of confidentiality and integrity. Using the mapping created by our approach we can identify the sources and sinks of the assets in the concrete implementations. For instance, the asset *secret* read by the *plugin* in Fig. 1 is read from the *cache* which corresponds with a field of the type `Map` in the class `SecurePreferences`. The correspondence mappings make it possible to identify program locations where proof obligations should be injected. Specifically, the Java Modeling Language (JML) enables annotating verifiable conditions for method implementations. Existing projects provide tools (i.e., KeY [36]) for statically verifying a correct behavior with respect to such proof obligations.

After the release of the system, the security information can be used to monitor the system at run-time. For example JBlare can be used to monitor sensitive information flows [37]. As security information for internal parts of a program is usually unavailable, JBlare can currently only be used to check IO calls, but not if a specific piece of data is allowed to enter some part of the program. The tracing of the security annotations to the code can be used to allow JBlare to monitor sensitive data flows within the program.

IV. Implementation and Evaluation

To evaluate our approach we implemented a tool prototype on which we performed the evaluation. In this section, first, we introduce this prototype and how it can be used by a developer or reviewer. Afterwards, we describe the performed evaluation and its threats to validity.

A. Implementation

The approach is implemented and packaged as a publicly available Eclipse plugin [38]. The implementation leverages an existing implementation for modeling SecDFDs with an Xtext DSL with editor support [16]. We use an existing plugin for generating the program model from Java source code [25].

Fig. 6 shows a screenshot of the user interface in Eclipse. On the left hand side of the figure, users can see the Package Explorer. The bottom windows are used for displaying and defining the mappings. The top two windows are used for displaying the source code (left) and the SecDFD (right). The target audience of the tool are software developers (or code reviewers) with training in the principles of software architecture. After the installation of the required packages, the program is started as a running Eclipse instance.

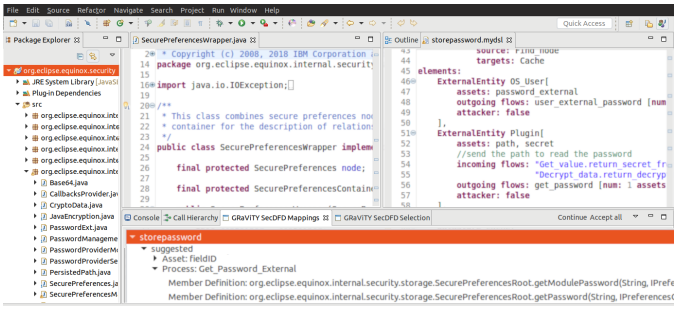

Fig. 6: Screenshot of the UI in Eclipse

The developers first manually create one or several SecDFDs for representing the high-level architecture of a Java project (c.f., top right window in Fig. 6). Next, using context menu entries, the developers trigger the automated generation of a program model from the source code, and start the first iteration of the semi-automated process for mapping the SecDFD elements to source code elements (see Sect. III).

At the start of each iteration, the developers are shown a list of suggested mappings (c.f., bottom window in Fig. 6). Since one SecDFD element is usually mapped to several program elements, the results are grouped by the SecDFD elements. For each SecDFD element, the list of mapped program model elements is shown, each with its path in the source code. The developers can interact with the tool by accepting, rejecting, and manually defining mappings. A suggested mapping is accepted or rejected with a right-click on the entry and selecting *accept* or *reject*, respectively. Once a mapping is accepted, corresponding in-line markers are created on the SecDFD and in the source code. Double-clicking a mapping will open the correct source file and navigate to the correct line in the file. Accepted mappings can always be rejected. If all the suggested mappings are correct, the developers can select *accept all*. Rejected mappings will never be suggested again. Manual definition works by right-clicking and selecting *Map Selection to SecDFD* on source code elements. At the end of the iteration, developers can either stop or select *continue* to trigger a new search refining the present mapping.

