Faculty of Computer Science, Electrical Engineering and Mathematics
Department of Computer Science
Research Group Secure Software Engineering

Helping Java Developers Reduce Cryptographic API Misuses

Michael Schlichtig

Dissertation

Submitted in partial fulfillment of the requirements for the degree of

Doktor der Naturwissenschaften (Dr. rer. nat.)

Advisor

Prof. Dr. Eric Bodden

Paderborn, August 26, 2026

Michael Schlichtig
Helping Java Developers Reduce Cryptographic API Misuses
Dissertation, August 26, 2026
Advisor: Prof. Dr. Eric Bodden

Paderborn University
Research Group Secure Software Engineering
Department of Computer Science
Faculty of Computer Science, Electrical Engineering and Mathematics
Warburger Straße 100
33098 Paderborn

Abstract

Software systems play an integral role in modern society, e.g., in communication, healthcare, finance, and education. To ensure confidentiality, integrity, and availability, developers rely on cryptographic Application Programming Interfaces (APIs). However, developers often struggle to use cryptographic APIs securely as they are complex and require expert knowledge for secure configuration. Even small misuses of cryptographic APIs can lead to severe vulnerabilities. Static analysis tools can support developers by checking source code for security-relevant properties without executing it. In practice, however, their impact is often limited by high rates of false positives and usability problems, including a lack of actionable explanations in reports and limited support for remediation.

This thesis investigates how static analysis tools can better support developers in reducing cryptographic API misuses by improving usability, explainability, and handling of false positives. It provides empirical and conceptual foundations for developing prototypes of tool support. These foundations include usability criteria and developer needs, adoption challenges, and the representation, explanation, and evaluation of cryptographic API misuse. These foundations are then used to develop *SecAI*, the final prototype of this thesis, systematically.

This thesis identifies 36 usability criteria and maps them to six categories: *Warning Messages*, *Fix Support*, *False Positives*, *User Feedback*, *Workflow Integration*, and *User Interface*. These criteria show that usable static analysis tools require both sophisticated analysis algorithms that identify important information and graphical user interfaces that present it in an actionable way. To increase the explainability of *Warning Messages*, this thesis presents *FUM*, a framework for API misuse classification, and empirical studies on cryptographic API misuses. It also contributes *CamBench*, a benchmark for evaluating static analysis tools for cryptographic API misuse detection. Finally, *LLM-SAST*$_{Repair}$, a prototype for *Fix Support*, and *FP*$_{Predictor}$, a prototype for *False Positive* detection, are integrated as features in *SecAI*. This usability-focused tool helps developers reduce cryptographic API misuse.

Overall, this thesis shows that helping developers reduce cryptographic API misuse via tool support requires a holistic approach. Usability features and analysis capabilities must be designed together, since usable interfaces depend on the information

provided by the underlying analysis. *SecAI*, which was technically assessed, demonstrates this through eight usability features that address all six identified usability categories. It also provides a demonstrator for future developer studies to evaluate the individual and combined impact of these usability features.

Zusammenfassung

Softwaresysteme spielen in der modernen Gesellschaft eine zentrale Rolle, z. B. in den Bereichen Kommunikation, Gesundheitswesen, Finanzwesen und Bildung. Um Vertraulichkeit, Integrität und Verfügbarkeit zu gewährleisten, greifen Entwickler auf kryptografische Anwendungsprogrammierschnittstellen (APIs) zurück. Allerdings haben Entwickler oft Schwierigkeiten, kryptografische APIs sicher zu nutzen, da diese komplex sind und für eine sichere Konfiguration Fachwissen erfordern. Selbst geringfügige Fehlanwendungen kryptografischer APIs können zu schwerwiegenden Sicherheitslücken führen. Statische Analysewerkzeuge können Entwickler unterstützen, indem sie den Quellcode auf sicherheitsrelevante Eigenschaften überprüfen, ohne ihn auszuführen. In der Praxis ist ihre Wirksamkeit jedoch oft durch hohe False-Positive-Raten und Usability-Probleme eingeschränkt, darunter ein Mangel an praxistauglichen Erklärungen in Berichten und eine begrenzte Unterstützung bei der Behebung von Problemen.

Diese Arbeit untersucht, wie statische Analysewerkzeuge Entwickler besser dabei unterstützen können, den Missbrauch kryptografischer APIs zu reduzieren, indem sie die Benutzerfreundlichkeit, die Erklärbarkeit und den Umgang mit False Positives verbessern. Sie liefert empirische und konzeptionelle Grundlagen für die anschließende Entwicklung von Prototypen zur Werkzeugunterstützung. Zu diesen Grundlagen gehören Kriterien zur Benutzerfreundlichkeit und Entwicklerbedürfnisse, Herausforderungen bei der Einführung sowie die Darstellung, Erklärung und Bewertung des Missbrauchs kryptografischer APIs. Auf dieser Grundlage wird systematisch *SecAI* entwickelt, der abschließende Prototyp dieser Arbeit.

In dieser Arbeit werden 36 Usability-Kriterien identifiziert und sechs Kategorien zugeordnet: *Warning Messages*, *Fix Support*, *False Positives*, *User Feedback*, *Workflow Integration* und *User Interface*. Diese Kriterien zeigen, dass benutzerfreundliche Tools zur statischen Analyse sowohl ausgefeilte Analysealgorithmen zur Erkennung wichtiger Informationen als auch grafische Benutzeroberflächen erfordern, die diese Informationen in umsetzbarer Form darstellen. Um die Erklärbarkeit von *Warning Messages* zu verbessern, stellt diese Arbeit *FUM* vor, ein Framework zur Klassifizierung von API-Missbrauch, sowie empirische Studien zu Missbrauch kryptografischer APIs. Außerdem wird *CamBench* vorgestellt, ein Benchmark zur Bewertung von statischen Analysewerkzeugen für die Erkennung von Missbrauch kryptografischer APIs. Schließlich werden *LLM-SAST*$_{Repair}$, ein Prototyp für *Fix Support*, und *FP*$_{Predictor}$, ein Prototyp zur Erkennung von *False Positives*, als Funktionen in *SecAI* integriert, einem auf Benutzerfreundlichkeit ausgerichteten Werkzeug, das Entwickler dabei unterstützt, den Missbrauch kryptografischer APIs zu reduzieren.

Insgesamt zeigt diese Arbeit, dass die Unterstützung von Entwicklern bei der Reduzierung von Missbrauch kryptografischer APIs durch Tool-Unterstützung einen ganzheitlichen Ansatz erfordert. Usability-Funktionen und Analysefähigkeiten müssen gemeinsam entworfen werden, da benutzerfreundliche Oberflächen von den Informationen abhängen, die durch die zugrunde liegende Analyse bereitgestellt werden. *SecAI*, welches technisch bewertet wurde, demonstriert dies mit acht Usability-Funktionen, die alle sechs identifizierten Usability-Kategorien abdecken. Es bietet zudem einen Demonstrator für zukünftige Entwicklerstudien, um die individuellen und kombinierten Auswirkungen dieser Usability-Funktionen zu untersuchen.

Acknowledgments

Throughout the adventure of my studies and dissertation research, I have had a lot of experiences, faced many challenges, and had the pleasure of meeting incredible people, some of whom have become dear friends. I made many memories and grew during my time in the SSE group and CROSSING. Trying to express my gratitude to everyone while keeping this thank-you note reasonably short is difficult. Even though I will not mention everyone by name, I am very grateful to all of you who have supported me on this journey!

First, I am grateful to my advisor, Eric Bodden. Your support, advice, and understanding went above and beyond, helping me navigate the challenges of this phase of my life.

Thank you, Anna-Katharina Wickert, so much for your advice and patience. We built a lot on the foundations laid by those who researched prior to us. Special thanks go to Stefan Krüger, for your excellent dissertation, which paved the way for my research, and for helping me in the beginning. This was made possible by CROSSING, especially thanks to its wonderful people. CROSSING as a community was exactly what I needed when I joined. Thank you so much for the positive energy and amazing collaboration retreats, Stefanie Kettler, Daniela Fleckenstein, Maximilian Tippmann, Kilian Demuth, and Frank Nelles.

