Towards Automatically Generating Security Analyses from Machine-Learned Library Models

Maria Kober[0000−0001−9560−1527] and Steven Arzt[0000−0002−5807−9431]

Fraunhofer Institute for Secure Information Technology, Darmstadt, Germany
{maria.kober, steven.arzt}@sit.fraunhofer.de

Abstract. Automatic code vulnerability scanners identify security antipatterns in application code, such as insecure uses of library methods. However, current scanners must regularly be updated manually with new library models, patterns, and corresponding security analyses. We propose a novel, two-phase approach called Mod4Sec for automatically generating static and dynamic code analyses targeting vulnerabilities based on library (mis)usage. In the first phase, we automatically infer semantic properties of libraries on a method and parameter level with supervised machine learning. In the second phase, we combine these models with high-level security policies. We present preliminary results from the first phase of Mod4Sec, where we identify security-relevant methods, with categorical f1-scores between 0.81 and 0.93.

Keywords: Vulnerability scanner · Vulnerability detection · Security analysis · Specialized domain language · Automated analysis · Mod4Sec

1 Introduction and Motivation

Implementation flaws are prevalent in software. While modern platforms and operating systems offer powerful APIs for sensitive security operations such as encryption or authentication, developers often misuse these APIs, leading to numerous vulnerabilities [2, 1]. Manual approaches like penetration tests or code reviews can detect such vulnerabilities but require substantial effort. They are therefore unsuitable for agile development processes with fast release cycles [9]. Automated code scanning [4, 13, 16, 3, 11, 15] can identify known antipatterns in code, but requires the respective patterns. If a scanner has no support for the respective API, i.e., no antipatterns to look for, the security vulnerability remains undetected. Such scanners need to be updated regularly with new antipatterns, which is largely a manual effort. For example, on popular platforms like Android more than 13,700 third-party libraries are used in apps additional to the Java Standard Library and Android SDK, according to our pre-study on 9,373 apps from the 2020 and 2021 Google Play Store. If only considering libraries that are used in at least 10% of Android apps, a code scanner must be kept up-to-date with 65 libraries, yet vulnerabilities regarding other libraries would remain undetected. New library versions may become available in frequencies of several days to months [17, 5], leading to constant requirements for manual maintenance.

In this paper, we present our idea and vision for Mod4Sec, a novel approach for automatically inferring semantic models of individual libraries and platforms, i.e., API specifications, and linking them to security policies for program analyses. Library-agnostic security properties such as "cryptographic keys must not be hard-coded" change rarely. APIs, on the other hand, evolve. We therefore propose to automatically infer specific library models (e.g., "second parameter of method `encrypt()` in class A in library B is a cryptographic key") and match them to library-independent security properties (e.g., "RSA keys must be of 2048 bits or more"). This mapping is then used to automatically generate antipatterns for static and dynamic code analyses. These antipatterns are further processed to generate analysis code, so that scanners can detect violations of the security properties in apps that use libraries for which a model has been inferred.

This paper is organized as follows. Section 2 describes details of our vision. Section 3 presents details on and results of first experiments. Section 4 presents related work. Section 5 concludes this paper and points out future work.

2 Vision

Our vision is to automatically generate semantic models for libraries and platforms using machine learning, and to use these models to link generic security assets (keys, passwords, certificates, etc.) to concrete APIs. Thus, we are able to automatically generate static and dynamic code analyses, thereby reducing manual effort for tool developers and security analysts.

The core idea behind Mod4Sec is that security properties are derived from only a handful of core assets and concepts that usually remain unchanged for years. For example, the concept of a password is the same, regardless of how this password is used (e.g., for authentication or deriving an encryption key using a KDF). Likewise, authentication methods that consume passwords are equivalent from the point of a security scanner, regardless of the target of authentication, even though the APIs may look differently. We therefore propose to identify these generic concepts in implementations and documentations of APIs using machine learning and natural language processing.

In our proposed workflow, a security analyst only needs to reason about problems within the domain of security, fully abstracting from the concrete implementation of applications and programming libraries. The analyst specifies properties in terms of abstract domain knowledge such as "MD-4 must not be used as cryptographic hash function". The link back to code, i.e., identifying what should be checked within an application, is done using the auto-generated library models. As these models are auto-generated, they can easily be updated by the tool developer when new or updated libraries are available.

