Automating the Early Detection of Security Design Flaws

Katja Tuma[†], Laurens Sion[*], Riccardo Scandariato[†], Koen Yskout[*]
[†] University of Gothenburg and Chalmers University of Technology, Sweden
[*] imec-DistriNet, KU Leuven, Belgium
katja.tuma@cse.gu.se,laurens.sion@cs.kuleuven.be,riccardo.scandariato@cse.gu.se,koen.yskout@cs.kuleuven.be

ABSTRACT

Security by design is a key principle for realizing secure software systems and it is advised to hunt for security flaws from the very early stages of development. At design-time, security analysis is often performed manually by means of either threat modeling or expert-based design inspections. However, when leveraging the wide range of established knowledge bases on security design flaws (e.g., CWE, CAWE), these manual assessments become too time consuming, error-prone, and infeasible in the context of contemporary development practices with frequent iterations. This paper focuses on design inspection and explores the potential for automating the application of inspection rules to speed up the security analysis.

The contributions of this paper are: (i) the creation of a publicly available data set consisting of 26 design models annotated with security flaws, (ii) an automated approach for following inspection guidelines using model query patterns, and (iii) an empirical comparison of the results from this automated approach with those from manual inspection. Even though our results show that a complete automation of the security design flaw detection is hard to achieve, we find that some flaws (e.g., insecure data exposure) are more amenable to automation. Compared to manual analysis techniques, our results are encouraging and suggest that the automated technique could guide security analysts towards a more complete inspection of the software design, especially for large models.

CCS CONCEPTS

• **General and reference** → **Empirical studies**; • **Security and privacy** → **Software security engineering**; • **Software and its engineering** → **Designing software**; • **Social and professional topics** → **Automation**.

KEYWORDS

security-by-design, secure design, security flaw, design flaw detection, automation, empirical software engineering

ACM Reference Format:
Katja Tuma[†], Laurens Sion[*], Riccardo Scandariato[†], Koen Yskout[*]. 2020. Automating the Early Detection of Security Design Flaws. In *ACM/IEEE 23rd International Conference on Model Driven Engineering Languages and Systems (MODELS '20), October 18–23, 2020, Virtual Event, Canada.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3365438.3410954

MODELS '20, October 18–23, 2020, Virtual Event, Canada

ACM ISBN 978-1-4503-7019-6/20/10…$15.00
https://doi.org/10.1145/3365438.3410954

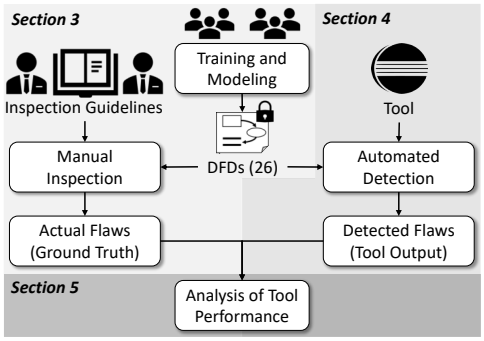

Figure 1: Research activities and paper structure

1 INTRODUCTION

In the current software development culture, agility and speed are paramount. However, software quality in general, and security in particular, cannot be sacrificed in lieu of a faster pace of development. This is where the "shift left" concept comes into place. Namely, software validation and verification should be applied as early as possible in each agile iteration, including the analysis of the design. Indeed, recent work has highlighted that a large share of vulnerabilities disclosed in industrial control systems had their root cause in the design [15].

At design level, mainstream analysis and validation techniques, like threat analysis and design inspections [13, 21, 28], are heavily based on the use of experts performing manual tasks. Therefore, they do not fit well in the agile paradigm of continuous integration and continuous development [10]. More automation of design-level security techniques is necessary, and the research community has responded to this challenge [5, 7, 14, 27]. This paper continues on this research path and explores the automation of design-level security inspection guidelines, which has not been attempted before.

In particular, we select a subset of 5 inspection guidelines from the catalog of Tuma et al. [34] (see Section 2.1) and define **a technique to perform the model inspection automatically**. We assume that a software system is modeled as a Data Flow Diagram. This choice of this model type is justified by the fact that DFDs are widely used in the industry for security analysis purposes. For instance, DFDs are central to threat analysis and are, therefore, already available in companies that have a secure software process in place [16]. In addition, a recent case study [8] involving four companies shows that DFDs are used for threat analysis in agile

Table 1: Security design flaws from the catalog proposed by Tuma et al. [34] (flaws in bold are the focus of this work)

	Flaw number and name	Description
1	Missing authentication	An absence of an authentication mechanism in the system.
2	**Authentication bypass**	There exists an entry point with authentication mechanism that can be bypassed.
3	Relying on single factor authentication	The authentication mechanisms rely on the use of passwords.
4	Insufficient session management	Sessions are not managed securely throughout their life cycle.
5	Downgrade authentication	Possibility to authenticate with a weaker (or obsolete) authentication mechanism.
6	**Insufficient crypto key management**	Keys are not managed securely throughout their life cycle.
7	Missing authorization	An absence of an authorization mechanism in the system.
8	Missing access control	An absence of access control in the system.
9	No re-authentication	An absence of re-authentication during critical operations.
10	Unmonitored execution	Uncontrolled resource consumption due to interactions with external entities.
11	No context when authorizing	An absence of conditional checks for access control.
12	Not revoking authorization	An absence of a process for revoking user access.
13	**Insecure data storage**	Storage of sensitive data is in clear or access control mechanisms are weak.
14	Insufficient credentials management	Credentials are not managed securely throughout their life cycle.
15	**Insecure data exposure**	Sensitive data is transported in clear.
16	Use of custom/weak encryption	Generating small keys, using obsolete encryption schemes.
17	Not validating input/data	Absence of validation checks when receiving data from external entities.
18	**Insufficient auditing**	Access to critical resources or operations is not logged.
19	Uncontrolled resource consumption	Uncontrolled resource consumption of internal components.

organizations. We use an enriched version of DFDs, which are annotated with additional security information (see Section 2.2). As shown in the right-hand side of Figure 1, in Section 4 we describe how the inspection guidelines have been (i) represented as model query patterns by means of VIATRA and (ii) implemented in a prototype tool as an Eclipse plugin. A match of a model query pattern executed by the tool would correspond to the presence of a security design flaw in the analyzed DFD.

To evaluate our technique, we need a ground truth, i.e., design models (DFDs) that are labeled with information concerning the security design flaws that are present in each model. Such a data set did not exist and, in general, the lack of validation data has been a recurring challenge in our field of research. As shown in left-hand side Figure 1, in Section 3 we describe how we have created **a curated data set of 26 security-oriented DFDs** by enrolling 13 modelers who have worked on 4 different software systems, under controlled conditions and with the prescriptions of empirical studies. Additionally, we have employed 2 security experts (co-authors of this paper) to assess the models. The experts have manually applied the 5 inspection guidelines under investigation in this work and have identified the design flaws in all models. The assessment has been performed in an unbiased way, i.e., without any prior knowledge of how the automated technique works. Further, the experts have worked independently and have checked each other's work to a large extent, which provides assurance about the quality of the resulting data set. The data set is now publicly available to the research community and has been used in Section 5 in order to validate the automated technique we propose.