Markus Schmidt, thank you for working with me and for being someone I can count on. You help me bring my ideas to life. Together, we have supervised many projects, students, and the project group. To all of our students, thank you for your contributions and for learning together: Tom, Carina, Yashik, Aniruddh, Mohit, Dharmik, Hazem, Roshan, Samarth, Akshaya, Florian, Aashish, Sven, Tim, Sven, Lukas, Marvin, David, Sharzad, Enri, Seena, Taslima, Anemone, André, Rakshit, Santosh, Shashi, Nihad, Jonas, Niklas, Tim, Lukas, Steffen, and Daniel.

Joining your research, Mugdha Khedkar, taught me a lot, and I had a lot of fun while doing my part and testing my ideas through various prototypes.

I want to thank all my co-authors; I have learned a great deal from our collaboration, which made all of this possible. I am also grateful to the anonymous reviewers, whose feedback helped improve the quality of my research. I want to thank the

scientific community, where I have met so many helpful, open-minded, and kind people.

Thank you for joining my defense committee: Rodrigo Bonifácio, Mohamed Soliman, Carsten Schulte, and Harald Selke.
Rodrigo and Luis Amaral, I am very grateful for our collaboration and your hospitality during my research visit to Brazil. Thank you to everyone I met during this visit. I will cherish all these experiences forever. Mohamed, I am thankful for the opportunity to learn from you through our discussions and collaborations. Carsten, thank you; I gained a lot of experience in a new field of research during my first year of doing a PhD with you. Harald, thank you for your thoughtful advice.

I also want to thank all my colleagues in the SSE group and Fraunhofer. When I first started, Jan Martin Persch, Lisa Nguyen Quang Do, Stefan, Martin Mory, Marcus Nachtigall, Philipp Schubert, Ben Hermann, and Goran Piskachev helped me get settled. Thanks to my office, Jan, Marcus, Anna Lena Rotthaler, Markus, and Marcus Hüwe, for the great atmosphere and mutual support. Thanks to Stefan Schott, Jonas Klauke, Kadiray Karakaya, Ashwin Prasad Shivarpatna Venkatesh, Sriteja Kummita, Aashish Prajapati, Palaniappan Muthuraman, and Fabian Schiebel for many years of working together and making fun memories. And to my colleagues who have joined more recently, thanks for all the good moments and coffee breaks we have had in the meantime; I wish you all a great experience pursuing your PhDs!

I want to thank everyone who helped me with the organizational tasks and paperwork: Nicole Graskamp, Sammy Maniera, Vera Meyer, and Conny Wiederhold. I would also like to thank my former research groups, the DDI and the Kontextuelle Informatik.

This research was made possible by funding from the DFG as part of the Collaborative Research Center 1119 CROSSING, from the BMBF as part of the Software Campus program, and from the Heinz Nixdorf Institute.

Last but not least, I want to thank my friends and family and everyone who rooted for me. You supported me on this journey and way before! Many thanks to my parents, who have given me so many opportunities to follow my own life goals. And thanks to my sister, who showed me different ways to navigate college.

Michael Schlichtig

Publications

This dissertation presents original contributions, of which many have already been published in conference and workshop papers. For the publications listed in this chapter, the author of this thesis is also the lead author or co-author. This dissertation is the result of a collaborative effort — the contributions of the author are explicitly stated per publication. In particular, this includes the following work; the main contributions claimed for this thesis are highlighted:

- **Chapters:** Chapters 1 and 2 make use of all papers in this list. Parts of these chapters are taken from the contribution to the Doctoral Symposium at ICSE'24.

  Michael Schlichtig. "Building a Framework to Improve the User Experience of Static Analysis Tools". In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. ICSE-Companion '24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 165–169 [Sch24]

  Content: This paper for the doctoral symposium discusses the research goal of this thesis in helping Java developers reduce API misuse by building usable static analysis tools. It provides an overview of the research performed in this thesis.

  Contribution: I summarized my PhD research for the doctoral symposium.

- **Chapters:** Chapter 3

  Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. "A large-scale study of usability criteria addressed by static analysis tools". In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 532–543 [NSB22] (***Contribution 1***)

  Content: In this paper, we conducted a systematic, large-scale study to assess the state of the art in the usability criteria addressed by static analysis tools. Based on the existing literature, we defined our study design, including inclusion and exclusion criteria for static analysis tools, predefined evaluation

metrics, and the study protocol employing a multiple-raters approach. We compiled a list of existing and recommended static analysis tools. After applying the inclusion and exclusion criteria, we installed all static analysis tools on instances of the same standardized virtual machine (one per operating system). Both raters examined each tool independently. Results were compared and discussed until agreement was reached. We concluded the study by discussing the results in six main categories of usability criteria.

Contribution: Marcus Nachtigall had the idea for this study as part of his research on the explainability of static analysis results. I actively designed the study, the metrics to assess and their scoring, the study protocol, and the study setup, e.g., the virtual machines. Marcus and I conducted the empirical study, e.g., by setting up static analysis tools in the virtual machines and using the multiple-raters approach: first independently, then in agreement discussions. Lastly, we evaluated and discussed the results. I contributed to the paper, especially to the visualization of results, and to the review and editing process.

- **Chapters:** Chapter 4

Michael Schlichtig, Steffen Sassalla, Krishna Narasimhan, and Eric Bodden. "FUM - A Framework for API Usage constraint and Misuse Classification". In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022, pp. 673–684 [Sch+22a] (***Contribution 2***)

Content: In this paper, we investigated our hypothesis that *CrySL* [Krü+19] and *CogniCrypt* [Krü+19] could be used to specify and detect misuses of non-cryptographic APIs. After initial testing, we found that this was not the case and therefore examined the reasons. We conducted a systematic literature review of API misuse and its origins. The results yielded *FUM*, a new ***F**ramework for **U**sage Constraint and **M**isuse classification*. We evaluated *FUM* in a case study to determine which *FUM* types *CogniCrypt* and *CrySL* can specify and detect, and which directed extensions are reasonable to increase *CogniCrypt's* coverage of *FUM* types.

Contribution: I proposed investigating the generalizability of *CogniCrypt* and *CrySL* after taking over the maintenance and development of *CogniCrypt* in CRC 1119 CROSSING. I led this project, which began with supervising Steffen Sassalla's bachelor's thesis from which the results of this paper were derived. The thesis was co-supervised by Krishna Narasimhan. I created the original paper draft and led the review and editing process.

- **Chapters:** Chapter 5

  Anna-Katharina Wickert, Lars Baumgärtner, Michael Schlichtig, Krishna Narasimhan, and Mira Mezini. “To Fix or Not to Fix: A Critical Study of Crypto-misuses in the Wild”. In: *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2022, pp. 315–322 [Wic+22]
  Parts of Chapter 2 and Section 5.1 are taken from this paper.

  Content: In this paper, we investigated the prevalence of cryptographic API misuses in open-source business projects. So far, research has shown in several studies that high rates, even over 95%, of projects contain at least one cryptographic API misuse. In this study, we investigated the special domain of business projects for which we expected higher quality standards. We used the static analysis tool *CogniCrypt* [Krü+19] to analyze a dataset of business projects for cryptographic API misuse and observed at least one misuse in over 88% of projects. We further extended a risk model to classify the detected misuses. To validate our analysis results, we manually sampled and rated 157 detected misuses, classifying them according to our extended threat model.

  Contribution: This project was led by Anna-Katharina Wickert. I joined the project to manually validate the random sampling of analysis reports from *CogniCrypt*. Here, I assisted in performing and evaluating the manual validation. Part of the manual validation involved discussing the risk model for each finding. I was actively involved in writing, reviewing, and editing the paper.

  Anna-Katharina Wickert, Michael Schlichtig, Marvin Vogel, Lukas Winter, Mira Mezini, and Eric Bodden. “Supporting Error Chains in Static Analysis for Precise Evaluation Results and Enhanced Usability”. In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2024, pp. 693–704 [Wic+24]
  Parts of Chapter 2 and Section 5.2 are taken from this paper.

