ML-Supported Identification and Prioritization of Threats in the OVVL Threat Modelling Tool

Andreas Schaad(✉) and Dominik Binder

Department of Media and Information, University of Applied Sciences Offenburg, Badstraße 24, 77652 Offenburg, Germany
{andreas.schaad,dbinder}@hs-offenburg.de

Abstract. Threat Modelling is an accepted technique to identify general threats as early as possible in the software development lifecycle. Previous work of ours did present an open-source framework and web-based tool (OVVL) for automating threat analysis on software architectures using STRIDE. However, one open problem is that available threat catalogues are either too general or proprietary with respect to a certain domain (e.g. .Net). Another problem is that a threat analyst should not only be presented (repeatedly) with a list of all possible threats, but already with some automated support for prioritizing these. This paper presents an approach to dynamically generate individual threat catalogues on basis of the established CWE as well as related CVE databases. Roughly 60% of this threat catalogue generation can be done by identifying and matching certain key values. To map the remaining 40% of our data (~50.000 CVE entries) we train a text classification model by using the already mapped 60% of our dataset to perform a supervised machine-learning based text classification. The generated entire dataset allows us to identify possible threats for each individual architectural element and automatically provide an initial prioritization. Our dataset as well as a supporting Jupyter notebook are openly available.

Keywords: Threat analysis · Vulnerability management · Risk assessment

1 Introduction

STRIDE [1] allows to determine possible threats as part of a secure system design activity [2, 3]. It is an accepted industrial-strength technique within the overall secure software development lifecycle [4]. The basic idea of STRIDE is to model a system as a data flow diagram (DFD) and then apply the STRIDE mnemonic on the elements of the DFD. Table 1 summarizes this general mapping. Various tools have been proposed that implement this technique [5–9], including our recently introduced OVVL (https://ovvl.org/) tool [10, 11]. In essence, all of them allow an architect to sketch a data flow diagram of a system and the tools will provide him with a list of possible threats. However, there are several problems with this approach.

A core problem is that the identified threats (the underlying threat catalogue) are often too abstract and vague ("Process could be subject to Tampering") [12] or too domain specific and proprietary to the tool that is used ("An adversary having access to Microsoft Azure Cosmos DB may read sensitive clear-text data") [5].

Table 1. Static mapping of STRIDE threats to DFD elements

	Spoofing	Tampering	Repudiation	Information disclosure	Denial of service	Elevation of privilege
Interactors	X		X			
Processes	X	X	X	X	X	X
Data stores		X		X	X	
Data flows		X		X	X	

Another problem is that a threat analyst will be presented repeatedly with the same information and not with selected subsets of threats individually generated for each architectural element [13].

A third problem is that existing threat modelling tools generate too many "false positives" as they bluntly apply the STRIDE technique over the DFD. Earlier work of ours [6] has reported that an experienced security architect may only deem ~20% of automatically reported threats as relevant enough to push them to an issue management system as part of the later secure software lifecycle.

While many different factors (e.g. professional experience, internal/external guidelines, tacit knowledge) do influence the decisions about automatically reported vs. realistically applicable threats, we believe that a threat modelling tool should provide some support for automated identification and prioritization of "potential" threats as well as individual assignments to each architectural element.

Our approach thus aims at replacing static and proprietary threat catalogues with dynamic threat catalogues that are generated based on additional information about the designed system and information from current external weakness and vulnerability databases such as CWE and CVE. This approach also supports an initial prioritization of identified threats. However, this reconciliation can only be done for ~60% of the CVE entries by means of standard database key matching. For the remaining 40% (~50.000 entries) we suggest a different automated matching technique using machine-learning based text classification.

This paper will thus provide a more detailed discussion of the required technical background regarding STRIDE-based threat modelling automation and our existing OVVL tool (Sect. 2). We then suggest a design and implementation of mapping STRIDE against the established CWE and CVE databases (Sect. 3). In particular, we explain our machine-learning based text classification approach to support a substantial part of this mapping as part of Sect. 4. Section 5 will address integration and usage of the obtained dataset (threat catalogue) in the OVVL framework. Section 6 will provide a summary and discussion about the validity of our approach as well as its limitations. Our dataset as well as a supporting Jupyter notebook are openly available in the OVVL project repository [14].