B. Evaluation

In an experiment we applied our approach to five open source projects to evaluate the performance of our implementation. In what follows, we briefly describe the design of the experiment, the projects, and the results.

a) Design of study: In our evaluation, we investigated the correctness of the automatically generated mappings. To this end, we set up an experiment to compare a ground truth of manually created mappings with the generated mappings for each of the five considered projects. The iterative approach

TABLE I: Projects considered in the evaluation

	source code			DFD
project	lloc	classes	methods	elements
jpetstore	1,221	17	277	47
ATM simulation	2,290	57	225	85
Eclipse secure storage	2,900	39	330	41
CoCoME	4,786	120	512	44
iTrust	28,133	423	3,691	31

involves the user to guide the generation of mappings in the desired direction. As per this design choice, we intentionally investigate the correctness of the automated mappings and the impact of the user separately. Consequently, the evaluation aims to answer the following research questions.

RQ1. *What is the correctness of the automated mappings generated by the plugin?* We measured correctness in terms of *precision* and *recall* (dependent variables). Conventionally, precision $(TP/(TP + FP))$ is measured as a ratio between the true positives (i.e., correct mappings) and all generated mappings (including the false mappings). A true positive TP is a correct mapping between the source code and the SecDFD element which is listed in the ground truth. A false positive FP is a mapping between the source code and SecDFD element that is not listed in the ground truth. Recall $(TP/(TP + FN))$ is measured as a ratio between the true positives and all correct mappings (including the overlooked mappings). A false negative FN is a mapping between the source code and the SecDFD element which is present in the ground truth, but has not been identified.

RQ2. *What is the impact of the user on the correctness of mappings?* The implementation automatically derives trivial mappings from the user defined mappings, raising the recall before a new iteration starts. Therefore, the impact of the user defined mappings is measured as the difference in recall before, and after the added mappings.

b) Evaluation subjects: Table I depicts the characteristics of five open source Java projects used in the evaluation.

*Jpetstore [39].*This is a web application built on top of MyBatis 3, Spring and the Stripes Framework. This is an example with very few classes, implementing the basic functionalities of a web store. In principle, the users are able to create their accounts, browse, and order goods online. Jpetstore has been designed as minimal demonstration application for MyBatis, which should have a good design and documentation. The developers tried to strictly follow the MVC pattern.

ATM simulation [40]. This is an implementation of a simulation for an ATM machine developed for academic purposes. The ATM simulation implements the main procedure of a control system. Upon start-up a new session is initiated, and the users are able to insert their card and PIN number. The session continues upon a correct PIN entry, and provides the users with the option of a withdrawal, deposit, balance inquiry, and money transfer. After a completion of desired transactions, the ATM returns the card and optionally prints the receipt.

Eclipse secure storage [17]. As described in Section II, Eclipse secure storage is used for ensuring secure storage and management of sensitive data within the developer's Eclipse workspace. The secure storage allows for plugins to authenticate and have controlled access to workspace resources.

CoCoME [41]. CoCoMe is a platform for collaborative empirical research on information system evolution [42]. This platform helps engineers manage different aspect of software evolution, such as the system life-cycle, versioning artifacts, and comprehensive evolution scenarios. The implemented system is a cash register.

iTrust [43]. The iTrust example is a web application for a hospital which allows the hospital's staff to manage medical records of patients, based on 55 use cases. The example originally stems from a course project, has been maintained by the Realsearch research group at North Carolina State University, and was used as an evaluation example in research papers before [44]. Detailed requirements describing different activities are available online [43]. However, the available requirements and use cases mostly describe very simple tasks and only a few of them are realized in the implementation.

c) Execution: The experiment was executed by the first and second author. The authors worked on the projects individually and compared their results at each step. First, the authors created the SecDFDs for all five projects models manually. To this aim, the authors inspected all available documentation (including the source code) and reverse engineered a high-level architecture. Second, a ground truth was created for each SecDFD by following the execution of the modeled scenarios and manually mapping the executed methods and transferred data to the processes and assets of the according step. The ground truth is a JSON file with a list of correspondence mappings between the elements of the SecDFD and a uniquely identifiable location of the source code element. Third, the implemented plugin was used to find the automated mappings in several iterations. Each iteration included accepting, rejecting the automated mappings, and defining mappings manually by highlighting elements in the source code and specifying the corresponding SecDFD elements. After each iteration the precision and recall of the automated mappings were logged.

d) Results: This study shows promising results for guiding the user in the discovery of compliance violations. In particular, Table II shows measurements of high precision and recall only after a few iterations for realistic Java projects. Each iteration consists of an automated, and a manual (user input) phase. We present the precision and recall for the automatically suggested mappings in each iteration. We also depict the amount of manually accepted, user defined, the sum of all accepted and user defined, rejected mappings, and the impact of the user defined mappings on recall (in that order). Notice that the later iterations make use of the manually defined mappings.