  Content: In this paper, we investigated the phenomenon of transitive or related errors in the Java Cryptographic Architecture (JCA). Static analysis tools such as *CogniCrypt* [Krü+19] may report multiple warnings for errors caused by the same location as they propagate. We extended the analysis algorithm of *CogniCrypt* to identify and reason about transitive errors. This extension was evaluated on real-world projects crawled from GitHub. The analysis runtime only increased by about 4%, and we identified which JCA classes typically cause error chains. We further conducted an expert interview indicating that

reporting error chains with their causation locations helps developers resolve reported cryptographic API misuses.

Contribution: This project was led by Anna-Katharina Wickert. Together, we supervised the bachelor's theses of Marvin Vogel and Lukas Winter, in which the analysis algorithm extensions were developed, and an expert interview was conducted. I was actively involved in designing the methodology, extending the analysis of *CogniCrypt*, designing the expert interview, and data curation. Further, I was actively involved in writing, reviewing, and editing the paper.

Michael Schlichtig, Anna-Katharina Wickert, Stefan Krüger, Eric Bodden, and Mira Mezini. *CamBench – Cryptographic API Misuse Detection Tool Benchmark Suite*. 2022. arXiv: `2204.06447 [cs.SE]` [Sch+22c]; equal contribution and shared first-authorship with Anna-Katharina Wickert. Registered report with IPA (In-Principal Acceptance) at *MSR'22*[1]. (***Contribution 3***)
Parts of Section 5.3 are taken from this paper.

Content: In this registered report, we identified the need for a domain-specific benchmark for the detection of cryptographic API misuses in Java. We proposed a methodology for its creation based on the scientific literature. In our literature review, we identified several static analysis tools for detecting cryptographic API misuse in Java. However, a lack of a standard benchmark for this domain was observed, as only two static analysis tools were evaluated on the same benchmark. To address this, we proposed *CamBench*, a benchmark consisting of a real-world benchmark element, a synthetic micro-benchmark to evaluate analysis capabilities, and a heuristic for cryptographic API coverage.

Contribution: This project was led by Anna-Katharina Wickert and me as equally contributing shared first authors. My focus in this project was the benchmark generation, especially $CamBench_{CAP}$ for evaluating analysis capabilities and sensitivities. The registered report was accepted at *MSR'22* with IPA (In-Principal Acceptance). I was actively involved in writing the first draft, reviewing, and editing the registered report. For the process of creating *CamBench*, we supervised the master's thesis of Niklas Doppelstein to initially create the synthetic micro-benchmark to evaluate analysis capabilities, named $CamBench_{CAP}$. Based on the findings of this thesis, I reworked and completed the creation of $CamBench_{CAP}$. Further, I supported Anna-Katharina Wickert in the manual effort of creating the real-world benchmark, $CamBench_{REAL}$.

[1] `https://conf.researchr.org/track/msr-2022/msr-2022-registered-reports` (last accessed: 2026-06-06)

Overall, I was actively involved in designing the methodology, evaluating the systematic literature review, benchmark creation, and data curation.

- **Chapters:** Chapter 6

  Michael Schlichtig, Aashish Prajapati, Ashwin Prasad Shivarpatna Venkatesh, Markus Schmidt, Mohamed Soliman, and Eric Bodden. "Can SAST Tools Support LLMs in Automatically Repairing Cryptographic API Misuses?" Unpublished manuscript. 2026 [Sch+26b] (***Contribution 4***)

  Content: In this paper, we evaluated our hypothesis that large language models (LLMs) can be employed in combination with Static Application Security Testing (SAST) tools such as *CogniCrypt* [Krü+19] and *CryptoGuard* [Rah+19] to automatically repair cryptographic API misuses. Based on advances in LLM research, we identified a combination of LLMs and SAST tools as promising for automatic repair. We developed *LLM-SAST*$_{Repair}$ to facilitate automated repair across multiple combinations of LLMs and SAST tools, making it generalizable and extensible. A systematic evaluation of multiple combinations and configurations of LLMs and SAST tools on two benchmarks, *CamBench* [Sch+22c] and *CryptoAPI-Bench* [ARY19], showed that using SAST tools to aid the repair process yields better results. The best-performing configurations used *CogniCrypt* and could increase the benchmarks' security by about 70%, respectively.

  Contribution: I proposed combining LLMs and SAST tools to repair cryptographic API misuses automatically. I led this project, which began with supervising Aashish Prajapati's master's thesis, during which he developed *LLM-SAST*$_{Repair}$ and ran the experiments for this paper. The thesis was co-supervised by Ashwin Prasad Shivarpatna Venkatesh and Markus Schmidt. I was actively involved in designing the approach, defining new evaluation metrics, conducting systematic evaluation, and data curation. I drafted the original paper and led the review and editing.

- **Chapters:** Chapter 7

  Michael Schlichtig, Markus Schmidt, Tom Ohlmer, Carina Brenke, Yashikkumar Khunt, Aniruddh Vyas, Mohit Jain, Dharmik Savaliya, Kilian Demuth, Frank Nelles, Felix Pauck, Christian Reuter, and Eric Bodden. "*SecAI* — Improving the Usability of Static Application Security Testing". Unpublished manuscript. 2026 [Sch+26c] (***Contribution 5***)

Content: In this paper, we have built *SecAI*, a *CogniCrypt*-plugin [Krü+19] for *SonarQube* [Sonb], based on our previous research contributions and the research literature. The goal of *SecAI* is to build a prototype to evaluate to what extent different usability features of SAST tools support developers in reducing cryptographic API misuses.

Contribution: I led this project to develop the plugin, which began with the supervision of a one-year project group of six master's students, co-supervised by Markus Schmidt. I was actively involved in designing the approach, designing *SecAI*, the partial evaluation, and data curation. I was actively involved in writing, reviewing, and editing the paper.

Tom Ohlmer, Michael Schlichtig, and Eric Bodden. "FP-Predictor – False Positive Prediction for Static Analysis Reports". In: *2026 IEEE/ACM 2nd International Workshop on Advancing Static Analysis for Researchers and Industry Practitioners in Software Engineering (STATIC)*. 2026 [OSB26b]
Parts of Section 7.4.8 are taken from this paper.

Content: In this paper, we investigated how well Graph Convolutional Networks (GCNs) can be used to predict false positives in static analysis reports. We used two benchmarks in the domain of cryptographic API misuse detection, *CamBench* [Sch+22c] and *CryptoAPI-Bench* [ARY19], and *CogniCrypt* [Krü+19] as the SAST tool in this paper. We used *CamBench* to train and test our GCN model, $FP_{Predictor}$. To evaluate $FP_{Predictor}$, we analyzed *CryptoAPI-Bench* with *CogniCrypt* and used $FP_{Predictor}$ to predict the analysis reports. Our manual inspection showed that $FP_{Predictor}$ reached an overall accuracy of about 96.6% after manual inspection.

Contribution: I led this subproject as part of the one-year project group. I was actively involved in designing the approach, designing the evaluation, and data curation. I was actively involved in writing, reviewing, and editing the paper.

Another important collaboration was part of deriving the requirements for the final main contribution in Chapter 7:

> Luis Amaral, Michael Schlichtig, Wagner Emanuel, Joilton Almeida, Carine Ferreira, Jerome Kempf, Rodrigo Bonifácio, Eric Bodden, Laerte Peotta, Gustavo Pinto, and Márcio Ribeiro. "From Legacy Designs to Vulnerability Fixes: Understanding SAST Adoption in Non-Technological Companies". In: *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2026 [Ama+26]
>
> **Content:** In this paper, we investigated the adoption of SAST tools in industry as well as related challenges. We used the Socio-Technical Grounded Theory (STGT) [Hod22] method and conducted focus group sessions across three companies to develop a set of hypotheses on the state of SAST tool adoption. In the focus group sessions, we presented developers with selected cryptographic misuses in their company's codebase reported by the SAST tools *CogniCrypt* [Krü+19] and *CryptoGuard* [Rah+19]. For hypothesis formation, we coded the interviews and identified several findings. These include the experienced benefit of SAST tools, their inherent limitations as well as missing features, and a possible overconfidence of developers in tooling.
>
> **Contribution:** This project was led by Luis Amaral. I joined the project to code the focus group sessions and to apply the STGT method for hypothesis formation. I was actively involved in writing, reviewing, and editing the paper.