Our results (discussed in Section 6) show that three inspection guidelines have the promising potential of being amenable to automation. Clearly, these results are valid within the confines of the threats to validity presented in Section 7.

2 BACKGROUND

This section provides some background on design flaws, the catalog of inspection guidelines, the Data Flow Diagram (DFD) [12] representation, and its security extensions.

2.1 Design Flaws and Inspection Guidelines

We refer to a security design flaw as a weakness in the high-level design of a system (e.g., software architecture), which exposes the system to security threats. Flaws may lead to code defects [11]. This paper relies on a catalog of security design flaws proposed by Tuma et al. [34]. As shown in Table 1, the catalog consists of 19 common security design flaws concerning authentication, access control, authorization, availability of resources, integrity, and confidentiality of data. It was compiled by means of a systematic analysis of existing vulnerability database entries from several sources (CVE [1], CWE [2], OWASP [23], and SANS [24]). This study focuses on **five** security design flaws in particular, marked in bold in Table 1.

As shown by the example in Listing 1, each design flaw specifies an inspection guideline. The guidelines were developed for manually determining the presence of this security design flaw in a software architecture. Each guideline leads the analyst to the identification of certain locations in the model where the flaw could be present. At those locations, the analyst has to evaluate some criteria (rules) in the form of yes/no questions. A '*no*' answer means that a flaw is present. To help the analyst, the criteria sometimes refer to certain security solutions. But, they do not account for all existing security solutions protecting a data property. For instance, the criterion 'Is there any form of time-stamping, message sequencing or checksum in the exchanged packages?' does not require cryptographic hashing (as opposed to a simple checksum) to be satisfied. In addition, TLS provides message authentication in addition to encryption. Manually exploring design models in such a way is effort intensive and prone to errors. Therefore, automating this assessment activity is desirable, especially in the context of frequent design iterations where redoing such an assessment is prohibitively expensive.

2.2 Data Flow Diagram and Security Extensions

In this work, we automate the inspection of DFDs, which are already extensively used in security threat modeling [17, 28, 33]. The DFD notation is used to graphically represent a system architecture. It highlights the flows of data, showing how the information enters,

Security Design Flaw 15: Insecure Data Exposure

Flaw description Data is not transferred in a secure way. For example a web application uses the HTTP instead of HTTPS. This leaves the channel vulnerable to eavesdropping, Man In The Middle (MITM) attacks etc.

Inspection guideline to detect this flaw

- Locate the valuable information in the model.
- Track them through the architecture to determine where and how they are transferred.
- At each step examine the following:
  - Is the reuse of packets prevented (Replay attacks)?
  - Is there any form of time-stamping, message sequencing or checksum in the exchanged packages?
  - Is the data transferred over an encrypted channel (SSL/TLS)?

Listing 1: Inspection guideline for flaw 15 [34]

traverses, and leaves the system. Figure 2 depicts a high-level DFD for a social network application. The diagram shows how private users and ad companies (both *external entities*) interact with the system, which is modeled as a set of *processes* for authentication (Authenticate), core business logic (Service Provider) and access to the persistence layer (DB access provider). Data is persisted in two *data stores*: key material is in the Key Storage, while user data is in the Social Network Database.

The regular DFD notation is limited in denoting security-related information, making it hard for practitioners to reason about security at design time [7]. To this aim, the DFD notation has been extended in the literature with security properties [32] and security solutions [30]. As shown in Figure 2, the modeler can specify the type of information that is passed around (e.g., sensitive, encrypted data or key material). Furthermore, the modeler can represent the use of security mechanisms: (i) secure pipes (optionally with client authentication) to protect the confidentiality and integrity of data transmitted over data flows, (ii) encrypting data in a data store, (iii) key management solutions (creation, replacement, and destruction), (iv) secure log of access to data stores, and (v) authentication.

3 A CURATED DATA SET OF DESIGN MODELS AND THEIR SECURITY FLAWS

The research field of secure design is plagued by the lack of publicly available 'case studies' that could be used to validate new techniques. In order to overcome this shortcoming, we have set up a series of workshops where we asked 13 participants to model a variety of systems using a DFD-like notation in a design tool. The resulting models have been analyzed for security flaws by 2 expert assessors. The workshops have been carried out with scientific rigor in the form of an empirical study (i.e., under controlled conditions) in order to guarantee the quality of the outcome. The outcome of this study is two-fold: (i) the creation of a data set of 26 DFD design models enriched with security solutions and data types, and (ii) for each model, a report of the existing design flaws (for 5 flaws, shown in bold in Table 1) and their locations. All the material is publicly available on this paper's companion website [3].

3.1 Study Design

Participants and training. The volunteering participants of this study are 13 academic researchers. All participants finished a higher-level degree in the field of computer science and software engineering and are employed at two universities in two different countries. About half of participants (8/13, herein Group A) have a strong background in software design, requirements engineering and modeling. Yet, they are less experienced in software security. The other half (5/13, herein Group B) have a deeper understanding of security-related topics, including secure software design and formal methods for security. All the participants received a training session of 1 hour. This **training** session included an introduction to the DFD modeling notation, the extensions to the DFD notation used in this work, a brush-up on concepts related to software security, and a demonstration of the design tool they will use. The same training material has been used at both universities.

Modeled systems. We prepared a brief description (about one page) for 4 different systems. Each description included an explanation of the system functionality and a list of security requirements.

DriveSafe A smartphone application for achieving safety on the roads collaboratively by continuously updating nearby drivers on current road safety conditions.

BeSocial A proximity-based collaborative messaging smartphone application to support creating and maintaining virtual chat rooms for nearby users.

PhotoFriends A media sharing smartphone application to enable users to share photos and build a network of friends.

SmartTex A collaborative document management web service targeting members of the scientific community to support creating, editing, and compiling LaTeX documents in a collaborative way.

Model creation. In a **randomized** assignment, each participant was given the task to model two of the four systems, by using the DFD notation and its security extension. Individual participants met with the experimenters for a modeling session of about 3 hours. Each participant received a handout package including (i) printed training slides, (ii) a cheat-sheet for the model notation, (iii) a computer with the design tool, and (iv) the descriptions of the systems they had to model. The descriptions are designed in such a way that they can be easily understood in a limited amount of time. Further, the experimenters were available to answer any questions.

Before they started with the task, the participants carried out a short **warm-up** modeling exercise (15 min) to get familiar with the tool. Next, they were given the documentation of their first system. Participants were tasked to read the documentation carefully, and use the tool to create a DFD enhanced with security solutions and data types. To enhance the DFD with security solutions, they instantiated solutions from a provided catalog and bound them to concrete DFD elements. Similarly, they labeled data flows with data types according to a provided data type catalog.

During the modeling session the experimenters took notes and monitored their progress. Finally, the participants were asked to shortly explain their model. After finishing the first model, they received the documentation for the second and repeated the task.