Figure 1 shows an overview of our approach, which consists of two phases. In the first phase, we generate the library models as explained in Section 2.1. In the second phase, we match them with security policies to generate static and dynamic analyses as explained in Section 2.2. Additionally, developers can give feedback on the analysis results, increasing the accuracy of the model over time.

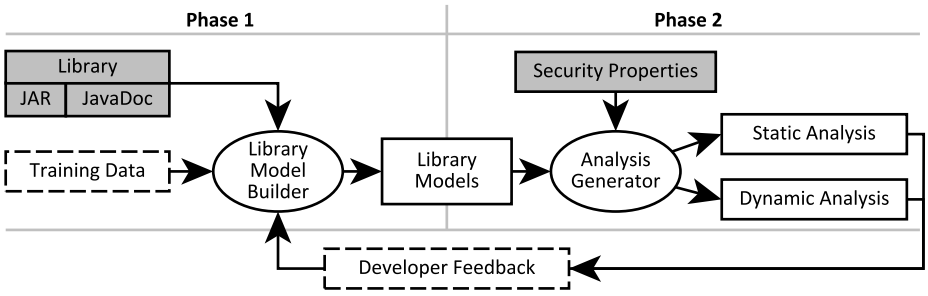

Fig. 1. The two phases of our vision. Input data to generate security analyses is denoted in gray. Data required solely for training is marked by dashed lines.

2.1 Phase 1: Generate Library Models

In the first phase, MOD4SEC uses supervised machine learning to automatically associate library methods and method parameters with pre-defined categories, e.g., "encryption" or "authentication" for methods, and "username", "crypto key" or "filename" for parameters. The learning algorithm learns on a set of implementation JAR files and the corresponding JavaDoc JARs. The resulting *Generic Domain Model*, an integral part of the *Library Model Builder* in Figure 1, generalizes over all input libraries and is therefore library-agnostic. It can be re-used for classification whenever new or updated libraries become available. Thus, obtaining high-quality training data and training the model is a rare task, if not a one-time effort. Note that we train two different classifiers for method and parameter classification, thereby reducing complexity, as the set of parameter labels is category-specific, e.g., "crypto key" is not within "file access methods".

When classifying a concrete library, MOD4SEC uses the Generic Domain Model and takes the respective library as input. We call the resulting library-specific model, which describes the semantics of the individual methods and parameters inside the library, a *Library Model*. Thus, utilizing MOD4SEC's generic domain model results in one model per provided library and library version, which can seamlessly be integrated into code scanners and used with analyses.

2.2 Phase 2: Generate Security Analyses

In the second phase, MOD4SEC uses the library models to automatically generate static and dynamic code analyses for previously defined security policies. The key idea is to encode security properties in a language that can be mapped to specific functionalities in a particular analysis framework such as Soot [8], and to properties of the library models generated in phase 1 of MOD4SEC.

Our proposed description language SECPLANG is declarative, i.e., it defines which properties should be checked. How properties are evaluated is up to the specific analysis framework. SECPLANG contains basic logic operations, and a small set of predefined functions and predicates, e.g., `has(x)` to check whether a collection `x` contains items, and $\texttt{value}_\texttt{s}\texttt{(y)}$ resp. $\texttt{value}_\texttt{d}\texttt{(y)}$ to statically resp. dynamically retrieve all values of parameter `y` of a program statement. We further

Table 1. Categorization results on three security categories.

Category	Precision	Recall	F1-Score	# Samples
Cryptography	0.92	0.93	0.93	320
Authentication	0.83	0.78	0.81	161
Network/TLS	0.81	0.81	0.81	303
Other	0.99	0.99	0.99	10,412

define one auto-generated placeholder for each method parameter label of the library models, e.g., `p-key` references parameters labeled as cryptographic keys.