The chapter *Implementations and Data* provides a comprehensive overview of available implementations and datasets.
One goal of this thesis is to make static analysis results usable for users. Therefore, several publications leading to the PhD thesis of Mugdha Khedkar have been supported as a co-author, but are not part of this thesis. This includes the following publications:

- Mugdha Khedkar, Michael Schlichtig, and Eric Bodden. "Advancing android privacy assessments with automation". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. ACM, 2024, pp. 218–222 [KSB24]
- Mugdha Khedkar, Michael Schlichtig, Nihad Atakishiyev, and Eric Bodden. "Between law and code: challenges and opportunities for automating privacy assessments". In: *Automated Software Engineering* 33.2 (2026), p. 56 [Khe+26a]

- Mugdha Khedkar, Michael Schlichtig, and Eric Bodden. "Source Code-Driven GDPR Documentation: Supporting RoPA with Assessor View." In: *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. IEEE, 2026 [KSB26]
- Mugdha Khedkar, Michael Schlichtig, Santhosh Mohan, and Eric Bodden. *Visualizing Privacy-Relevant Data Flows in Android Applications*. 2025. arXiv: 2503.16640 [cs.CR] [Khe+25]
- Mugdha Khedkar, Michael Schlichtig, Mohamed Soliman, and Eric Bodden. "Challenges in Android Data Disclosure: An Empirical Study". In: *2026 IEEE/ACM 13th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 2026 [Khe+26b]

This dissertation was independently written by the author unless otherwise noted, e.g., by the explanation of the author's contributions in this chapter. Language and editorial support, formatting of tables and figures, as well as code assistance were provided by DeepL[2], Grammarly[3], Claude[4], AI-Chat[5] of Paderborn University with large language models, e.g., OpenAI's ChatGPT[6], Meta's Llama[7], Google's Gemma[8], and Alibaba Cloud's Qwen[9].

[2] https://www.deepl.com/
[3] https://www.grammarly.com/
[4] https://claude.ai/
[5] https://ai-chat.uni-paderborn.de/
[6] https://chatgpt.com/
[7] https://www.llama.com/
[8] https://ai.google.dev/
[9] https://qwen.ai/

Contents

1 Introduction

The increasing prevalence of software systems has led to an exponential growth in the amount of data being collected and processed. Protecting this data relies heavily on the correct application of cryptographic techniques. Usually, the majority of developers of these software systems are not experts in cryptography [Nad+16] and struggle with applying Java cryptographic Application Programming Interfaces (*API*s) securely, e.g., because of the API's complexity or lack of useful documentation. Furthermore, the recent surge in code generation via large language models (*LLM*s) introduced new risks, including a new source of insecure implementations [Aku+23]. Ensuring the secure use of cryptographic APIs in software systems is therefore a critical challenge.

Prior studies have shown that developers frequently encounter difficulties when applying cryptographic algorithms correctly [HGN20; Nad+16; Wic+22]. For example, a study by Hazhirpasand et al. [HGN20] reported that 85% of the analyzed projects contained at least one misuse of cryptographic APIs, while Egele et al. [Ege+13] found that 10,327 out of 11,748 Android applications (about 88%) in the Google Play marketplace exhibited at least one cryptographic misuse. Lazar et al. [Laz+14] found that among 269 vulnerabilities associated with cryptography, 83% resulted from developers' misuse of cryptographic APIs. Furthermore, Gajrani et al. [Gaj+17] noted that 90% of applications sourced from various app stores possess cryptographic vulnerabilities that can be exploited.

To support developers in developing secure software, static analysis tools analyze program code to detect potential errors without executing it. They are used in different contexts and for different purposes, spanning varying levels of complexity. For instance, there are rather simple tools for code quality [Dli; Sta], more complex bug finders [PMD; Spo], and even more sophisticated security scanners that search for exploitable vulnerabilities [Sec; Art20]. Research in static analysis has made significant progress, enabling a broader range of applications.

Static Application Security Testing (*SAST*) tools can assist, e.g., by detecting *cryptographic API misuses* [Sch+22a] within codebases. One common approach to SAST is static analysis, which examines code for errors, vulnerabilities, and potential improvements without actual execution. By analyzing source code, intermediate

representations, or binaries, static analysis tools can detect issues such as null pointer dereferences, buffer overflows, dead code, or violations of coding standards [RY20]. They operate by examining the source code or compiled bytecode of a software application without executing it [CW07]. Such tools analyze code structure, logic, and data flow to identify potential security vulnerabilities, enabling developers to address security flaws at the source code level.

By identifying and reporting errors and vulnerabilities to developers, static analysis is a crucial and recommended technique in software development [NSA22; HL06]. Due to their general usefulness, there is a wide variety of static analysis tools for different contexts, purposes, and levels of analysis complexity. They range from more basic linters for code quality checks [Dli; Sta] to increasingly complex bug finders [PMD; Spo] and more sophisticated static application tests for detecting exploitable vulnerabilities [Sec; Art20; Krü+19]. SAST tools such as *CogniCrypt* [Krü+19], *CryptoGuard* [Rah+19], *FindSecBugs* [Art20], *SonarQube* [Sonb], etc., have already proven extremely useful. Each has its own ruleset, which security experts can easily update. However, due to the inherent undecidability of static code analysis, they can only provide partial solutions in practice [Lan92]. For instance, de Mendonça et al. [Men+13] revealed that SAST tools such as *FindSecBugs* [Art20], *PMD* [PMD], and *CheckStyle* [Che] produce a high rate of false positives.

Such tools have the capacity to provide significant support by automatically detecting such misuses in codebases. Nevertheless, existing tools are often limited by usability shortcomings, manifesting in inappropriate default configurations, unclear warning messages, and other neglected usability criteria [CB16]. Johnson et al. [Joh+13] identified that too many false positive reports and a general developer overload are key issues. Further, they suggest that static analysis tools should present issues in an intuitive manner. As a result, developers abandon these tools or ignore their findings, leaving vulnerabilities unaddressed and undermining the overall trust and adoption of static analysis solutions in practice [CB16]. Overall, research has identified various usability factors and issues with static analysis tools [CB16; Joh+13; SDM20; NSB22].

As shown in a study by Nadi et al. [Nad+16], such tools are needed to support the developers. In the study, developers found cryptographic APIs too complex and difficult to use. Therefore, developers need support, which static analysis tools can provide. However, research has shown that although developers perceive using a static analysis tool as beneficial, many barriers hinder its adoption, including usability issues [Joh+13; CB16]. These barriers include too many false positives, poor presentation of warning messages, difficulty in workflow integration, and

limited customization options [NDB19]. While too many false positives are an important issue caused by the imprecision of static analysis and can be addressed by improving the analysis itself, research has shown that usability aspects need to be addressed [DWA20; SDM20] as well.

For secure software development, supporting developers is integral [Krü20]. Many vulnerabilities stem from misuse of cryptographic APIs [Ege+13; Laz+14; Krü20]. Helping developers reduce cryptographic API misuses thus aids in secure coding. Achieving better support for developers via SAST tools involves several steps, including improving the underlying static analysis and building usable tools that provide actionable insights from the analysis results. Hence, the goal of this thesis is to better support developers in reducing cryptographic API misuses. To achieve the goal of this thesis, several relevant aspects were identified and investigated. These include the state of the art in usability criteria addressed by static analysis tools [NSB22] to identify important usability features, the origin of API misuses [Sch+22a] to explain identified issues better, a benchmark for cryptographic API misuse detection tools [Sch+22c], new tooling for automated cryptographic API misuse repair [Sch+26b], and a usability-focused user interface for the SAST tool *CogniCrypt* [Sch+26c; OSB26b].