As a result of the modeling sessions, we obtained 26 models [3]. Each DFD model is annotated with labels (e.g., sensitive data) for

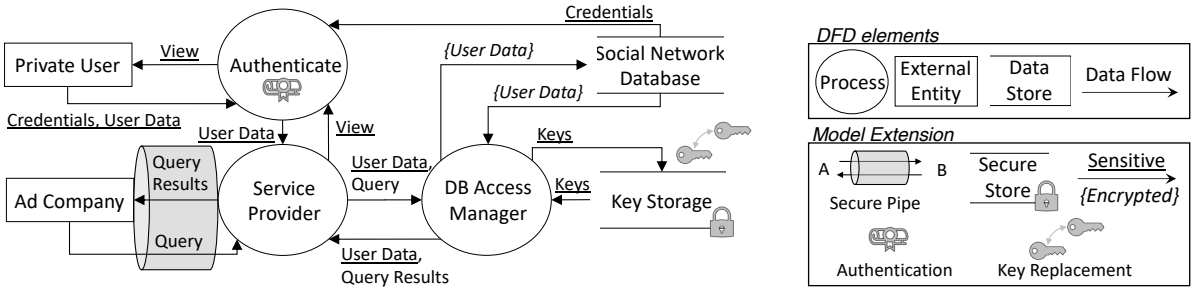

Figure 2: A Data Flow Diagram (DFD) of a social network application (with security extensions)

information assets on the data flows. Also, the models contain elements representing certain security solutions, like encryption on data flows, authentication of external entities, and so on.

Manual model inspection. Two experts manually scrutinized the created models to identify five types of design flaws (in bold in Table 1) by applying the inspection guidelines described in Section 2. As a form of calibration, the assessors independently inspected four randomly selected models (covering the four different systems) and then they compared their results in a joint session.

The discussion of the disagreements resulted in the explicit formulation of common criteria for the subsequent inspections:

- If the participant made any mistakes in the use of notation or logical mistakes (that is, in case of minor mistakes), the experts agreed to take their intention into account.
- If two inspection rules (for different design flaws guidelines) triggered a violation in the same model location, the experts agreed to report only one flaw for this location (the first time it was found). This is related to the fact that some inspection guidelines overlap, as discussed later in the paper.
- They agreed to only consider assets that are mentioned in the security requirements contained in each system description, despite possible deviations in the created models.
- They agreed to assess each model in its entirety, including any additional logic not required according to the documentation.

After the joint session, the experts independently inspected an equal share of the remaining models, which have been assigned randomly to the assessors (by blocking on the four systems). On average, the experts spent about 30 minutes to manually inspect a single model. In the end, they marked a total of five models as requiring further discussion. These models were handed over to the other assessor for a second inspection. The analysis reports were then compared, and any disagreements resolved.

3.2 The Resulting Data Set

The curated data set that emerged from this study can be found online [3]. The data set includes 26 security enriched DFDs, accompanied by expert reports of the flaws identified according to the inspection guidelines. In particular, the flaws are localized on the model and associated to a type (see Table 1).

Statistics about the models. Figure 3 depicts the average number of elements used in the models of each system. On average, a model

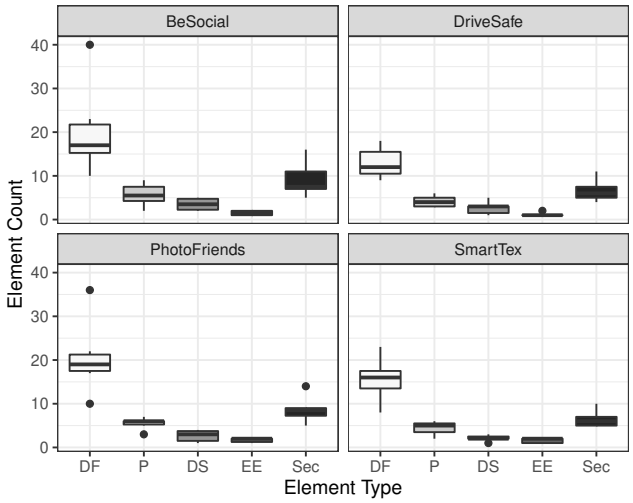

Figure 3: Overview of the model sizes per system

in our data set consists of 17.1 data flow elements, 4.9 processes, 2.7 data stores, 1.5 external entities, and 7.6 security solutions. With respect to element type, the models are fairly uniform across systems. Similar distributions of DFD element types have been observed in the related work [26]. Overall, DriveSafe models are smaller compared to the rest, in particular with respect to the data flows. This is explained by the fact that DriveSafe has a very simple and unidirectional interaction model from the users' perspective.

We also investigate the differences in the created models across participant groups (i.e., the two campuses). The two groups created models of comparable size. Yet, the number of modeled data flow elements varies more within Group B (from 10 to 20). This may indicate that participants of Group A were in fact more experienced in software design modeling and created more homogeneous DFDs.

Statistics about the violations. On average 15.6 flaws are found on a single model. First, we investigated the flaws reported by the assessors to make sure that their analysis was comparable. Overall, the assessors found a similar number of design flaws of each type. Second, we investigated the flaws for each of the four systems. Slight differences can be observed across the four systems. On average, the DriveSafe models contain the smallest number of flaws

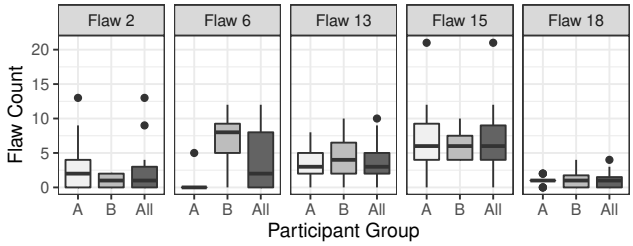

Figure 4: The number of flaws per participant group

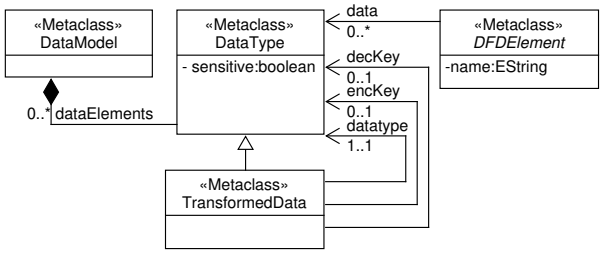

Figure 5: The Data Type Meta-Model

(average per model: DriveSafe: 12.5, SmartTex: 15, BeSocial: 17, PhotoFriends: 18.1). As expected, the average number of flaws seems to correlate with the model size. Systems with larger models (BeSocial and PhotoFriends) contain on average more flaws.

Figure 4 shows the number of security flaws in models created by Group A, Group B, and both groups together. Notably, the total number of insufficient auditing flaws (Flaw 18), regardless of the group, is much smaller compared to the other flaws. A possible explanation is that, in contrast to the other flaws, every instance of this flaw is only associated to a data store element, of which there are typically just a few in each model (see Figure 3). The number of flaws of type 13, 15, and 18 does not differ significantly across groups. This suggests that despite a lesser security background, Group A created similarly (in)secure models with respect to these design flaws. Yet, differences can be observed for what concerns flaw 6 (crypto key management). Often, the less security-oriented group (Group A) did not model key management explicitly, hence making this inspection guideline not applicable. After the modeling sessions, Group A participants explained that they did not feel confident in their security knowledge to model cryptographic details. Also, only a few flaws of type 2 were identified on models created by Group B (average per group: Group A: 2.8, Group B: 1). Possibly, correctly modeling authentication requires a deeper understanding of security mechanisms.