2 Background Work

2.1 STRIDE-Based Threat Modelling

STRIDE [1] is a method to determine possible threats as part of a secure system design activity. It is an accepted industrial-strength technique within the overall secure software development lifecycle.

Microsoft's Threat Modelling tool [5], though not supported anymore, is an openly available tool that allows a threat analyst to depict a complex system as a set of data flow diagrams. The STRIDE mnemonic is applied on model-elements (compare Table 1) and the analyst is presented with a list of possible threats. However, this toolset is not open-source and the threat catalogue is proprietary with respect to .Net and Microsoft Azure architectures. The OWASP Threat Dragon is another openly available tool [7] that uses STRIDE on DFDs as its primary technique.

Similar commercial tools do exist such as IriusRisk [8] and ThreatModeler [9]. IriusRisk does promote that threat modelling should have an impact on the later stages of a secure development lifecycle. ThreatModeler is based on the VAST method which explicitly distinguishes between application threat models and operational or infrastructure threat models. We will now discuss how these approaches are addressed in our OVVL tool.

2.2 Weakness and Vulnerability Databases

The core idea of this paper is that we can use existing weakness and vulnerability databases in combination with the STRIDE approach and our OVVL data model to dynamically generate threat catalogues for each model element, including already a system suggested prioritization.

CWE is a community-developed list of common software security weaknesses [15]. It contains 808 common weaknesses and 295 categories (i.e. sets of CWEs that share a common characteristic). As part of the CWE data model, the "Common Consequences" attribute already provides a link to technical impacts that can result from each weakness in CWE. For example, CWE-20: "Improper Input Validation" belongs to the category "Validate Inputs" and one of the possible common consequences may be "Modify Memory; Execute Unauthorized Code or Commands". This common consequence applies to the scope of Confidentiality, Integrity and Availability and is attributed with a generally "High" Impact. CWE-20 then points to more detailed CWEs such as CWE-129 "Improper Validation of Array Index", each with more specific consequences, scope and impact.

CVE is a list of publicly known technical vulnerabilities that exploit weaknesses [16]. CVEs point to none, one or more related CWEs. This allows common static code analysis tools such as [17] to scan code for weakness or vulnerabilities such as the mentioned input validation. Interestingly, only 60% of all existing CVEs will directly map to a unique CWE, i.e. without any indirection via a category. In total, this mapping will yield a subset of 170 CWEs (out of 808). The remaining 40% of all CVEs (~51,489) can not be mapped directly to a unique CWE, however 30% of these (~15,280) will only point to a CWE category instead. This is an important observation as we can only use the “Common Consequences” data structure of a CWE to determine suitable keys for an automated mapping of CWEs to the STRIDE elements. In contrast, a CWE category does not offer any data structure similar to these “Common Consequences” and as such we require a different approach to map CWEs to STRIDE.

An immediate question may now be why we do not just directly map STRIDE to a CWE. The short answer is that we will need to analyse the association between CWEs and CVEs to determine a possible priority of a CWE-based threat when presented to the security architect.

2.3 The OVVL Approach and Tool

As part of our ongoing research we developed a method and supporting tool called OVVL (Open Weakness and Vulnerability Modeler) [4, 5]. Similar to Microsoft’s SDL tool [5] and TAM2 [6] that was developed at SAP, we support the graphical definition of data flow diagrams consisting of processes, interactors, data stores and data flows. Once such a diagram has been established, the STRIDE mnemonic is applied over all model elements resulting in a list of possible threats per element. Where deployment information is already known for an element, we support the search for a CPE identifier (a unique serial number for a software component) and can thus already identify all existing CVE entries for that CPE.

3 Design and Technical Approach

Overall, we first build a new database of threats (a threat catalogue) and then use this in combination with information from the system model to identify threats for each architectural element. Our data mapping approach to support the first activity consists of three steps:

1. We build a dataset out of the CVE and CWE database by using the already existing database key that points from a CVE to a CWE (bearing in mind that such a key either does not exist or is not suitable for 40% of the CVEs because it only points to a CWE category).
2. We then map elements of this new dataset against STRIDE using the “Common Consequences” data structure we obtained from the previously mapped CWEs.
3. We use a text classification approach which will eventually allow us to map the remaining 40% of our new dataset to STRIDE. The training set for this is based on the results of the previous step 2.

The remainder of this section will now discuss the preparatory steps 1 and 2, while we will address the problem of machine-learning based text classification as part of the dedicated Sect. 4.