1.1 Thesis Contributions and Structure

An overview of the contributions in this thesis is given in Figure 1.1, which shows the relevant publications in boxes with their titles and literature references. The contributions are referenced with circled numbers indicating the chapter or section in which they are mainly discussed; e.g., ③ denotes Chapter 3. Main contributions of this thesis are visualized in orange boxes, while white boxes show co-authored publications that are foundational to the final main contribution, *SecAI*.

The thesis contributions are split into two elements:

- Research results shown in the top large box, which are presented in Chapter 3, Chapter 4, and Chapter 5, lay the foundations for developing new prototypes and tools [Sch+26b; OSB26b; Sch+26c].
- New tools and prototypes are presented in the bottom large box; they are discussed in Chapter 6, Section 7.4.8, and Chapter 7. Here, Chapter 7 introduces *SecAI*, a *CogniCrypt*-plugin for *SonarQube* [Sonb], with the goal of improving

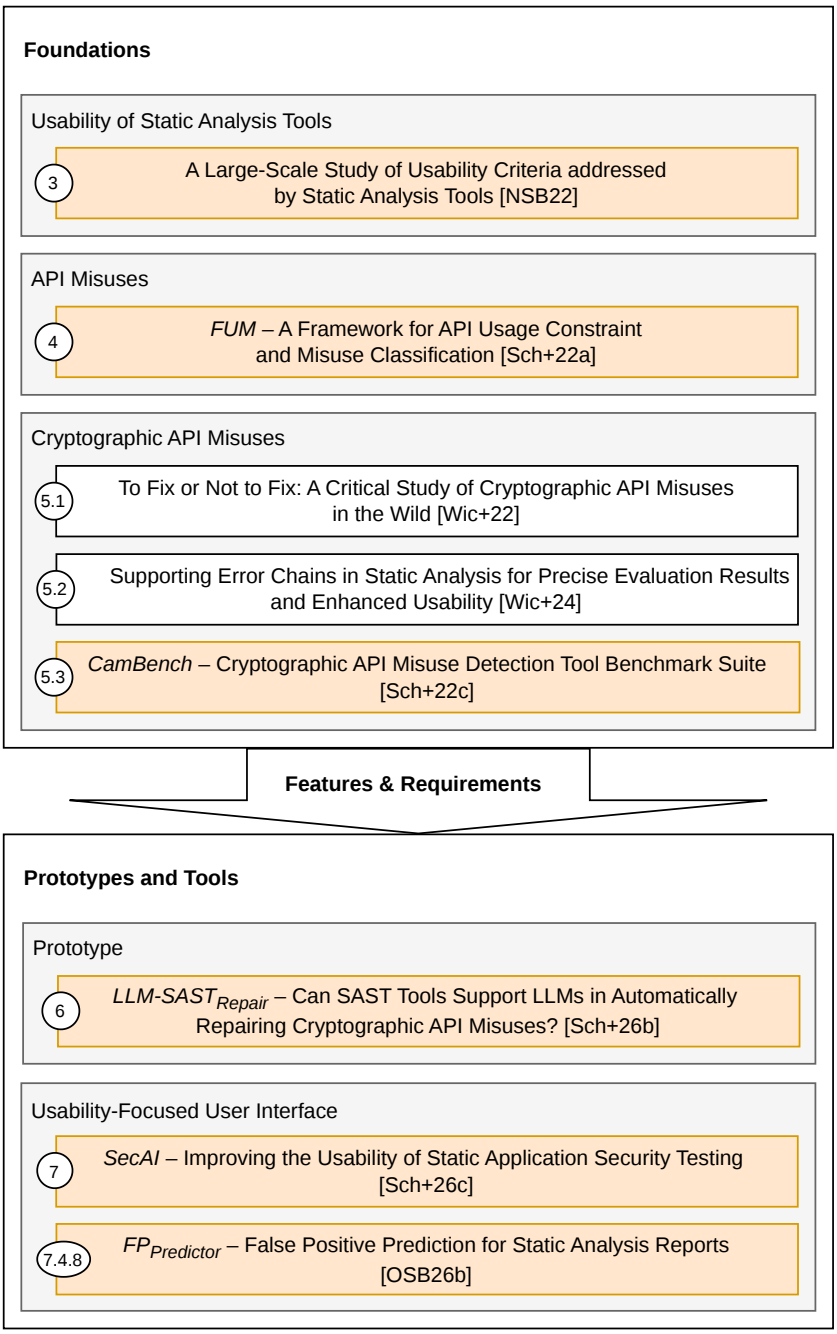


Figure 1.1: Overview of contributions of this thesis. Circled numbers denote chapters in which the respective contributions are mainly presented, e.g., ③ denotes *Contribution 1* in Chapter 3. The main contributions of this thesis are shown in orange boxes, and the white boxes contain co-authored publications that serve as the foundations of these contributions, especially *SecAI* [Sch+26c].

the usability of *CogniCrypt* [Krü+19] and thereby better supporting developers in reducing cryptographic API misuses.

The name *SecAI* combines the words ***Sec***urity and ***AI***, leveraging the domain knowledge of an existing static analysis tool for ***Sec***urity, while maintaining the analysis tool's warning reports and enhancing them with ***AI***-based features, including false positive detection [OSB26b] and automatic repair [Sch+26b].

SecAI has been developed based on all the other contributions of this thesis. Contributions from Chapter 3 [NSB22] (*Contribution 1*) were used to derive the requirements and features for *SecAI* (cf. Section 7.4). To better understand the causes of Java API misuses, Java API Usage Constraints were studied [Sch+22a] in Chapter 4 (*Contribution 2*), laying foundations for other contributions of this thesis. For feature development, co-authored contributions from Section 5.1 [Wic+22] with a risk model (cf. Table 5.1) for the *Severity Score* feature (cf. Section 7.4.7) and Section 5.2, namely the adapted analysis of *CogniCrypt* [Krü+19] for the feature to visualize error chains (cf. Section 7.4.3), were used. Further, the benchmark *CamBench* [Sch+22c] (*Contribution 3*, cf. Section 5.3) was contributed and used to build and evaluate the AI features for false positive prediction, $FP_{Predictor}$ [OSB26b] (cf. Section 7.4.8), and cryptographic API misuse repair, $LLM\text{-}SAST_{Repair}$ [Sch+26b] (*Contribution 4*, cf. Chapter 6). Overall, all contributions culminate in the development of the new, usability-focused *CogniCrypt*-plugin for *SonarQube*, *SecAI* (*Contribution 5*, cf. Chapter 7).

CogniCrypt [Krü+19] is the static analysis tool for cryptographic API misuse detection that our thesis primarily revolves around, but is not limited to, because we took over the maintenance and further development of *CogniCrypt* as part of the research effort performed for this thesis. *CogniCrypt* was one of the main contributions of Stefan Krüger's PhD thesis [Krü20], which set a very similar goal to ours. Krüger's work [Krü20] preceded our thesis and laid a very solid foundation. Aiming to better support developers in reducing cryptographic API misuse, we found similarities in our respective contributions; however, thanks to Krüger's efforts and new advances, e.g., in machine learning with LLMs [Aku+23], we were able to make improved contributions. Since Krüger et al. already contributed *CogniCrypt* and its domain-specific language *CrySL* [Krü+19], in our thesis we could shift our focus to adapting it, e.g., to reason about error chains [Wic+24], as Krüger already described them as "[...] unsatisfied predicates accumulate transitively" [Krü20].

To distinguish features and contributions around *CogniCrypt*, Krüger introduced specific terms: (i) $CogniCrypt_{\text{SAST}}$ [Krü+19] is the static program analyzer for cryptographic API misuses and uses *CrySL* [Krü+19] for its configuration, (ii) $CogniCrypt_{\text{GEN}}$ [KAB20] generates secure code employing a template-based approach, and (iii) $CogniCrypt_{\text{FIX}}$, a concept on how to repair detected cryptographic API

misuses automatically. Within this thesis, when we use the term *CogniCrypt*, we usually refer to *CogniCrypt*$_{\text{SAST}}$.