4 AUTOMATED DETECTION OF FLAWS

This section describes the design and implementation of the automated design flaw detection. First, the required model extensions for the automated detection are presented. Next, we describe how these extensions are leveraged in the security design flaw detection. Finally, the model query patterns are discussed.

4.1 DFD Model Extension

The detection of security design flaws relies on the representation of two key concepts in the DFD models: (i) security solutions, which define existing countermeasures, and (ii) data types, which specify what type of information is being processed (especially whether it is sensitive or encrypted data).

Security solutions. The design flaw detection leverages information about existing security solutions in the model. More concretely, checking for the presence of a design flaw can incorporate the following knowledge: (i) the presence of security solutions at correct locations in the model (e.g., the presence of authentication mechanism at the entry points), (ii) the correct instantiation of these solutions, and (iii) the appropriateness of the protection provided by the solution with respect to the involved sensitive data. The existing representation of Sion et al. [30] is used to model the security solutions and capture their effects.

Data types. Data types are an essential concept in security design flaw descriptions [29] and are thus required in the models to support their detection. A concrete DFD is extended with a data model, which is a catalog of all data types that are used in the DFD. All elements in the model (i.e., processes, data flows, data stores, external entities) are linked to the relevant data types in the catalog to track how data moves across the system. Furthermore, the data model allows one to express the relationship between an encrypted piece of data and the original data, including the key (data type) used for encryption and decryption. This way, we capture the notion that 'encrypted' data is a transformation of the original (sensitive) data, such that we can still track where sensitive data is sent or stored after it has been encrypted. Figure 5 shows the meta-model we created for this study to represent these data types. The encrypted version of data is represented as a *TransformedData* instance.

4.2 Leveraging the Extensions for Detection

Table 2 illustrates how the DFD model extensions are used for flaw detection. The top part of this table shows the relationship between security solutions, threat types, and flaws. Rather than hard-coding the set of solutions that can impact the detection of a flaw, the detection criteria of flaws 2, 13 and 15 are expressed using a threat type (e.g., 'information disclosure'). The solutions are associated to the threat types that they prevent. Note that the actual relationship that is implemented in the detection logic is more involved, because it also needs to be verified that an instantiated solution prevents the threat at the correct model location to avoid a flaw. For the key management and logging solutions, the detection logic (for flaws 6 and 18) directly checks for their presence.

The bottom part of Table 2 shows the data types used in this study, and the corresponding flaws that rely on these data types in their detection logic.

Table 2: The use of DFD model extensions for flaw detection

Extension Name	Affected Threat Types	Flaws
Secure pipe	Information disclosure, Tampering, Spoofing	2, 15
Secure pipe with client authentication	Information disclosure, Tampering, Spoofing	2, 15
Authentication	Spoofing	2
Encrypted storage	Information disclosure	13
Key creation	—	6
Key replacement	—	6
Key destruction	—	6
Secure logging	Repudiation	18
Sensitive data	—	2, 13, 15, 18
Encrypted data	Information disclosure	2, 6, 13, 15
Crypto key data	—	2, 6, 13, 15
Session data	—	2

4.3 Detecting Flaws

The security design flaws in focus (see Table 1) were translated to a set of criteria to enable their detection. Below, we describe how the query patterns detect instances of these flaws in concrete models.

Authentication bypass (Flaw 2) This flaw is detected by first filtering for data flows from an external entity to a process which transfer sensitive data. For each of these data flows the flaw triggers if: (i) the data is sent without protection against information disclosure; or (ii) there is no protection against spoofing the external entity. Further, in the case of session data type, the flaw also triggers when there is no protection against tampering.

Insufficient crypto key management (Flaw 6) This flaw is detected by filtering for DFD elements handling a data of type key. The flaw triggers if: (i) the key is insecurely distributed (i.e., there is no protection on the flow against information disclosure, tampering, and spoofing on data flows or processes), (ii) the key is stored insecurely (i.e., there is no protection against information disclosure and tampering on data stores), (iii) a solution for key creation is missing, (iv) a solution for key replacement is missing, or (v) a solution for key destruction is missing.

Insecure data storage (Flaw 13) This flaw is detected by filtering for data stores containing sensitive data and triggers if: (i) the sensitive data is not encrypted (i.e., is not stored as an 'encrypted' *TransformedData*), or (ii) there is no solution to protect against information disclosure.

Insecure data exposure (Flaw 15) This flaw is detected by filtering for data flows transferring sensitive data and triggers if: (i) the sensitive data is not encrypted, or (ii) there is no solution to protect against information disclosure.

Insufficient auditing (Flaw 18) This flaw is detecting by filtering for data stores containing sensitive data and triggers if there is no solution to provide secure logging of access to this data store.

To avoid biasing the results of this work, the development of the query patterns was carefully isolated from the model creation step, and the manual model inspection (see Section 3). First, the implemented query patterns were tested against a separate example system (not part of the data set). Second, implementing and testing the query patterns was completed before the start of participant training. Finally, the experts that performed the manual inspection were not aware of how the automated detection was implemented.

```
// Pattern for Security Design Flaw 15
pattern insecureDataExposure(df : DataFlow){
    // only data flows with sensitive data
    DataFlow.data(df,data);
    find sensitiveDataType(data);

    // if sensitive info is not encrypted
    neg find dataEncrypted(data);
    // and there is no appropriate solution
    neg find flowMitigationAgainstInfoDiscl(df);
}

// Helper pattern to find sensitive data
private pattern sensitiveDataType(dataType : DataType) {
    // data type itself is sensitive
    DataType(dataType);
    DataType.sensitive(dataType,true);
} or {
    // data type is transformation of sensitive data
    TransformedData(dataType);
    TransformedData.data(dataType, data);
    find sensitiveDataType(data);
}
```

Listing 2: Insecure data exposure query pattern in VIATRA

4.4 Implementation

This section briefly describes the implementation of the tool provided to the participants, and the detection of security design flaws. To provide the participants with a tool environment to create the models, we have developed a modeling tool based on the Eclipse platform. The tool uses Ecore[1] to express the meta-model of DFDs, security solutions, and the data types (as discussed earlier). Furthermore, a graphical modeling editor was developed using Sirius.[2] To detect the security design flaws, the above criteria are implemented with VIATRA model query patterns[3] (see, for example, Listing 2 for the pattern for flaw 15). Every security design flaw is specified as a pattern. These patterns are typed with the meta-model elements and declaratively list the criteria for triggering the flaw. To specify more complex situations, the presence or absence of other *helper* patterns can be used. For example, Listing 2 shows how the detection of insecure data exposure can only match if there is sensitive data involved, which can be a data type with the 'sensitive' flag set to true, or a *TransformedData* of sensitive data (determined recursively). The automated detection in concrete user models uses the VIATRA query engine to automatically query the model and provide a list of all the discovered matches in the model, which are exported for subsequent analyses.