3.1 Step 1: CWE and CVE Mapping

Both datasets, though in parts heavily segmented, are publicly available as JSON (CVE) or XML (CWE) structures. For CVE we worked with all available data (2002–2019) (730 MB size) and for CWE we equally use data up until December 2019 (9 MB size).

In the case of the CVE dataset, this information includes a unique identifier, a description of the vulnerability, which is later used for our text classification process (Sect. 4), and the input values and metrics for the calculation of the CVSS score (Common Vulnerability Scoring System), which will be used for the OVVL framework integration (Sect. 5). Additionally, in most cases, a reference to one or more CWEs is included, indicating the weakness a vulnerability may exploit.

As part of the CWE record, each weakness also contains a unique identifier that matches the identifier of the CVE record so that it can be used to merge the two datasets in form of `pandas DataFrames`. However, since the CVE record can have multiple CWE-IDs per entry, these entries are extended (a CVE entry with two CWE-IDs becomes two entries with one CWE-ID each). From the CWE dataset we also get further information about the circumstances in which a weakness occurs, which are also used in the attribute selection as part of the threat modelling process (Sect. 5). Additionally, the dataset contains the already mentioned "Common Consequences" field, which is used for the STRIDE mapping in the following step (Sect. 3.2).

3.2 Step 2: STRIDE Mapping

Our new dataset contains the "Common Consequences" data structure which consists of a "Scope" and an "Impact". We can use these to map each STRIDE category as shown in Tables 2 and 3.

As some information is redundant, we initially perform our mapping on basis of Table 2 (Scope) and if additionally required using Table 3 (Impact).

The scope indicates which protection target is affected by a certain weakness. As the threats used in the STRIDE methodology are also designed to cover a specific protection target, this allows us to map our dataset directly to the related STRIDE threats based on the given scope.

Table 2. Mapping STRIDE threats based on the Scope

Scope	Affected entries	(STRIDE) threat
Availability	55,240	**D**enial of service
Confidentiality	70,996	**I**nformation disclosure
Non-Repudiation	6,378	**R**epudiation
Authentication	408	**S**poofing
Accountability	76	**R**epudiation
Integrity	62,755	**T**ampering

To additionally assign the values for "Elevation of Privilege" and "Spoofing"-Threats, two impact values were also used, which in contrast to the scope values do not indicate which concrete protection goal is affected, but rather which effects can arise from an attack on that protection goal.

Table 3. Mapping STRIDE threats based on the Impact

Impact	Affected entries	(STRIDE) threat
Gain Privileges or Assume Identity	10,655	**S**poofing, **E**levation of Privilege
Execute Unauthorized Code or Command	50,333	Spoofing, Elevation of Privilege

However, as we already indicated earlier, this overall mapping exercise can only be used for 60% of our dataset using available keys and the remaining 40% will now be mapped on basis of a text classification approach (compare Fig. 2).

4 Text Classification

To map the remaining 40% of our data we train a text classification model by using the already mapped 60% of our dataset to perform a supervised machine-learning based classification. To accomplish this, we use the available "CVE-Description" for each entry in the dataset, since this field exists for all entries and provides the most information about what threat may potentially give rise to a vulnerability. Since the STRIDE methodology distinguishes six different threats, we create six separate models, each of which performs a binary classification that uses a pre-processed description of a vulnerability to determine if the threat may be enabling that vulnerability. Our dataset as well as a supporting Jupyter notebook allowing to reproduce all steps is available as part of the OVVL project repository [14].

4.1 Pre-processing

We transform the gathered data (CVE-Description) into a machine interpretable format. This activity mainly includes performing a word-tokenization, removing stop words as they have no meaning in our application scenario, perform a lemmatisation and finally vectorize the data.

With the word-tokenization, the textual representation of the CVE description, in the form of a string, is broken down into individual words and converted to lower case.

Stop words are then removed from these individual words, as they contain hardly any useful information for our classification process and are frequently found in natural language. We therefore use a predefined list of 179 words from the python library "Natural Language Tool Kit" (NLTK).

By performing a lemmatization these different inflections are reduced to the lemmata of the word, so that all the inflections are treated equally in the following steps.