RQ1. We start by reporting the correctness of the automated mappings in the first iteration. The average precision of the first iteration is 50.5%. On average, the recall of the first iteration is 69.8%. Yet, both the precision and the recall increase after the first iteration. On average, the final precision

TABLE II: Results of the mapping after each iteration

		automated		manual		
project	it.	precision[%]	recall[%]	accept+u ($\sum$)	reject	recall[%](Δ)
jpetstore	1	56.1	51.1	23 + 3 (26)	18	57.8 (+6.7)
	2	96.4	60.0	1 + 3 (30)	1	66.7 (+6.7)
	3	96.8	66.7	0 + 5 (35)	1	77.8 (+11.1)
	4	97.4	82.2	2 + 3 (40)	1	88.9 (+6.7)
	5	100	93.3	2 + 3 (**45**)	0	**100** (+6.7)
ATM simulation	1	72.0	40.0	18 + 3 (21)	7	46.7 (+6.7)
	2	67.6	51.1	2 + 5 (28)	11	62.2 (+11.1)
	3	70.5	68.9	3 + 5 (36)	11	80.0 (+11.1)
	4	76.6	80	0 + 4 (40)	13	88.9 (+8.9)
	5	95.5	93.3	2 + 3 (**45**)	2	**100** (+6.7)
Eclipse sec. storage	1	73.0	90.5	40 + 1 (41)	14	92.9 (+2.4)
	2	67.7	**100**	1 + 0 (**42**)	12	—
CoCoME	1	27.9	77.3	17 + 1 (18)	44	81.8 (+4.5)
	2	86.4	90.5	1 + 1 (20)	2	90.9 (+0.4)
	3	90.9	83.3	0 + 2 (**22**)	4	**100** (+16.7)
iTrust	1	23.5	80.0	8 + 1 (9)	26	90.0 (+10.0)
	2	81.8	90.0	0 + 1 (**10**)	2	**100** (+10.0)

and recall of the automated phase are very good (87.2% and 92%, respectively).

The average difference between the recall of the second iteration and the the user-impacted recall of the first iteration (last column in Table II) is 4.5%. This means that on average, the automated search was able to increase the recall between the first and second iteration by 4.5%. On the other hand, the average difference between the user-impacted recall of the second iteration and the recall of the third iteration is minimal. This means that, the automated search was not able to increase the recall significantly between the second and third iteration.

RQ2. On average, the user accepted less (7) mappings then they rejected (9.6), and defined only 2.6 mappings manually. However, in three cases (jpetstore, ATM simulation, Eclipse Secure Storage) the user accepted more mappings then rejected. This means that the user could quickly scan the suggested mappings and eliminate the ones that are obviously wrong. Overall, adding a few mappings manually resulted in a more fruitful next iteration. For instance, adding three mappings manually in the first iteration of evaluating the ATM simulation resulted in two new correct mappings (see accepted mappings of the second iteration).

On average, the user impact on the recall was an increase of 7.9%. This means that the users were indeed able to guide the discovery of compliance violations. Further, the users had a larger impact on increasing the recall in later iterations compared to the automated search (7.9% vs 4.5%). Notice, that on average 75% of all correct mappings (TP) are suggested to the user and do not have to be manually defined.

e) Additional observations: While we where executing the evaluation we made different observations which are not directly covered by our research questions, but give further proof for the effectiveness of our approach.