5 PERFORMANCE OF THE AUTOMATED INSPECTION TECHNIQUE

We have analyzed the 26 models described in Section 3 with the automated inspection tool described in Section 4.

5.1 Research Questions

We measure the performance of the query patterns in terms of precision and recall with respect to the ground truth, i.e., the inspection performed by expert assessors. Accordingly, we pose two research questions.

[1] https://www.eclipse.org/ecore
[2] https://www.eclipse.org/sirius
[3] https://www.eclipse.org/viatra

*RQ1. What is the **precision** of the automated inspection guidelines (implemented as query patterns) for the detection of five security design flaws?*

We measure true positives (*TPs*) and false positives (*FPs*) to calculate the precision $P = \frac{TP}{TP+FP}$. A true positive is a flaw (guideline violation) which is detected by the tool and that matches an actual flaw reported by experts (i.e., part of the ground truth). A detected flaw matches an actual flaw when they have the same type (design flaw ID) and are attached to the same location in the diagram (model element ID). Otherwise, the flaw is considered a false positive.

A high precision would mean that the automated detection produces a low number of false alarms, which makes the technique meaningful in the context of design-level security analysis by focusing the attention of the analyst. As a term of comparison, the precision of manual design analysis techniques is known to be high (e.g., 0.81 in [26]).

*RQ2. What is the **recall** of the automated inspection guidelines (implemented as query patterns) for the detection of five security design flaws?*

To calculate the recall *R*, we measure false negatives (*FNs*), and calculate $R = \frac{TP}{TP+FN}$. A false negative is an actual flaw (i.e., part of the ground truth) which is not detected by the tool.

A high recall would mean that the automated detection is able to find most actual flaws that are present in the model, providing assurance to the analyst regarding its completeness. However, we remind that the recall of manual design analysis techniques is known to be low (e.g., 0.36 in [26] and around 0.50 in other studies [35]).

5.2 Results

Table 3 presents a summary of the performance results. As shown in the last row, the overall average precision of the automated technique is **P=0.53** and the recall is **R=0.76**. As shown in Table 4 these results are consistent across the four analyzed systems, i.e., there are only small variations in how the technique performs in different systems. The results for about half of the models (15/26) were inspected by the first author against the ground truth to determine if the *FPs* of the tool were in fact overlooked flaws by experts. Though we did not find many overlooked flaws in the ground truth, a more systematic quality evaluation would be beneficial for the data set.

The *first take home message* is that, not surprisingly, it is very hard to attain good performance (precision and recall) when automating the inspection rules. Compared to manual threat analysis techniques (e.g., STRIDE), the precision of our automation is too low to replace expert analysis. However, the higher value of recall is somewhat encouraging, as the automated technique could be used to present an expert with a list of potential issues to sieve through.
The *second take home message* is that some rules seem to be more promising than others as being amenable to automation. Indeed, the precision and recall differ significantly across the query patterns implementing the 5 inspection guidelines, as shown in Table 3.

In the rest of this section we analyze the reasons for false positives and false negatives in the detection of each design flaw. We start from the query patterns with a lower precision and recall (i.e., flaws 18 and 2—in order of increasing performance) and continue with the query patterns with a slightly better precision and recall (i.e., flaws 15, 16, and 13—in order of increasing performance).

Table 3: Precision (P) and recall (R) of the query patterns

Security Design Flaws	TP	FP	FN	P	R
Flaw 2: Authentication bypass	28	58	29	0.33	0.49
Flaw 6: Insufficient key management	56	36	4	0.61	**0.93**
Flaw 13: Insecure data storage	76	16	31	**0.83**	0.71
Flaw 15: Insecure data exposure	166	162	24	0.51	**0.87**
Flaw 18: Insufficient auditing	8	28	17	0.22	0.32
Total	334	300	105	**0.53**	**0.76**

Table 4: Overall precision and recall across systems

System	TP	FP	FN	P	R
BeSocial	95	88	24	0.52	**0.80**
DriveSafe	67	59	21	0.53	0.76
PhotoFriends	95	70	32	**0.58**	0.75
SmartTex	77	83	28	0.48	0.73

Insufficient auditing (Flaw 18) achieved the worse precision (0.22) and recall (0.32). The inspection guidelines for this flaw dictate an analysis of logging mechanisms for critical resources and operations. One possible explanation for the high number of *FPs* (28 compared to 8 *TPs*) is that the participants chose to model assets as sensitive, even when they were not (e.g., "list of user followers" is public in the context of a social network application, but was sometimes labeled as sensitive.). A correct data model is crucial for automated detection since most inspection guidelines suggest focusing on security critical information in the model. Given an incorrect data model, the query pattern was looking for flaws in the wrong locations, producing *FPs*. During the manual inspection of the results vis-a-vis the ground truth (on 15/26 models), we have marked such *FPs* to determine their weight. For this flaw, 4 out of 17 *FPs* were due to mislabeled assets. Therefore, aligning the sensitivity of the modeled assets to the expert analysis would already increase the precision of detecting this flaw.

Authentication bypass (Flaw 2) requires inspecting the entry points of the system to determine if authentication is modeled correctly between the external entities and the processes of the system. In total, there are 57 actual flaws (*TPs* + *FNs*) of this type. Yet, the query pattern detects 86 flaws (*TPs* + *FPs*). Out of those, many are *FPs* (58), and only 28 are *TPs*. We provide two possible explanations. First, the experts took modelers' intention into account while inspecting the models. If the participants modeled authentication incorrectly, minor mistakes were intentionally overlooked, and the flaw was not reported. The query pattern does not perform any quality check of the diagram, which yields *FPs*. Second, compared to the query patterns, experts often reported this flaw on different DFD elements. Given that the DFD model is a kind of *directed* graph, our model distinguishes incoming (element is consuming the data) to outgoing data flows (element is sending data). The query patterns report this flaw on the outgoing data flows (i.e., for the data being sent from the external entity), whereas the experts reported this flaw on the incoming data flow (i.e., for the data being consumed by the external entity). This yields both *FNs* and *FPs*.

Insecure data exposure (Flaw 15) achieved a high recall (0.87) but performed worse in terms of precision (0.51). Similar to flaw 18, a possible explanation for the high number of *FPs* (162 compared to 166 *TPs* and 24 *FNs*) is an incorrect data model. Here, the assets are traced, and the flaws are reported for each model element, from asset source to asset sink. Thus, incorrectly labeled assets may have a larger impact on the precision and recall of the query pattern for detecting design flaw 15. The relatively small number of *FNs* (24) shows that, at least, violations were not overlooked by this query pattern. This also suggests that the participants over-approximated (rather than under-approximated) the sensitivity of assets.