To be able to use a set of words for the machine-learning process, they must be transformed into a vector in a suitable form. Therefore, we used the (frequently used) method of "term frequency–inverse document frequency" (tf-idf) vectorization. This allows us to vectorize existing unigrams and bigrams from all vulnerability descriptions by calculating the relation of a term-frequency divided by the document frequency of each term. For the practical implementation the `TfidfVectorizer` of the `scikit-learn` library with a maximum of 5000 features was used.

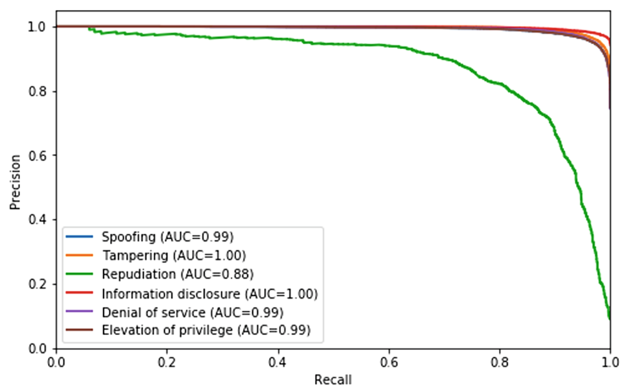


Fig. 1. Precision-Recall-Curve for each threat with corresponding area under the curve (AUC)

4.2 Model Selection

This step involves deciding about which machine-learning model may be most suitable for the intended text classification. Due to the sparseness of the vectorized features, not every type of model is suitable for our classification task. Therefore, we test and evaluate a selection of several models (which are listed in Table 4). For the evaluation it has to be considered that the classes to be calculated are partially heavily unbalanced, which is why an f1-score metric is used for the evaluation. A k-fold cross-validation method with k = 3 is also used for the comparison (Table 4).

Table 4. Model comparison, average f1-score for each threat

Model	S	T	R	I	D	E	Avg.
Naive Bayes	0.932	0.938	0.699	0.966	0.932	0.932	0.9
Logistic Regression	0.952	0.96	0.755	0.978	0.953	0.952	0.925
Decision Tree	0.933	0.946	0.733	0.97	0.935	0.933	0.908
Random Forest	0.954	0.962	0.778	0.979	0.955	0.954	0.93
LightGBM	**0.954**	**0.963**	**0.799**	**0.98**	**0.956**	**0.954**	**0.934**

4.3 Hyper-parameter Optimization

In this step, we sought suitable hyper-parameters to further improve the results of the `LightGBM` model [18]. For this purpose, a set of parameters were defined from which the best combination was determined by grid-search. To avoid overfitting, the pre-processed data was divided into a training and a validation dataset. The training dataset was again used for a k-fold training process with k = 3. The best parameters were then tested on the validation dataset. Since we use a single model for each threat, this process is run

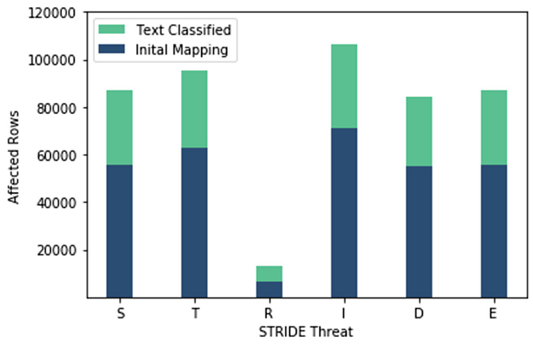


Fig. 2. Complete dataset (60% through key mapping/40% textual classification)

six times separately and the models are stored for further use. However, the achieved optimization appeared to be negligible in our application context.

4.4 Evaluation

In Sect. 4.2, the quality of the model was already measured by calculating the f1-score. In this section, the obtained results will be examined in more detail and based on these evaluations, decisions will be made for practical use, i.e. analysing the PRC curves and determining thresholds by means of the ROC curves (which are excluded for reasons of space but are included in the Jupyter notebook [14]).

Precision-Recall-Curve (PRC). The f1-score (Table 4) is calculated from the precision and the recall score of the model. This calculation takes place independently of the used threshold, which indicates the value at which a calculation (in our case a threat) should be considered positive. However, it is extremely important to observe the threshold, especially in the case of unbalanced data, as in this case. The precision-recall-curve indicates the ratio between precision and recall for the respective threshold. The area under the curve can be used to measure the general quality of the model.