For example, the policy "crypto keys must not be hard-coded" can be encoded as `encryption[not has (value$_s$ (p-key))]` for static analyses. During the analysis, all API calls that are of type `encryption` according to the library models of phase 1 are collected. The placeholders in the Boolean formula are replaced with the respective statement parameters. If the declaration of the API method is, e.g., `encrypt(input, key, algorithm)`, and it is called as `encrypt(a, b, c)` in the code under analysis, the placeholder `p-key` in the Boolean formula of the security property is replaced with `b` because the second argument is the cryptographic key according to the library model. Evaluating the Boolean formula only requires invoking the correct building blocks of the analysis framework. In the example, a value analysis is invoked on variable `b`, modeled as predefined `value`$_s$-function in the formula. The predefined `has`-function checks whether `value`$_s$ returned at least one value, i.e., there is a hard-coded key.

3 Experiments and Preliminary Results

To evaluate the feasibility of our approach, we implemented a prototype for the first phase of Mod4Sec as described in Section 2.1. From the Maven central repository, we obtained cryptography, network, and authentication libraries. We hand-annotated 784 security-relevant methods in the Java Standard Library and in 11 relevant third-party libraries (altogether 3x cryptography, 5x network, 6x authentication), omitting method parameter annotation. We extracted JavaDoc and signatures for 11,196 methods with JavaParser. For supervised machine learning, we transformed input data to bags-of-words and used scikit-learn [12] with one fully connected neural network having one hidden layer with 16 nodes.

Table 1 presents the results of a stratified ten-fold cross-validation on our data set. The precision of Mod4Sec is between 0.81 and 0.99 for different categories. The recall is between 0.78 and 0.99, with an f1-score between 0.81 and 0.99. In total, more than three-quarters of all methods are categorized correctly and we are able to correctly identify a high number of security-relevant methods, even when those are a minority in the analyzed code.

To assess whether finer-grained categories are beneficial, we divided the `cryptography` category into several subcategories as shown in Table 2[1]. All subcategories have a precision between 0.74 and 1.0, and a recall between 0.71

[1] Due to space limitations, we only include the most relevant subcategories. Hash and MAC APIs, for example, have fewer methods in our sample set.

Table 2. Results for five subcategories of cryptography APIs.

Category	Precision	Recall	F1-Score	# Samples
Cipher-Configuration	0.74	0.90	0.81	61
Crypt. Randomness	1.00	0.88	0.93	8
En- & Decryption	0.75	0.71	0.73	21
Key Creation	0.89	0.86	0.88	125
Signatures	0.84	0.72	0.78	29

and 0.9. These numbers indicate that it is possible to correctly distinguish individual cryptographic functionalities in libraries, making an essential part of the first phase of Mod4Sec applicable for software analysis.

Note that a simple keyword-search is not sufficient for our purpose. For example, the keyword "key" is present, amongst others, in the context of cryptographic keys, but also in key-value pairs in maps and object builders. When searching through our dataset for "key", we found 903 methods, which is almost thrice as much as there are cryptographic methods of interest and about six times the number of methods related to cryptographic keys, encryption, or decryption.

4 Related Work

Checking applications for specific misuses of security-sensitive APIs is common practice [13, 16, 3, 15]. CogniCrypt [6, 7] further assists developers with automatic code generation for using such APIs. FixDroid [11] evaluates code snippets against a database of known insecure code. PQL [10] allows for declarative queries on code the analyst is familiar with. All of these approaches rely on manually assembled patterns for APIs. In contrast, we propose a general approach for automatically generating analysis rules, based on a library-agnostic semantic domain model. $\text{SWAN}_{\text{ASSIST}}$ [14] could be integrated in step 1 of Mod4Sec. It utilizes developer feedback to actively learn security-relevant methods.

5 Conclusion and Future Work

In this paper, we have presented our vision of Mod4Sec for automatically generating static and dynamic code analyses from generic security properties described in our newly introduced API-agnostic language SecPLang. Mod4Sec uses machine learning for automatically generating library models, which then allow it to map generic security properties to concrete APIs. We presented first experimental results to demonstrate the feasibility of the first phase of our vision.

As future work, we will provide a full implementation and evaluation of our vision, including automatic classification for method parameters using machine learning. We plan to re-evaluate our preliminary results on a larger dataset and aim to increase Mod4Sec's precision and recall. Furthermore, we will extend Mod4Sec with additional categories of methods. We plan to formally describe SecPLang and extend its scope to more complex security properties. We plan to build an implementation of the analysis generator for static and dynamic analyses on top of the Soot program analysis framework [8].