Insufficient key management (Flaw 6) achieved the highest recall (0.93) but performed worse in terms of precision (0.61). The inspection guidelines for insufficient key management suggest identifying cryptographic keys in the model, and analyzing their distribution, storage, creation, replacement, and destruction. The extensions to the DFD notation enable modeling key creation, replacement, and destruction. These security solutions are linked to the assets (of type 'key') and are checked for presence by the execution of the query pattern. For key distribution and storage, the query pattern leverages helper queries implemented for detecting design flaws 15 and 13, respectively. Therefore, the observed *FPs* occur for similar reasons to the ones discussed in those flaws.

Insecure data storage (Flaw 13) achieved a fairly acceptable recall (0.71), and a relatively good precision (0.83). We investigated the reason for a sub-optimal number of *FNs*. One possible explanation is that the inspection rules for security design flaws 13 and 18 overlap. For instance, consider the inspection rule "Is access to data logged?" (from Flaw 13) and "Is access to sensitive data and operations logged?" (from Flaw 18). A systematic application of the inspection rules therefore results in reporting the same violation twice (once for Flaw 13 and once for Flaw 18). During model inspection, experts agreed to report such a flaw only once (the first one they found, which was usually while inspecting for Flaw 13). Instead, missing logs of access trigger the query pattern detecting Flaw 18 (and not Flaw 13). This yields *FPs* for the pattern detecting Flaw 18, and *FNs* for the pattern detecting Flaw 13.

6 DISCUSSION

This section discusses the construction of the data set, and the challenges specific to automating design flaw inspection.

6.1 Creation of the Data Set

During the creation of the data set we have taken additional steps to ensure that the expert assessors calibrated their inspection to achieve repeatable results. Even so, 33% of the reported flaws (over 5 problematic models) were not agreed upon and had to be revisited. The experts had to agree on a common strategy for understanding different requirement interpretations and handling modeling ambiguities. This required more calibration than anticipated.

Requirement interpretations. Early-architecture design models are often created from incomplete system descriptions. Therefore, creating such models means dealing with unknowns and under-specified documentation. If the participants made functional mistakes, the experimenters intervened and warned them to revisit the system description. A systematic assessment of functional correctness was not carried out, as this was seldom the case. But, some security requirements were interpreted differently by our participants. For instance, the requirement: "*In no event, the documents of a customer should be exposed due to a security breach. Hence, the documents have to be stored securely,*" was often understood to require assurance of confidentiality (of documents) for transfer and storage. Another interpretation is that the documents must be stored in a secure storage to which access is logged. In particular, Group A (less security background) often made over-approximations when interpreting security requirements Different requirements interpretations caused participants to extend the DFD with a different data model and, in consequence, different security solutions. This has an effect on the presence or absence of security design flaws. To understand the model (in particular the rationale for extensions), the assessors had to reconstruct the rationale for the created data model vis-a-vis the requirements.

Modeling ambiguities. Different modelers have a variety of ways to model the same software system with the same requirements. As shown in Figure 3, models of the same system can vary in size (e.g., the largest (56) and smallest (17) BeSocial model). These different modeling options have an effect on the presence or absence of security design flaws. For instance, sometimes the participants modeled interactions between the external entity and an authentication process, and between the external entity and all processes representing system functionalities. The participants implicitly assumed sequential and conditional data flows (i.e., the authentication process is invoked first, and only upon success, can the other functionalities be executed). Since the extended notation does not allow a specification of conditional or sequential data flows, this model is ambiguous, and the authentication bypass flaw could be present. The assessors had to interpret the modeler's intention to handle ambiguously modeled DFDs.

To help a manual inspection, we see benefit in (i) operationalizing the guidelines for inspection with reference to element types, and (ii) introducing quality checks for the extended DFD notation.

6.2 Automation

In what follows we describe challenges specific to the automation of design flaw detection and discuss how they can be overcome.

Informal notation. The query patterns were developed by translating natural language inspection rules to relations between elements of the extended DFD notation. According to this translation, the query patterns search for concrete diagram element combinations that are incorrect or problematic. Such an implementation can be broad (e.g., checking for the absence of a security solution). Still, this cannot account for all potential modeling options as modelers may apply shortcuts and (un)intentionally circumvent the detection mechanism. For instance, if the model does not contain sensitive assets, the query patterns will not find insecure data exposure flaws on data in transit. Therefore, the security design flaw detection inherits the problems from the DFD modeling ambiguities.

Modeler assumptions. Furthermore, any model-based detection mechanism relies on these models to precisely reflect the modeler's intentions. It may, however, be possible (due to misinterpretations) that the models actually represent a different situation than intended by

the modeler. For example, modeling application-level encryption, but specifying the resulting encrypted data as sensitive, will cause automated assessments to consider the encrypted data not to be protected against information disclosure.

Similarly, the modeling concept of data is very open and supports many interpretations of which data types exactly would need to be modeled. Given the reliance of some flaws on the sensitivity of data as a key criterion to determine their applicability, the degree of detail in modeling data and correctly assigning its sensitivity has a considerable impact on the detection, as shown in the example with the encrypted sensitive data above.

Going forward. The query patterns are executed on finished models, aiming to achieve a fully automated design flaw detection. This approach does not explore the potential benefits of providing modeling feedback to the modeler while the model is being constructed. Computer-aided detection could overcome some of the modeling challenges discussed above. For instance, our approach could be extended with an appropriate user interface to guide modelers and alert them for potential security design flaws, continuously. Such guidance can also assist modelers in avoiding modeling ambiguities and ensure a more accurate detection of security design flaws. Finally, our approach can be extended to implement the detection of other security design flaws from the catalog.

7 THREATS TO VALIDITY

7.1 Internal Validity

The threats to internal validity that we have identified relate to (i) the descriptions of the four systems, (ii) the construction of the DFD models, (iii) the extension of the DFD models with the data types, and (iv) the construction of the expert assessment baseline.

Descriptions of the four systems. Some of the security requirements mentioned in the descriptions of the four systems might have required the participants to use security solutions which were not provided (as out-of-the-box extensions) or straightforward to model. This threat also relates to the limited security expertise of some participants. In addition, some security requirements were open to interpretation (as discussed in Section 6).

Construction of the models. For the construction of the 26 models there are three concerns. First, the participants had a limited familiarity with the graphical user interface of the modeling tool. To counter this threat, all participants started with a warm-up modeling exercise to ensure they were able to create models and had no remaining questions. Also, at least one author was always present to assist in case any questions or issues arose. Second, the learning effect of working on two systems per participant was controlled by a balanced distribution of the systems. Third and finally, the participant might have perceived some stress in trying to create the models in the foreseen time slots and fatigue due to the length of the session. However, we remark that all participants finished earlier and they could take short breaks if needed.

Extending the models with data types. Since there was no graphical modeling support for adding the data types, participants had to use textual labels on the data flows to specify the data types. These descriptions later had to be included in the model to enable the automated flaw detection patterns to take them into account. We consider the threat of modeling errors that could have been introduced by the authors, when creating the data model from the textual labels. To reduce the impact of errors in the modeling, all models have been checked by two authors.

Construction of the expert assessment. Concerning the assessment of the models by the two experts, we acknowledge that such an assessment could contain errors. For instance, the experts had to interpret the modelers' intentions when assessing the presence of the security design flaws. However, the probability of these errors has been minimized by applying two separate calibration steps between the security experts that performed the assessment.