It can be seen from Fig. 1 that our models are highly suitable for identifying the STRIDE threats using the CVE description. The area under the curve for five of the models is nearly (rounded) 1, indicating an extremely effective model. Only the calculation of the "Repudiation" threat performs worse, which we assume is due to the fact that this class is severely underrepresented in the training dataset.

5 OVVL Framework Integration

In order to integrate the obtained mappings into our existing tool we need to decide about suitable attributes that a security architect can select when modelling a system. These attributes will then support the querying of our dataset to generate threat catalogues for each model element. As our dataset is rather large and feeds into a web application, we suggest an approach to pre-compute and reduce threat catalogues based on possible combinations of selected attributes.

5.1 Attribute Selection

In the OVVL Threat Modelling Tool, an analyst can select different attributes and values for each architectural element (e.g. a webserver is "remotely" accessible and requires "user authentication").

The attributes we suggest act as parameters to the queries that yield individual threats for each element in a threat model (i.e. DFD based system description). When choosing suitable attributes, attention was paid to the fact that these are available as completely as possible in our dataset. They should also have a meaningful relation to the threats as well as the environment in which a threat may give rise to vulnerability. Based on these criteria, we were able to identify four suitable attributes (Fig. 3) within our dataset.

The "Access Vector" attribute describes whether a DFD element is accessible either via an external network or locally, which is relevant because certain vulnerabilities and thus weaknesses can only be exploited via one of the two access vectors.

The "Authentication" attribute specifies whether a user must authenticate to access the element and thus also whether the vulnerabilities found can only be exploited by authenticated users

Some more specific weaknesses from the CWE dataset can be traced back to the used programming language. This information is of interest because certain weaknesses are highly dependent on the programming language used. For example, weaknesses that involve memory manipulation are more likely to occur in low-level languages such as C or C++, since the correct memory management must be observed during programming. For our approach, we have assigned individual languages of the CWE dataset to meaningful groups, based on their characteristics. These groups are low level languages (C, C++, Assembly), interpreted languages (Python, Ruby, JavaScript, Perl and other interpreted languages), languages with just-in-time compilation (C#, Java) and languages that are primarily used in web development (PHP, ASP .NET).

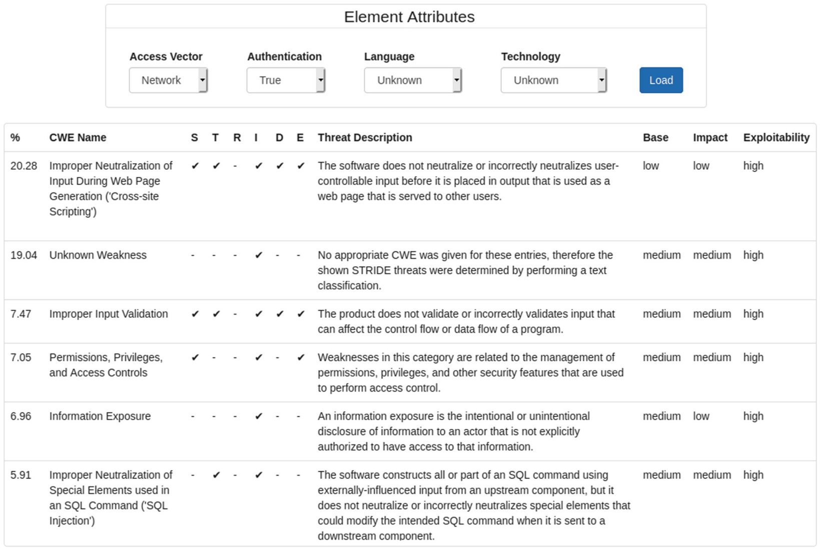
The "Technology" Attribute allows differentiation between web and database servers. For both variants, there are several threats that usually only occur with these technologies, such as the frequently occurring sql injection in database systems or a variety of web application related vulnerabilities in web servers. In addition to these two, there are clearly other technologies (e.g. relating to virtualization) or application types (e.g. directly communicating via sockets) which cannot be further specified on the basis of our dataset. For these, as well as for all other attributes, it is possible to indicate that the respective value is unknown, whereby this attribute is not taken into account (compare Sect. 5.2).