References

1. Chatzikonstantinou, A., Ntantogian, C., Karopoulos, G., Xenakis, C.: Evaluation of Cryptography Usage in Android Applications. In: Proceedings of the 9th EAI International Conference on Bio-inspired Information and Communications Technologies (formerly BIONETICS). pp. 83–90 (2016). https://doi.org/10.4108/eai.3-12-2015.2262471
2. Egele, M., Brumley, D., Fratantonio, Y., Kruegel, C.: An Empirical Study of Cryptographic Misuse in Android Applications. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 73–84 (2013). https://doi.org/10.1145/2508859.2516693
3. Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B., Smith, M.: Why eve and mallory love android: An analysis of android ssl (in) security. In: Proceedings of the 2012 ACM conference on Computer and communications security. pp. 50–61 (2012). https://doi.org/10.1145/2382196.2382205
4. Felderer, M., Büchler, M., Johns, M., Brucker, A.D., Breu, R., Pretschner, A.: Security testing: A survey. In: Advances in Computers, vol. 101, pp. 1–51. Elsevier (2016). https://doi.org/10.1016/bs.adcom.2015.11.003
5. Ihara, A., Fujibayashi, D., Suwa, H., Kula, R.G., Matsumoto, K.: Understanding when to adopt a library: A case study on ASF projects. In: IFIP International Conference on Open Source Systems. pp. 128–138. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-57735-7_13
6. Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., Göpfert, F., Günther, F., Weinert, C., Demmler, D., et al.: Cognicrypt: Supporting developers in using cryptography. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 931–936. IEEE (2017)
7. Krüger, S., Späth, J., Ali, K., Bodden, E., Mezini, M.: Crysl: validating correct usage of cryptographic apis. arXiv preprint arXiv:1710.00564 (2017)
8. Lam, P., Bodden, E., Lhotak, O., Hendren, L.: The soot framework for java program analysis: a retrospective. In: Cetus Users and Compiler Infastructure Workshop (CETUS 2011) (Oktober 2011)
9. Maqsood, H.M., Bondavalli, A.: Agility of Security Practices and Agile Process Models: An Evaluation of Cost for Incorporating Security in Agile Process Models. In: ENASE 2020. pp. 331–338 (2020). https://doi.org/10.5220/0009356403310338
10. Martin, M., Livshits, B., Lam, M.S.: Finding application errors and security flaws using pql: a program query language. Acm Sigplan Notices **40**(10), 365–383 (2005)
11. Nguyen, D.C., Wermke, D., Acar, Y., Backes, M., Weir, C., Fahl, S.: A stitch in time: Supporting android developers in writingsecure code. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 1065–1077 (2017). https://doi.org/10.1145/3133956.3133977
12. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E.: Scikit-learn: Machine learning in Python. Journal of Machine Learning Research **12**, 2825–2830 (2011)
13. Piccolboni, L., Di Guglielmo, G., Carloni, L.P., Sethumadhavan, S.: Crylogger: Detecting crypto misuses dynamically. arXiv preprint arXiv:2007.01061 (2020)
14. Piskachev, G., Do, L.N.Q., Johnson, O., Bodden, E.: Swan_assist: Semi-automated detection of code-specific, security-relevant methods. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1094–1097. IEEE (2019). https://doi.org/10.1109/ASE.2019.00110

15. Saccente, N., Dehlinger, J., Deng, L., Chakraborty, S., Xiong, Y.: Project achilles: A prototype tool for static method-level vulnerability detection of java source code using a recurrent neural network. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW). pp. 114–121. IEEE (2019). https://doi.org/10.1109/ASEW.2019.00040
16. Shuai, S., Guowei, D., Tao, G., Tianchang, Y., Chenjie, S.: Modelling analysis and auto-detection of cryptographic misuse in android applications. In: 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing. pp. 75–80. IEEE (2014). https://doi.org/10.1109/DASC.2014.22
17. Suwa, H., Ihara, A., Kula, R.G., Fujibayashi, D., Matsumoto, K.: An Analysis of Library Rollbacks: A Case Study of Java Libraries. In: 2017 24th Asia-Pacific Software Engineering Conference Workshops (APSECW). pp. 63–70. IEEE (2017). https://doi.org/10.1109/APSECW.2017.25