7.2 External Validity

With respect to the generalization of the results, there are two main threats to the external validity. First, the participants might not be representative of industry professionals. All the participants were researchers with modeling experience, while 5 out of 13 participants had security expertise. Second, the results might be specific to the systems used in this paper. To reduce the impact of this threat, we relied on four different system descriptions which were randomly assigned to the different participants. However, these four systems are relatively similar in size because the participants had to be able to create them in a limited amount of time. An evaluation on systems with varying sizes may be useful to evaluate the impact of the model sizes on the effectiveness of the automated detection.

8 RELATED WORK

In this section, we position our contributions in the context of the related work on automating security design analysis. We also discuss related security design flaw catalogs and works on automating the detection of architectural bad smells and anti-patterns.

8.1 Automation of Security Design Analysis

Recently, Seifermann et al. [27] presented an approach for automatically analyzing security of data-driven architectures. To this aim, they propose an architectural description language enriched with a data model. They transform the architecture to an operation model, which in turn, is automatically transformed to a logic program, where the security analysis is executed. Similar to our data transformations, Seifermann et al. define data processing operations, which seem to be essential for analyzing confidentiality. But, our detection also considers existing countermeasures. Further, Seifermann et al. demonstrate the analysis with logical rules for detecting unauthorized access. Instead, our work is automating the detection of flaws related to several concerns (namely, authentication, confidentiality, integrity, and accountability).

Almorsy et al. [5] propose an approach for automating security analysis by means of capturing security metrics and vulnerabilities as constraints over a detailed system description model. It is beneficial for the analysis to consider system vulnerabilities and defenses side-by-side. Similarly, our query patterns consider design flaws with respect to the security solutions. In addition, the constraints developed by Almorsy et al. [5] rely on the modeler to provide a model. In particular, the constraint about data tampering seems similar to our query pattern detecting insecure data exposure (15). Yet, the introduced constraints detect attack scenarios (e.g.,

denial of service) and assess the system's implemented security (e.g., defense-in-depth), rather than security design flaws.

Berger et al. [7] develop graph query rules to check for vulnerabilities in extended DFD models and evaluate them with case studies. Similar to this work, these rules are based on the descriptions of existing repositories (namely, CWE, and CAPEC). In addition, their approach also extends the DFD model with asset sensitivity. Among others, Berger et al. develop queries to check for authentication bypass and clear text transmission of sensitive information (similar to our query patterns for flaws 2 and 15). An important distinction is that our DFD extended notation includes security mechanisms.

UMLsec [18, 19] is a security extension of the Unified Modeling Language. It enables developers to express security relevant information in system specification diagrams. Similar to our approach, UMLsec relies on security extensions to automatically analyze design models. In contrast, our approach is focused on detecting early security design flaws on architectural models, as opposed to evaluating constraints over specification diagrams.

Katkalov et al. [20] developed a model-driven approach (IFlow) for specifying and analyzing information flow properties on UML models. The authors extend the UML model and transform it to a formal model, which is used to refine the UML generated code skeleton. The proposed approach can leverage static analysis to verify information flow properties in the implemented code skeleton, as well as an interactive theorem prover to verify the properties on the formal model. Similar to this work, IFlow requires the developer to provide information about the sources (of confidential information). In contrast, IFlow is a formal approach that analyzes the non-interference property.

Hoisl et al. [16] present an approach for modeling and enforcing object flows in process-driven Service-Oriented Applications (SOAs). The authors provide a metamodel for defining and enforcing secure object flows in process-driven Service-Oriented Architectures and develop model transformations to generate platform specific models. Similar to the data transformations in our data model, they introduce a semantics of control nodes (i.e., fork, join, decision, and merge) to reason about secure object flows. In addition, their approach is used to automatically analyze the confidentiality and integrity of data flows in a model representation. But, this work extends the model notation with security solutions and focuses on the detection of a variety of security design flaws.

Frydman et al. [14] propose an approach accompanied by a tool (AutSEC) for automating threat modeling and risk assessment of software designs. To identify threats in annotated DFDs, the authors introduce identification and mitigation trees. They obtain the DFD annotations by maintaining maps of common DFD element labels (e.g., web server can be mapped to the "Apache" label). Similar to this paper, Frydman et al. extract information about assets and security mechanisms from the user-extended DFDs. Yet, their diagram extensions are based on user-provided labels and map-like data structures. In contrast, this work allows modelers to explicitly model data properties and security solutions in DFDs.

For a more detailed account on automated design analysis techniques, we refer the reader to a systematic literature review [33].

8.2 Security Design Flaw Catalogs

We briefly mention the related work on security design flaw catalogs. Santos et al. [25] compiled a catalog of common security architectural weaknesses. Similar to the catalog used in this work, their catalog focuses on design-level security flaws. Arce et al. [6] compiled a list of top 10 security design flaws to raise awareness among software architects about the most common design issues leading to security breaches. Indeed, a few inspection guidelines of the catalog used in this study are in line with this list (e.g., 'use cryptography correctly' is related to our design flaw 6). Nafees et al. [22] propose a template for detecting architectural anti-patterns and a catalog of 12 Vulnerability Anti-Patterns. The purpose of their catalog is to bridge the communication gap between security experts and software developers. We also mention the existing corpora (i.e., CWE, CVE, OWASP, SANS, CAPEC) describing common security weaknesses, vulnerabilities, and mitigations.

8.3 Architectural Bad Smells and Anti-Patterns

We briefly mention the related work on detecting architectural bad smells and anti-patterns but remind the reader that none of these approaches analyze the architecture with respect to security. Bouhours et al. [9] introduce a catalog of so called 'spoiled patterns' and automatically detect them in architecture models. To this aim, they extend an existing OWL ontology. Their approach suggests according model transformations to the user. Taibi and Lenarduzzi [31] compile a catalog of 11 bad smells specific to microservice architectures by means of conducting interviews with developers. They emphasize the importance of analyzing microservices that expose private data and shared resources, which is interesting from a security perspective. For a more complete account of existing literature on design smells detection we refer the reader to the mapping study by Alkharabsheh et al. [4].

9 CONCLUSION

This paper has presented three main contributions. First, we have shared with the research community a data set of design models (in a notation based on DFD), created by thirteen participants with a varied background, that model four systems with a varied set of security requirements, and that are annotated with identified design flaws. These models can be used to validate existing and future security techniques. Second, we have attempted to automate five model inspection guidelines for security to detect secure design flaws. These guidelines are meant to be used by security experts and, hence, are difficult to automate, as humans are better suited to execute tasks that involve fuzzy and/or incomplete instructions. Third, we have performed an empirical evaluation that, compared to the ground truth created by manual analysis, shows that automating some of the guidelines is possible with acceptable precision and recall, albeit others are more challenging. Also, our work has pointed out some limitations (e.g., overlaps, unclear rules) in the inspection guidelines themselves. As part of the future work, the results of this paper are being used to improve the quality of the inspection guidelines. Further, we are interested in extending the data set and particularly welcome the contribution of the wider research community.