In our thesis, we contributed *LLM-SAST*$_{Repair}$ [Sch+26b] (*Contribution 4*, cf. Chapter 6) which implements the concept of *CogniCrypt*$_{\text{FIX}}$. Using the *repair loop* of *LLM-SAST*$_{Repair}$, we also contributed a secure code generation feature (cf. Section 7.4.6) for any user prompt in our new usability-focused *CogniCrypt*-plugin for *SonarQube*, *SecAI* (*Contribution 5*, cf. Chapter 7) — in comparison, *CogniCrypt*$_{\text{GEN}}$ is limited to its predefined templates. Krüger's final contribution was a prototypical Eclipse-plugin [Coga], combining *CogniCrypt*$_{\text{SAST}}$ and *CogniCrypt*$_{\text{GEN}}$, to improve the usability of *CogniCrypt*, which was evaluated in a user study with 24 participants [Krü20; Krü+23] and well received by the participants. In contrast, we contributed *SecAI* (*Contribution 5*, cf. Chapter 7) following a systematic approach based on our first main contribution, a large-scale empirical study on usability criteria addressed by static analysis tools [NSB22] (*Contribution 1*, cf. Chapter 3). Combined with a study on SAST tool adoption in industry [Ama+26], we derived eight usability features (cf. Section 7.4) to fulfill most (32 of 36) of our identified usability criteria.

Overall, the contributions made in Krüger's PhD thesis [Krü20] laid the foundations that we have built upon. Beyond the explained similarities in contributions, we contributed *FUM* [Sch+22a] (*Contribution 2*, cf. Chapter 4), a framework for API usage constraint and misuse classification, and *CamBench* [Sch+22c] (*Contribution 3*, cf. Section 5.3), a benchmark for evaluating cryptographic API misuse detection tools.

CogniCrypt is considered state of the art in API misuse detection and has been shown to be able to cover a majority of cryptographic API misuses [Gao+19; HGN20; Bon+21]. While the research in this thesis relates to *CogniCrypt*, we aimed for generalizability in our contributions. In *Contribution 1*, we conducted a large-scale study of recommended static analysis tools in the literature, resulting in a comprehensive overview of the state of the art in the usability criteria addressed by static analysis tools [NSB22]. In *Contribution 2*, we investigated the origins of Java API misuses and proposed *FUM*, a framework for their classification [Sch+22a]. In *Contribution 3*, we created *CamBench*, a benchmark for evaluating and comparing static analysis tools for cryptographic API misuse detection [Sch+22c]. In *Contribution 4*, we built *LLM-SAST*$_{Repair}$ to automatically repair cryptographic API misuse [Sch+26b]. *LLM-SAST*$_{Repair}$ is designed to be generalizable; e.g., the evaluation experiments were performed using two static analysis tools, *CogniCrypt* [Krü+19] and *CryptoGuard* [Rah+19]. Finally, in *Contribution 5*, we built *SecAI*, our *CogniCrypt*-plugin

using *SonarQube* as a platform [Sch+26c]. Overall, if applicable in our implementations, we strived for adaptability and generalizability of results. Implementations and artifacts of this thesis are provided in Chapter 9.

***Contribution 1*: Usability of static analysis tools.** In Chapter 3, we present the first contribution of this thesis. To better support developers in remediation of SAST tool reports, it is important to increase their usability. Research [Joh+13; CB16; SDM20] has shown that developers often experience usability issues when using static analysis tools, which may lead them to avoid using such tools. Based on the work of Nachtigall et al. [NDB19], we performed a *large-scale study of usability criteria addressed by static analysis* [NSB22]. We compiled a list of recommended static analysis tools and usability criteria from the literature. In our empirical study, we rated 46 static analysis tools against 36 usability criteria, grouped into six categories. This study showed that static analysis tools often fail to meet many usability criteria. Moreover, about a quarter of the tools could not even be properly executed, further indicating a lack of user experience when using recommended open-source static analysis tools. Finally, in the context of this thesis, this contribution provides valuable insights into important usability criteria for static analysis tools and the features that fulfill them. Therefore, this contribution can provide the foundation for a systematic approach to building a usable user interface for a static analysis tool, for which these usability categories and criteria serve as the systematic design basis for developing *SecAI* (*Contribution 5*) as the final main contribution in Chapter 7.

***Contribution 2*: *FUM* – Framework for API Usage constraints and Misuse classification.** In Chapter 4, we introduce the second contribution of this thesis. To increase the explainability of warning reports from static analysis tools for developers, the cause of the reported issue must be well understood. In this thesis, we focus on API misuses. Recent studies show that developers often face several difficulties [SHA15; Nad+16; Laz+14; Isl20; Gu+19] when using APIs, including inadequate documentation, complex APIs, and inadequate abstraction [Nad+16]. As a result, developers tend to incorrectly integrate APIs into their code, leading to misbehavior, errors, or even program crashes. Research [Ama18; Ama+16; Li20; Luo+18] has shown that misuse of APIs extends beyond specific, complex domains like cryptographic APIs; it occurs across all API usage. In this contribution, we conducted a structured literature review to further the understanding of API misuse types and their causes. Based on previous work [Ama18; Ama+16; Li20; Mon+12], we formulate concrete definitions of API usage constraints and their violations as causation of API misuses. These definitions and prior work form the foundation for deriving *FUM*, our

***F**ramework for API **U**sage constraints and **M**isuse classification* [Sch+22a]. In further contributions, this allows us to classify and better explain API misuses.

***Contribution 3*: *CamBench* – Benchmark for cryptographic API misuse detectors.** To support developers and security auditors in identifying cryptographic API misuses, many detectors, such as *CryptoRex* [Zha+19], *CryptoLint* [Ege+13], *CogniCrypt* [Krü+19], and *CryptoGuard* [Rah+19], have been developed. However, there is a lack of a standard benchmark for this domain. None of the six identified relevant benchmarks [ARY19; Bra+17; Wic+19; Ama+16; MR17; Wic] fully cover the domain of cryptographic API misuse detectors, and only two of the 18 identified detectors have been evaluated on more than one benchmark.

Therefore, in Section 5.3, we propose the creation of *CamBench*, a "***C**ryptographic **A**PI **M**isuse Detection Tool **Bench**mark Suite*" [Sch+22c] based on existing benchmark generation approaches [Bla+06; Tem+10; DEB16], to evaluate cryptographic API misuse detectors. *CamBench* comprises three elements: a real-world benchmark, a synthetic micro-benchmark for static analysis capabilities ($CamBench_{CAP}$), such as field- and object-sensitivities, and a heuristic to measure cryptographic API coverage. We introduce the implementation of $CamBench_{CAP}$ (*Contribution 3*) in Section 5.3. $CamBench_{CAP}$ uses *FUM* [Sch+22a] (*Contribution 2*) to specify the type of API usage constraint and misuse in the ground truth files per case, and it is used to train an AI model for false positive prediction [OSB26b] and to evaluate the automatic repair of cryptographic API misuses [Sch+26b] (*Contribution 4*).

***Contribution 4*: $LLM\text{-}SAST_{Repair}$ – Automatic repair of cryptographic API misuses.** Recent results [SFM25; FG25] showed this to be a promising combination; thus, we applied it to the domain of cryptographic API misuses. In Chapter 6, we present $LLM\text{-}SAST_{Repair}$ [Sch+26b], our approach to combine LLMs and SAST tools to perform Automated Program Repair (*APR*) [LPR19] of cryptographic API misuses. We designed $LLM\text{-}SAST_{Repair}$ to be extendable and generalizable, supporting LLMs via API keys and multiple SAST tools when integrated via the Static Analysis Results Interchange Format (*SARIF*) [OAS20]. We systematically evaluated multiple combinations and configurations of LLMs and SAST tools on two benchmarks, *CamBench* [Sch+22c] (*Contribution 3*) and *CryptoAPI-Bench* [ARY19], showing that configurations with SAST tool guidance perform better at repairing cryptographic API misuses. In our experiments, we used a selection of LLMs and two SAST tools, namely *CogniCrypt* [Krü+19] and *CryptoGuard* [Rah+19]. Overall, *CogniCrypt* combinations performed best and were able to increase the overall security of a

benchmark by up to about 70%. In the context of this thesis, *LLM-SAST$_{Repair}$* provides developers with support for fixing cryptographic API misuses that were identified by SAST tools. The repair functionality is further used as a feature in the final contribution of this thesis, *SecAI* (*Contribution 5*).