1) All DFDs where created based on the available documentation. At executing the evaluation on the ATM simulation we recognized an absence between the created DFD and the implementation. Further investigations revealed that there is really an absence between the documentation of the ATM simulation and its implementation.
2) At studying the different examples from our evaluation we noticed big differences between the different implementations. The more realistic or real examples (Eclipse secure storage, CoCoME and iTrust) have a much better structured source code than the other two more artificial examples. While in the realistic examples functionalities are implemented in multiple methods, in the artificial examples single methods realize multiple functionalities. These differences are one of the reasons why our technique performed better on the realistic, larger examples. An hypothesis to be studied in the future is that writing the code with the DFD in mind can help structure it better and get better mappings.
3) In the experiments we had to manually accept and reject proposed mappings repeatedly. Thereby we learned that users can reduce the amount of necessary clicks by first rejecting asset mappings, then accepting process mappings and in the end accepting asset mappings and rejecting process mappings. This order ensures that a maximal amount of rejects and accepts is performed automatically.

C. Threats to Validity

Our experiment is subject to a number of threats.

The main threat to *external validity* is our selection of samples, based on a limited number of open source projects, partially originating from a teaching context. The rationale for our selection was the manual effort for creating the ground truth of our technique, a full mapping between high-level DFD elements and low-level program elements. However, as a result, the generalizability of the results to larger project in other domains is limited. To mitigate this threat, the considered projects were chosen to be representative for realistic projects by providing a good documentation, including architectural information (such as, wikis, use cases, scenarios, requirements, state charts, and the like). The available documentation enabled building good design models, close to the intended architecture. We plan to extend the evaluation in the future to include a more comprehensive set of projects.

Regarding *internal validity*, the main threat of our evaluation is researcher bias. In absence of pre-existing ground truths and design models, the ground truth and design models for our evaluation were created manually by the authors, possibly introducing a risk of creating a biased result. To mitigate this threat, the ground truths and the design-level models were carefully discussed between all authors. The created models and ground truths are of similar size and complexity and are available online: [38].

With respect to *construct validity* we consider the threat of misinterpreting divergence, absence, and convergence compliance violations in the context of design-level models and implementation-level models. However, to the best of our knowledge, our interpretations are in-line with the existing

literature [11]. As such, the implementation of the approach does not perform low-level static or dynamic checks to verify the intended security properties of SecDFD assets. This threatens the intension of the approach to holistically analyze security properties. We discuss the possibilities to extend the plugin to include static and dynamic check as future work. The implemented plugin only notifies the user about the accepted, defined, and missing mappings with in-line information markers. Thus, the user decides what compliance issues the mappings identify. Yet, the implementation can be easily extended to support active proposals of compliance violation types.

V. Related Work

Existing works on maintaining security consistently in different development stages focus on *forward* and *reverse engineering*, that is, the automated transformation of a more high-level to a more technical representation, and back. Considering forward engineering, Ramadan et al. [45] use model transformation to automatically generate security-annotated UML class models [1] from security-annotated BPMN models. For the classical reverse engineering scenario from source code to UML class models, Peldszus et al. [46] propagate hand-crafted security annotations from source code to the corresponding elements in automatically extracted class models. Most closely related to ours is the approach of Abi-Antoun et al. [47], which is concerned with DFD-to-code conformance checks. They automatically reverse engineer a DFD from the given implementation, calling it *source DFD*. The user has to specify a mapping between a manually created *high-level DFD* and the source DFD, which is then used to uncover inconsistencies. In contrast to this manual approach to mapping, our approach is semi-automated: It automatically proposes an initial set of mappings, which is iteratively refined based on user feedback.

Similar to our considered problem of mapping DFD elements to code, *feature location* techniques aim to find the code assets that contribute to the implementation of a given feature. Two existing surveys [48], [49] summarize the variety of available techniques, which largely differ in their assumed input, program representation, and required degree of user interaction. Most closely related to ours are those works that derive an initial mapping based on name similarities and use it as input for a structural search. Zhao et al.'s approach [50] assumes as input a set of features provided by means of one textual description for each feature. They use a information retrieval technique called *Latent Semantic Analysis* (LSA) to identify a set of seed elements deemed as relevant for each feature. They then filter a call graph representation of the input program to remove those branches that do not include a relevant element. Strüber et al.'s approach [51] uses LSA in the same way, then scores all elements based on topology measures and assigns each element to the feature it is deemed most relevant for. Font et al.'s approach [52] assumes user-specific input seeds that they extend with a genetic algorithm to generate a candidate for the implementation of the given feature; a textual description of the input feature is then used to judge the relevance of the identified fragment. In contrast to feature location techniques, which use textual descriptions and manually specified mappings as input, we rely on a different source of information. Our heuristics exploit the rich structural information given by the input DFDs to guide the search of the program model; that is, we exploit an assumed correspondence between the two models being available in our scenario.