REFERENCES

[1] 2020. CVE - Common Vulnerabilities and Exposures. Available from MITRE. https://cve.mitre.org

[2] 2020. CWE - Common Weakness Enumeration. Available from MITRE. https://cwe.mitre.org

[3] 2020. Security Design Flaw Detection: Companion Web-Site. Available from Google Sites. https://sites.google.com/view/companion-web-site/

[4] Khalid Alkharabsheh, Yania Crespo, Esperanza Manso, and José A Taboada. 2019. Software Design Smell Detection: a systematic mapping study. *Software Quality Journal* 27, 3 (2019), 1069–1148.

[5] Mohamed Almorsy, John Grundy, and Amani S Ibrahim. 2013. Automated software architecture security risk analysis using formalized signatures. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE, 662–671.

[6] Iván Arce, Neil Daswani, Jim Delgrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary Mcgraw, Brook Schoenfield, Margo Seltzer, Diomidis Spinellis, Izar Tarandach, and Jacob West. 2014. *Avoiding the Top 10 Software Security Design Flaws*. Technical Report. IEEE Center for Secure Design.

[7] Bernhard J Berger, Karsten Sohr, and Rainer Koschke. 2016. Automatically extracting threats from extended data flow diagrams. In *International Symposium on Engineering Secure Software and Systems*. Springer, 56–71.

[8] Karin Bernsmed and Martin Gilje Jaatun. 2019. Threat modelling and agile software development: Identified practice in four Norwegian organisations. In *2019 International Conference on Cyber Security and Protection of Digital Services*. IEEE, 1–8.

[9] Cédric Bouhours, Hervé Leblanc, and Christian Percebois. 2009. Bad smells in design and design patterns. *The Journal of Object Technology* 8, 3 (2009), 43–63.

[10] Daniela Soares Cruzes, Martin Gilje Jaatun, Karin Bernsmed, and Inger Anne Tøndel. 2018. Challenges and experiences with applying Microsoft threat modeling in agile development projects. In *2018 25th Australasian Software Engineering Conference (ASWEC)*. IEEE, 111–120.

[11] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. 2010. On the impact of design flaws on software defects. In *2010 10th International Conference on Quality Software*. IEEE, 23–31.

[12] Tom DeMarco. 1979. *Structured Analysis and System Specification*. Yourdon.

[13] Mina Deng, Kim Wuyts, Riccardo Scandariato, Bart Preneel, and Wouter Joosen. 2011. A privacy threat analysis framework: supporting the elicitation and fulfillment of privacy requirements. *Requirements Engineering* 16, 1 (2011), 3–32.

[14] Maxime Frydman, Guifré Ruiz, Elisa Heymann, Eduardo César, and Barton P Miller. 2014. Automating risk analysis of software design models. *The Scientific World Journal* 2014 (2014), 248–259.

[15] Danielle Gonzalez, Fawaz Alhenaki, and Mehdi Mirakhorli. 2019. Architectural security weaknesses in industrial control systems (ICS) an empirical study based on disclosed software vulnerabilities. In *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 31–40.

[16] Bernhard Hoisl, Stefan Sobernig, and Mark Strembeck. 2014. Modeling and enforcing secure object flows in process-driven SOAs: an integrated model-driven approach. *Software & Systems Modeling* 13, 2 (2014), 513–548.

[17] Michael Howard and Steve Lipner. 2006. *The security development lifecycle*. Vol. 8. Microsoft Press Redmond.

[18] Jan Jürjens. 2002. UMLsec: Extending UML for secure systems development. In *International Conference on The Unified Modeling Language*. Springer, 412–425.

[19] Jan Jürjens and Pasha Shabalin. 2004. Automated verification of UMLsec models for security requirements. In *International Conference on the Unified Modeling Language*. Springer, 365–379.

[20] Kuzman Katkalov, Kurt Stenzel, Marian Borek, and Wolfgang Reif. 2013. Model-driven development of information flow-secure systems with IFlow. In *2013 International Conference on Social Computing*. IEEE, 51–56.

[21] Mass Soldal Lund, Bjørnar Solhaug, and Ketil Stølen. 2011. A guided tour of the CORAS method. In *Model-Driven Risk Analysis*. Springer, 23–43.

[22] Tayyaba Nafees, Natalie Coull, Ian Ferguson, and Adam Sampson. 2017. Vulnerability anti-patterns: a timeless way to capture poor software practices (vulnerabilities). In *Proceedings of the 24th Conference on Pattern Languages of Programs*. The Hillside Group, ACM, 23.

[23] OWASP. 2017. OWASP Top Ten Project. https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

[24] SANS. 2011. SANS Top 25 Software Errors. https://www.sans.org/top25-software-errors/

[25] Joanna CS Santos, Katy Tarrit, and Mehdi Mirakhorli. 2017. A catalog of security architecture weaknesses. In *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 220–223.

[26] Riccardo Scandariato, Kim Wuyts, and Wouter Joosen. 2015. A descriptive study of Microsoft's threat modeling technique. *Requirements Engineering* 20, 2 (2015), 163–180.

[27] Stephan Seifermann, Robert Heinrich, and Ralf Reussner. 2019. Data-Driven Software Architecture for Analyzing Confidentiality. In *2019 IEEE International Conference on Software Architecture (ICSA)*. IEEE, 1–10.

[28] Adam Shostack. 2014. *Threat Modeling: Designing for Security*. John Wiley & Sons. 590 pages.

[29] Laurens Sion, Katja Tuma, Koen Yskout, Riccardo Scandariato, and Wouter Joosen. 2019. Towards Automated Security Design Flaw Detection. In *Proceedings of the 2nd International Workshop on Security Awareness from Design to Deployment (SEAD '19)*. IEEE, 49–56.

[30] Laurens Sion, Koen Yskout, Dimitri Van Landuyt, and Wouter Joosen. 2018. Solution-aware Data Flow Diagrams for Security Threat Modelling. In *Proceedings of SAC 2018: The 6th track on Software Architecture: Theory, Technology, and Applications (SA-TTA)*. ACM, 1425–1432.

[31] Davide Taibi and Valentina Lenarduzzi. 2018. On the definition of microservice bad smells. *IEEE software* 35, 3 (2018), 56–62.

[32] Katja Tuma, Musard Balliu, and Riccardo Scandariato. 2019. Flaws in Flows: Unveiling Design Flaws via Information Flow Analysis. In *ICSA*. IEEE, 191–200.

[33] Katja Tuma, Gül Calikli, and Riccardo Scandariato. 2018. Threat analysis of software systems: a systematic literature review. *Journal of Systems and Software* 144 (2018), 275–294.

[34] Katja Tuma, Danial Hosseini, Kyriakos Malamas, and Riccardo Scandariato. 2019. Inspection Guidelines to Identify Security Design Flaws. In *International Workshop on Designing and Measuring CyberSecurity in Software Architecture (DeMeSSA)*. ACM, 116–122.

[35] Katja Tuma and Riccardo Scandariato. 2018. Two Architectural Threat Analysis Techniques Compared. In *European Conference on Software Architecture*. Springer, 347–363.