Contribution 5*: *SecAI* – Usability-focused user interface for *CogniCrypt*.** In Chapter 7, we introduce the final contribution of this thesis in the form of a *CogniCrypt*-plugin for *SonarQube*, named *SecAI* [Sch+26c]. *SecAI* combines the domain knowledge of *CogniCrypt* in ***Security with ***AI***-based features to support developers in remedying reported cryptographic API misuses. *SecAI* is a prototype of a usability-focused user interface for the SAST tool *CogniCrypt* [Krü+19] and was built combining all other contributions and named publications of this thesis. The goal of *SecAI* fully aligns with the thesis goal of *helping Java developers reduce cryptographic API misuses*. We chose to develop a *SonarQube*-plugin [Sona] for dissemination [Ama+26] because *SonarQube* [Sonb] is widely used in industry, and because of the importance of fluent workflow integration [NSB22]. *SecAI* provides eight usability features to fulfill 32 of the 36 usability criteria identified by Nachtigall et al. [NSB22] (*Contribution 1*) and to address the industry need for SAST tool adoption [Ama+26].

For instance, *LLM-SAST$_{Repair}$* [Sch+26b] (*Contribution 4*) was integrated as one of the fix features for cryptographic API misuses – the repair functionality of *LLM-SAST$_{Repair}$* was combined with user interface elements such as a button for requesting a fix, user feedback on whether *CogniCrypt* verified the patch as successful (or the corresponding warning messages), and a button for opening a pull request to incorporate the patch into the codebase. Using a risk model [Wic+22] for a *Severity Score* and *FP$_{Predictor}$* [OSB26b] to predict whether a warning is a false positive as a *Confidence Score*, we further calculate a *Priority Score* for each *CogniCrypt* warning. These three scores are intended to help developers decide which warnings to address first.
Increased explainability of error messages and their descriptions was achieved based on insights from, e.g., *FUM* [Sch+22a] (*Contribution 2*) and *CamBench* [Sch+22c] (*Contribution 3*). Further, the adaptation of *CogniCrypt's* analysis algorithm to reason about error chains [Wic+24] was used to build a visualization of error trees.

Overall, *SecAI* is the overarching final tool contribution of this thesis to help Java developers reduce cryptographic API misuses. It was developed based on the foundational contribution efforts of this thesis and combines multiple tooling and functionality contributions into a single tool and concept. *SecAI* is a prototype of a usable user interface for a SAST tool that follows a concept to guide the developer

through warnings with multiple scores, provide aid in remediation via precise error localization, increase the explainability of error descriptions, provide visualization of error chains, and, finally, provide features for automated error fixing.

1.2 Overview

In this thesis, we aim to help developers reduce cryptographic API misuse by developing a usability-focused user interface for the existing static analysis tool *CogniCrypt* as a plugin for *SonarQube*, named *SecAI* (*Contribution 5*). The first three contributions lay the foundation for this plugin by examining the state of the art in usability criteria fulfilled by static analysis tools (*Contribution 1*), the origins of API misuses (*Contribution 2*), and, specifically, cryptographic API misuses and the creation of a benchmark for them (*Contribution 3*). Based on these contributions, new usability features are developed and integrated, e.g., automated repair of cryptographic API misuses, *LLM-SAST*$_{Repair}$ (*Contribution 4*), into a single usability-focused user interface, *SecAI* (*Contribution 5*), for *CogniCrypt*.

The remainder of this thesis is structured based on the order of contributions shown in Figure 1.1 and discussed in Section 1.1. The relevant state of the art at the beginning of this thesis and background information for this thesis are provided in Chapter 2. Chapter 3 presents the first contribution of this thesis: a large-scale study of the usability criteria addressed by static analysis tools (*Contribution 1*). Chapter 4 introduces our definitions around the term *API misuse* and our classification framework *FUM* (*Contribution 2*). Chapter 5 provides more specific investigations of cryptographic API misuses (cf. Sections 5.1 and 5.2) and a benchmark, *CamBench* (*Contribution 3*), for their detection tools (cf. Section 5.3). Chapter 6 discusses *LLM-SAST*$_{Repair}$, which combines SAST tools and LLMs to automatically repair cryptographic API misuses (*Contribution 4*). Chapter 7 presents the final contribution of this thesis, *SecAI* (*Contribution 5*), a usability-focused *CogniCrypt*-plugin for *SonarQube*. One feature of *SecAI* is described in more detail in Section 7.4.8. Chapter 8 concludes the thesis, and the availability of implementations and artifacts is presented in Chapter 9.

2 Background

In this chapter, we first present the relevant state of the art at the beginning of this thesis in Section 2.1. This is split into three topics this thesis addresses: the usability of static analysis tools in Section 2.1.1, research on API usage constraints and misuses in Section 2.1.2, and benchmarking of static analysis tools in Section 2.1.3. Then we present the fundamentals for the detection of Java cryptographic API misuses with the static analysis tool *CogniCrypt* [Krü+19], which we primarily used in this thesis, in Section 2.2. This section provides an overview of cryptographic API misuses in Section 2.2.1, explains how *CogniCrypt* and its domain-specific language *CrySL* work in Section 2.2.2, and briefly explains the term *effective false positive* [Sad+15] in Section 2.2.3.

2.1 State of the Art

Research has identified many aspects of how to improve the usability of static analysis tools (cf. Section 2.1.1), as well as the accuracy of static analysis tools, which can be evaluated using benchmarks (cf. Section 2.1.3). We will separately discuss the origins and fundamentals of API usage constraints and misuses (cf. Section 2.1.2) in more detail, as they are key to a good presentation and explainability of warning messages, which are part of the usability of static analysis tools. As described in Chapter 1, we focused on the domain of cryptography and Java.

2.1.1 Usability of Static Analysis Tools

The usability of static analysis tools has been investigated in several studies. Johnson et al. [Joh+13] conducted a study that interviewed 20 participants to identify reasons developers might not use static analysis tools. They found that, in addition to issues with the static analysis' accuracy (e.g., too many false positives), there are many usability barriers for developers (e.g., an understandable representation of warnings). Bessey et al. [Bes+10] reported on their experiences as a static-tools company and discussed what developers need, e.g., low rates of false positives and

well-explained warnings; if warnings are misunderstood, developers may ignore them or even perceive them as false positives. Sadowski et al. [Sad+15] discuss and define the interesting phenomenon that developers do not fix correct findings (true positives) reported by a static analysis tool because they perceive them as false positives, e.g., because of an insufficiently explained presentation of the finding, as *effective false positives* (cf. Section 2.2.3). Christakis et al. [CB16] surveyed developers and received 375 responses to investigate what developers would like static analysis tools to provide. The issues reported by developers included the accuracy and runtime of static analysis tools, but most can be categorized as usability issues.

A useful categorization of usability challenges, summarizing the literature, was provided by Nachtigall et al. [NDB19], focusing on the explainability of static analysis results and also addressing challenges such as *Workflow Integration*, *User Feedback*, and *Fix Support*. They used their categories to provide a further overview of existing static analysis tools while also classifying how the tools present their results to users, namely, *Standalone*, *IDE*, *Multi-Interface*, and *CLI*. Another study conducted by Nguyen Quang Do et al. [DWA20] with 87 participants at Software AG further revealed workflow integration as an important usability challenge. Smith et al. [SDM20] examined four static analysis tools in terms of usability by performing a heuristic walkthrough and employing a set of usability heuristics for exploration. They grouped their findings of usability issues into six themes. Overall, several studies have investigated the usability of static analysis tools, mostly interviewing or surveying developers, analyzing the literature, or performing a usability evaluation on a small set of static analysis tools.