5.2 Dataset Reduction

For an efficient practical implementation, a pre-calculation is made based on the previously defined attributes. The entire dataset contains 129,675 entries. For the implementation, however, only all possible combinations of the attribute selection are relevant, including cases in which the value of an attribute is not known and is therefore not specified more precisely. This results in 135 combinations which can be individually applied to each single architectural element.

For each of the possible combinations, the affected entries are determined from the dataset and the values required for the implementation are extracted. This primarily focuses on the weaknesses covered and the relative frequency of these. We assume that weaknesses that occur frequently within the dataset are also likely to occur more frequently in real applications, so we will use this relative frequency as a way to prioritize weaknesses. For all vulnerabilities that are based on a weakness, we also calculate the average values of the CVSSv2 base, impact, and exploitability metrics in order to have another means of prioritization, which is not based on the probability of a vulnerability, but on its impact and the resulting risks. In order to provide a simple measure of these values, the average values were then grouped into the categories low, medium and high.

It should be noted that due to the linear relationship between the attributes and the related threats in this type of implementation, it is possible that for some combinations no entries can be found (as the specified combination does not appear in the dataset). This is even more likely to happen if a large number of attributes of a DFD element are specified. Currently, this restriction is especially present when specifying the used programming language, as the CWE dataset does not necessarily list all affected programming languages in every case.


Element Attributes

Access Vector: Network
Authentication: True
Language: Unknown
Technology: Unknown

%	CWE Name	S	T	R	I	D	E	Threat Description	Base	Impact	Exploitability
20.28	Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	✓	✓	-	✓	✓	✓	The software does not neutralize or incorrectly neutralizes user-controllable input before it is placed in output that is used as a web page that is served to other users.	low	low	high
19.04	Unknown Weakness	-	-	-	✓	-	-	No appropriate CWE was given for these entries, therefore the shown STRIDE threats were determined by performing a text classification.	medium	medium	high
7.47	Improper Input Validation	✓	✓	-	✓	✓	✓	The product does not validate or incorrectly validates input that can affect the control flow or data flow of a program.	medium	medium	high
7.05	Permissions, Privileges, and Access Controls	✓	-	-	✓	-	✓	Weaknesses in this category are related to the management of permissions, privileges, and other security features that are used to perform access control.	medium	medium	high
6.96	Information Exposure	-	-	-	✓	-	-	An information exposure is the intentional or unintentional disclosure of information to an actor that is not explicitly authorized to have access to that information.	medium	low	high
5.91	Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')	-	✓	-	✓	-	-	The software constructs all or part of an SQL command using externally-influenced input from an upstream component, but it does not neutralize or incorrectly neutralizes special elements that could modify the intended SQL command when it is sent to a downstream component.	medium	medium	high

Fig. 3. Example of attribute selection in OVVL and individually identified threats

6 Summary and Conclusion

A general problem with automated threat modelling tools is to define the underlying threat catalogue [12]. This in turn leads to reported threats being perceived as too general (when just using plain STRIDE) and containing unnecessary duplicates as well as having no means of prioritizing reported threats [13].

Using STRIDE [1, 2] as a mapping technique, we reconciled the CWE and CVE databases to derive a threat catalogue that is used in the OVVL framework to address these concerns. A substantial part of this mapping (40%, ~50.000 records) was done on basis of a supervised machine-learning based text classification. We provided a detailed discussion on text (pre-)processing, model selection (resulting in adoption of LightGBM) as well as evaluation of the trained model. Compared to plain STRIDE-based threat modelling we demonstrated that our approach identifies more specific real-world threats and thus facilitate that more appropriate countermeasures can be defined.

We demonstrated how this derived threat catalogue can be queried on basis of attributes such as access vector or authentication for each single element of a threat model within the OVVL framework. This results in individually reported threats avoiding duplicates as well as the possibility to prioritize these by evaluating the associated impact and exploitability attributes as well as observed frequency of occurrence (i.e. related vulnerability) in the real world.

This approach also complements the already existing feature of OVVL to identify concrete CPEs and CVEs in case it is also known at system design time how a component is realized.

Though we can now automatically prioritize threats this does not necessarily exclude false positives (i.e. reported threats an experienced analyst will rule out on basis of information not captured in the threat model). Future work will thus focus on conducting user studies to understand possible correlations between prioritized threats and whether an analyst will push these as an issue into the project management tool/product backlog. OVVL already exhibits an OpenAPI interface to communicate with such tools in the later stages of the development lifecycle.