Beyond the security scope of this paper, conformance checking is generally a well-studied topic in model-driven engineering. Paige et al. [53] use meta-models as the common reference point to enable conformance checks between diagrams representing different views on a system. Diskin et al. [54] present a framework for global consistency checks of heterogeneous models based on constraints. By supporting the explicit specification of overlaps between the considered models, they avoid the need for a global meta-model. Expanding on this work, König and Diskin [55] improve the efficiency of this approach by supporting an early localization of relevant parts of the models whose consistency is to be checked. Reder and Egyed [56] propose an efficient approach to consistency checking based on predefined consistency rules. However, none of these works address security, and an application to data flow-related threats as addressed by DFDs is not obvious.

VI. Conclusion and Future Work

We presented an interactive, semi-automated approach for mapping concrete implementations to SecDFDs with the aim to perform conformance checks of the implementations with the SecDFDs as well as security checks on the implementations. In the proposed approach mappings are iteratively calculated by heuristics and are presented to a user for verification. Furthermore, the user guides the automated mapping by actively adding additional mappings. We demonstrated how users of our approach can discover convergence, absence and divergences between the SecDFDs and their implementations as well as how the security information available in the SecDFDs can be used for executing security analyses on the source code level.

The approach has been evaluated on five open-source projects (including Eclipse secure storage [17]) and shows good precision and recall for the initial, automatically created mapping. Our evaluation shows that new mappings can be found by considering the user input in later iterations. Consequently, both the user and the proposed heuristics contribute to the discovery of new mappings. All in all the user is not only guided through the implementation by our tool, but also assisted in creating the mappings between SecDFDs and their implementations.

In this work we only studied how we can feed security information from the SecDFDs into existing security analysis tools using the mappings created in this work. In future we will use the mappings to explicitly verify whether the security contracts defined on the SecDFDs can be verified in the implementation. Furthermore, we will study how the heuristics can be improved by considering additional information available in other notations like UML activity diagrams.

References

[1] J. Jürjens, *Secure Systems Development with UML*. Springer, 2005.

[2] C. Dougherty, K. Sayre, R. C. Seacord, D. Svoboda, and K. Togashi, "Secure Design Patterns," Carnegie-Mellon University Pittsburgh, Software Engineering Institute, Tech. Rep., 2009.

[3] A. Bacchelli and C. Bird, "Expectations, Outcomes, and Challenges of Modern Code Review," in *ICSE*, 2013, pp. 712–721.

[4] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The Impact of Code Review Coverage and Code Review Participation on Software Quality: A Case Study of the Qt, Vtk, and Itk Projects," in *MSR*, 2014, pp. 192–201.

[5] A. Shostack, *Threat Modeling: Designing for Security*. John Wiley & Sons, 2014.

[6] V. Saini, Q. Duan, and V. Paruchuri, "Threat Modeling Using Attack Trees," *CCSC*, vol. 23, no. 4, pp. 124–131, 2008.

[7] M. S. Lund, B. Solhaug, and K. Stølen, *Model-driven Risk Analysis: The Coras Approach*. Springer, 2011.

[8] T. Abe, S. Hayashi, and M. Saeki, "Modeling Security Threat Patterns to Derive Negative Scenarios," in *APSEC*, 2013, pp. 58–66.

[9] R. Scandariato, K. Wuyts, and W. Joosen, "A Descriptive Study of Microsoft's Threat Modeling Technique," *RE*, vol. 20, no. 2, pp. 163–180, 2015.