We are not aware of other directly related work, though vulnerability information extraction has been discussed in [19]. In [20, 21] a security risk analysis is performed on formalized UML models in combination with OCL rules which is at a much lower level of abstraction than our work. In fact, there is no indication on basis of what dataset risks are identified and the provided examples appear to be manually constructed. Contrary to that, [22] shows how attack trees are generated from vulnerability databases such as CVE. However, this is only done for network related attacks and the authors do not detail how they obtain the required environmental knowledge to analyse the CVE database.

We now want to work on further improving our approach by identifying other additional metadata a threat analyst could augment a system model with. We believe this will provide us with further capabilities to identify relevant threats, as it will allow us to prioritize based on actual software usage rather than the frequency of existing vulnerabilities. We also did already experiment with using data feeds generated by Shodan. Such real-world data also gives us much more information about the environment in use, allowing us to (further) tailor the identification of threats to the specifics of the environment and reconcile this with our DFD-based system model. However, the resulting increased amount of information that goes into modelling would then become too complex to produce a linear relationship between the input values and the threats. We therefore plan to also use machine-learning techniques for this purpose, which we hope will allow our model to establish a much deeper relationship between threats, vulnerabilities and the environment and enabling circumstances in which they may occur.

References

1. Shostack, A.: Threat Modeling: Designing for Security. Wiley, Hoboken (2014)
2. Shevchenko, et al.: Threat modeling: a summary of available methods. Software Engineering Institute, CMU (2018). https://resources.sei.cmu.edu/asset_files/WhitePaper/2018_019_001_524597.pdf. Accessed 25 Feb 2020
3. Khan, et al.: STRIDE-based threat modeling for cyber-physical systems. In: IEEE PES: Innovative Smart Grid Technologies Conference Europe (2017)
4. https://www.microsoft.com/en-us/securityengineering/sdl/threatmodeling. Accessed 25 Feb 2020
5. https://docs.microsoft.com/de-de/azure/security/develop/threat-modeling-tool. Accessed 25 Feb 2020
6. Schaad, A., Borozdin, M.: TAM2: Automated threat analysis. ACM SAC 2012
7. OWASP Threat Dragon. https://threatdragon.org/login. Accessed 25 Feb 2020
8. IriusRisk. https://iriusrisk.com/threat-modeling-tool/. Accessed 25 Feb 2020
9. ThreatModeler. https://threatmodeler.com/. Accessed 25 Feb 2020
10. Schaad, A., Reski, T.: Open weakness and vulnerability modeler (OVVL): an updated approach to threat modeling. In: ICETE (2), pp. 417–424 (2019)
11. Schaad, A.: Project OVVL - Threat Modeling Support for the entire secure development lifecycle. In: Sicherheit 2020, pp. 121–124 (2020)
12. Berger, B.J., Sohr, K., Koschke, R.: Automatically extracting threats from extended data flow diagrams. In: Caballero, J., Bodden, E., Athanasopoulos, E. (eds.) ESSoS 2016. LNCS, vol. 9639, pp. 56–71. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-30806-7_4
13. Sion, et al.: Risk-based design security analysis. In: 1st International Workshop on Security Awareness from Design to Deployment, Sweden (2018)
14. Jupyter Notebook. https://github.com/OVVL-HSO/Threat-Catalogue
15. CWE Database. https://cwe.mitre.org/. Accessed 25 Feb 2020
16. CVE Database. https://cve.mitre.org/. Accessed 25 Feb 2020
17. Sonarqube. https://www.sonarqube.org/. Accessed 25 Feb 2020
18. LightGBM. https://lightgbm.readthedocs.io/en/latest/. Accessed 25 Feb 2020
19. Weerawardhana, S.S., et al.: Automated extraction of vulnerability information for home computer security. In: FPS 2014, pp. 356–366 (2014)
20. Almorsy, M., et al.: Automated software architecture security risk analysis using formalized signatures. In: Proceedings of the 2013 International Conference on Software Engineering, pp. 662–671. IEEE Press (2013)
21. Basin, D., et al.: Automated analysis of security-design models. Inf. Softw. Technol. **51**(5), 815–831 (2009)
22. Birkholz, H., et al.: Efficient automated generation of attack trees from vulnerability databases. In: Working Notes for the 2010 AAAI Workshop on Intelligent Security (SecArt), pp. 47–55 (2010)