[10] K. Tuma, G. Calikli, and R. Scandariato, "Threat analysis of software systems: A systematic literature review," *Journal of Systems and Software*, vol. 144, pp. 275–294, 2018.

[11] L. De Silva and D. Balasubramaniam, "Controlling software architecture erosion: A survey," *Journal of Systems and Software*, vol. 85, no. 1, pp. 132–151, 2012.

[12] M. Deng, K. Wuyts, R. Scandariato, B. Preneel, and W. Joosen, "A Privacy Threat Analysis Framework: Supporting the Elicitation and Fulfillment of Privacy Requirements," *RE*, vol. 16, no. 1, pp. 3–32, 2011.

[13] L. Sion, K. Yskout, D. Van Landuyt, and W. Joosen, "Solution-aware Data Flow Diagrams for Security Threat Modeling," in *SAC*, 2018, pp. 1425–1432.

[14] G. Macher, E. Armengaud, E. Brenner, and C. Kreiner, "A review of threat analysis and risk assessment methods in the automotive context," in *SAFECOMP*, 2016, pp. 130–141.

[15] B. J. Berger, K. Sohr, and R. Koschke, "Extracting and Analyzing the Implemented Security Architecture of Business Applications," in *CSMR*, 2013, pp. 285–294.

[16] K. Tuma, M. Balliu, and R. Scandariato, "Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis," in *ICSA*, 2019, pp. 191–200.

[17] "Eclipse Documentation – Secure Storage," Eclipse Foundation, Tech. Rep. [Online]. Available: https://help.eclipse.org/oxygen/topic/org.eclipse.platform.doc.user/reference/ref-securestorage-start.htm

[18] N. D. et al., "EGit – User Guide," Eclipse Foundation, Tech. Rep., 2019. [Online]. Available: https://wiki.eclipse.org/EGit/User_Guide

[19] Object Management Group (OMG), "UML 2.5.1 Superstructure Specification," Tech. Rep. OMG Document Number: formal/2011-08-06, 2017.

[20] ——, "Business Process Model And Notation (BPMN)," Tech. Rep., 2011.

[21] K. Tuma, R. Scandariato, M. Widman, and C. Sandberg, "Towards Security Threats That Matter," in *Computer Security*. Springer, 2017, pp. 47–62.

[22] K. Tuma and R. Scandariato, "Two Architectural Threat Analysis Techniques Compared," in *ECSA*, 2018, pp. 347–363.

[23] S. Peldszus, G. Kulcsár, and M. Lochau, "A Solution to the Java Refactoring Case Study using eMoflon," in *TTC*, 2015, pp. 118–122.

[24] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze, "Incremental Co-Evolution of Java Programs based on Bidirectional Graph Transformation," in *PPPJ*, 2015, pp. 138–151.

[25] "GRaViTY Program Model." [Online]. Available: http://gravity-tool.org

[26] S. Peldszus, G. Kulcsár, M. Lochau, and S. Schulze, "Continuous Detection of Design Flaws in Evolving Object-Oriented Programs using Incremental Multi-pattern Matching," in *ASE*, 2016.

[27] S. Ruland, G. Kulcsár, E. Leblebici, S. Peldszus, and M. Lochau, "Controlling the Attack Surface of Object-Oriented Refactorings," in *FASE*, 2018, pp. 38–55.

[29] D. Ganesan, T. Keuler, and Y. Nishimura, "Architecture Compliance Checking at Run-time," *Information and Software Technology*, vol. 51, no. 11, pp. 1586–1600, 2009.

[28] J. Knodel and D. Popescu, "A comparison of static architecture compliance checking approaches," in *WICSA*, 2007, pp. 12–12.

[30] H. Ehrig, G. Rozenberg, and H.-J. Kreowski, *Handbook of Graph Grammars and Computing by Graph Transformation*. world Scientific, 1999, vol. 3.

[31] V. I. Levenshtein, "Binary Codes Capable of Correcting Deletions, Insertions, and Reversals," in *Soviet physics doklady*, vol. 10, no. 8, 1966, pp. 707–710.

[32] I. Chowdhury, B. Chan, and M. Zulkernine, "Security Metrics for Source Code Structures," in *SESS*, 2008, pp. 57–64.

[33] B. Alshammari, C. Fidge, and D. Corney, "Security Metrics for Object-oriented Class Designs," in *ICSQ*, 2009, pp. 11–20.

[34] J. Lerch, B. Hermann, E. Bodden, and M. Mezini, "Flowtwist: Efficient Context-sensitive Inside-out Taint Analysis for Large Codebases," in *SIGSOFT*, 2014, pp. 98–108.

[35] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 259–269, 2014.

[36] "KeY Project." [Online]. Available: https://www.key-project.org

[37] G. Hiet, V. V. T. Tong, L. Me, and B. Morin, "Policy-based Intrusion Detection in Web Applications by Monitoring Java Information Flows," in *CRiSIS*, 2008, pp. 53–60.

[38] "Implementation and Evaluation Data," 2019. [Online]. Available: https://github.com/SvenPeldszus/GRaViTY-SecDFD-Mapping

[39] "JPetStore." [Online]. Available: http://www.mybatis.org/jpetstore-6/

[40] "ATMExample." [Online]. Available: http://www.math-cs.gordon.edu/local/courses/cs211/ATMExample/

[41] "CoCoME." [Online]. Available: https://github.com/cocome-community-case-study

[42] R. Heinrich, K. Rostami, and R. Reussner, *The Cocome Platform for Collaborative Empirical Research on Information System Evolution*. KIT, 2016.

[43] A. Meneely, B. Smith, and L. Williams, "Itrust electronic health care system case study." [Online]. Available: http://agile.csc.ncsu.edu/iTrust

[44] J. Bürger, D. Strüber, S. Gärtner, T. Ruhroth, J. Jürjens, and K. Schneider, "A Framework for Semi-automated Co-evolution of Security Knowledge and System Models," *Journal of Systems and Software*, vol. 139, pp. 142–160, 2018.

[45] Q. Ramadan, M. Salnitri, D. Strüber, J. Jürjens, and P. Giorgini, "From Secure Business Process Modeling to Design-Level Security Verification," in *MODELS*, 2017, pp. 123–133.

[46] S. Peldszus, D. Strüber, and J. Jürjens, "Model-Based Security Analysis of Feature-Oriented Software Product Lines," in *GPCE*, 2018.

[47] M. Abi-Antoun, D. Wang, and P. Torr, "Checking Threat Modeling Data Flow Diagrams for Implementation Conformance and Security," in *ASE*, 2007, pp. 393–396.

[48] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source Code: A Taxonomy and Survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[49] J. Rubin and M. Chechik, "A Survey of Feature Location Techniques," in *Domain Engineering*. Springer, 2013, pp. 29–58.

[50] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang, "Sniafl: Towards a Static Noninteractive Approach to Feature Location," *TOSEM*, vol. 15, no. 2, pp. 195–226, 2006.

[51] D. Strüber, J. Rubin, G. Taentzer, and M. Chechik, "Splitting Models Using Information Retrieval and Model Crawling Techniques," in *FASE*, 2014, pp. 47–62.

[52] J. Font, L. Arcega, Ø. Haugen, and C. Cetina, "Feature Location in Models through a Genetic Algorithm Driven by Information Retrieval Techniques," in *MODELS*, 2016, pp. 272–282.

[53] R. F. Paige, P. J. Brooke, and J. S. Ostroff, "Metamodel-based Model Conformance and Multiview Consistency Checking," *TOSEM*, vol. 16, no. 3, p. 11, 2007.

[54] Z. Diskin, Y. Xiong, and K. Czarnecki, "Specifying Overlaps of Heterogeneous Models for Global Consistency Checking," in *MODELS*, 2010, pp. 165–179.

[55] H. König and Z. Diskin, "Efficient Consistency Checking of Interrelated Models," in *ECMFA*, 2017, pp. 161–178.

[56] A. Reder and A. Egyed, "Incremental Consistency Checking for Complex Design Rules and Larger Model Changes," in *MODELS*, 2012, pp. 202–218.