2.1.2 API Usage Constraints and Misuses

As discussed for the usability of static analysis tools (cf. Section 2.1.1), a well-explained warning message plays an important role in helping developers use them — misunderstood warnings often lead developers to reject them [Bes+10]. When developers decide to ignore reported warnings that are actual issues (true positives), this turns them into *effective false positives* [Sad+15] (cf. Section 2.2.3). A basic principle of efficient software development and modern programming languages is the reuse of software through APIs [Hei+11]. In cryptography, Nadi et al. [Nad+16] showed that developers need help, as cryptographic APIs are too complex, and that API misuse leads to security vulnerabilities. To better explain warnings, we must understand the reasons for a warning, which is why we discuss the origins and fundamentals of *API Usage Constraints and Misuses* as they are reported. When using an API, there exist (implicit) restrictions and explanations, e.g., documentation,

on how to use it correctly, e.g., not causing misbehavior or errors. Surveying the literature on API misuse reveals several terms and definitions.

Dekel and Herbsleb [DH09] investigated such restrictions as *usage directives* and how to increase developers' awareness. Monperrus et al. [Mon+12] further derived a classification framework consisting of 26 different *API directive* types. In contrast, Maalej and Robillard [MR13] empirically developed a detailed taxonomy of *API knowledge types*. Li et al. [Li+18] referred to directives where non-compliance may result in errors or unexpected behavior as *API caveats* and introduced a framework including ten API caveat types. Nguyen et al. [NVN19] introduced the *Statistical Approach for API Misuses* (SAM), which attempts to identify non-compliance of constraint types. Regarding the issue of such undocumented *API usage constraints*, Amann [Ama18] defined them as optional, implicit constraints without compiler enforcement. Non-compliance with such *API usage constraints* would result in errors. They also introduced an overview of categories of *API usage constraints* based on Monperrus et al.'s [Mon+12] framework of *API directive* types.

Other empirical studies [SHA15; Cer+18; BBA09; BKA11] examined certain constraints for required orders in which methods of an object need to be called for a program to run correctly — these are often called *object protocols*. The violation of the briefly listed terms capturing restrictions on how to use an API is referred to as *API misuse*. API misuses have been studied in various domains and from several perspectives such as classification, automatic detection, and mitigation. For instance, for cryptography, Egele et al. [Ege+13] studied Android apps empirically and found 88% of the investigated apps contain at least one misuse of cryptographic APIs.

2.1.3 Benchmarking of Static Analysis Tools

Benchmarks are an accepted way to evaluate and compare tools, algorithms, and more. A well-designed and fair benchmark can drive the advancement of tools and create a community that significantly improves the standard of technology. Furthermore, established benchmarks foster research and development in their domain [Bla+06]. The advantages of a properly designed benchmark, e.g., meaningful evaluation of a single tool and fair comparison of multiple tools, are significant; however, creating such a benchmark is very time-consuming and difficult.

For instance, this can be seen in the domain of cryptographic API misuse detection using static analysis. While a benchmark to set common goals, standards, and understanding in the community would be very beneficial for evaluating and

understanding the strengths and weaknesses of static analysis tools, only a few benchmarks [ARY19; Bra+17; Wic+19; MR17] exist and have been used only to evaluate and compare a few tools. Further, existing benchmarks have different focuses, and therefore a standard benchmark for this domain is still missing.

The *Ghera Benchmark* [MR17] targets Android applications and some of their vulnerabilities. The goal of the *Braga Benchmark* [Bra+17] is to help developers choose the static analysis tool best suited to their team. The *Parametric Crypto Misuse Benchmark* [Wic+19] is narrowed to passed function parameters that cause API misuses. *CryptoAPI-Bench* [ARY19] includes several analysis sensitivities for vulnerabilities that can be detected by *CryptoGuard* [Rah+19] and was used for its evaluation, too, which may have introduced a tool-author bias. In 2023, Torres et al. [Tor+23] corrected the ground truth of *CryptoAPI-Bench*.

2.2 Detection of Java Cryptographic API Misuses

In this section, we provide background on cryptographic API misuse in Java and introduce *CogniCrypt* [Krü+19], the static analysis tool for cryptographic API misuse detection that we primarily used in this thesis. In addition, we introduce the term *effective false positives*, as it denotes one of the usability issues this thesis aims to address.

2.2.1 Misuses of Java Cryptographic APIs

In Java, the *Java Cryptography Architecture* (*JCA*) provider architecture [Orah] provides cryptographic functionality in an implementation-independent way. The JCA provides a set of extensible cryptographic components ranging from encryption to authentication to access control, enabling developers to secure their applications. It is implementation-independent through a `provider` architecture; developers can plug in their implementations of cryptographic primitives for use with this architecture. Besides the default provider, JCA, which is shipped with the Java Development Kit (JDK), a commonly used provider is the Bouncy Castle library.

Listing 2.1 illustrates the use of the JCA to sign a byte-array `dataToSign`. For this, the `Signature` object is initialized with a signature algorithm (Line 3), and a private key, passed via the function `getPrivateKey`, is added to the `Signature` object `s` (Line 4). Next, the byte-array `dataToSign` is passed to `s` to actually compute the signature of

the data (Line 5) before the function returns the signed bytes. Unfortunately, the call to `sign` that would return the signature is missing (Line 6).

Listing 2.1: Code snippet that signs a byte array using a key.

```
1   private byte[] signByte(byte[] dataToSign){
2     byte[] signedBytes;
3     Signature s = Signature.getInstance("SHA1WithRSA");
4     s.initSign(getPrivateKey());
5     s.update(dataToSign);
6     // Call to signedBytes = s.sign() missing.
7     return signedBytes;
8   }
9
10  // Get a PrivateKey object with an insecure key length.
11  private PrivateKey getPrivateKey() {
12    return shortKey();
13  }
14 }
```

A *cryptographic API misuse* is a usage of a cryptographic API that violates the secure usage pattern and is therefore considered insecure by experts. A cryptographic API misuse may be syntactically correct, a working API usage, and may not even raise an exception. We will briefly discuss the error types defined by Krüger et al. [Krü+19] to illustrate some misuses.

1. **Constraint Errors** (cf. Listing 2.1, Line 3): cryptographic APIs use parameters to let developers select cryptographic algorithms when initializing cryptographic API objects. These parameters are often passed as strings in a specific format.

2. **Incomplete-Operation Errors** (cf. Listing 2.1, Line 6): The security of cryptographic API objects may rely on a specific protocol. For instance, `Signature` cryptographic API objects, once initialized and filled with data via a call to `update(byte[])`, require a call to the `sign` method to complete the signing operation. Thus, the required calls to complete the use of an initialized cryptographic API object are missing.

3. **Required-Predicate Errors** (cf. Listing 2.1, Line 4): cryptographic API objects often depend on each other. For example, `Signature` objects require a correctly generated `Key` object. In order for a composed cryptographic solution to be secure, it is required that the components on which it depends are secure. Thus, composing a cryptographic API object with required but insecure objects results in a misuse.

4. **Never-Type-of Error:** Sensitive information, e.g., a secret key, should never be of type `java.lang.String`, as strings are considered insecure compared

to mutable byte arrays. Strings are immutable and stay in memory until collected by Java's garbage collector. Thus, they are longer visible in memory for attackers than necessary and outside of the direct control of the developer[1].

5. **Forbidden-Method Errors:** Certain methods of cryptographic API objects should never be called for security reasons. For example, the `PBEKeySpec` object generates keys from passwords and requires a cryptographic salt while initializing the object. Therefore, constructors that are not parameterized with a salt cause a misuse.

6. **Typestate Error:** Such an error occurs when an object moves into an insecure state as the result of an improper method call sequence. For example, a `Signature` object requires a call to the `initSign` method prior to any number of calls to the `update` method.

   The key difference to an *Incomplete-Operation Error* is that the missing and expected call is within the call sequence, while for an *Incomplete-Operation Error* the expected call sequence is not finished.

2.2.2 *CogniCrypt* and *CrySL*

CrySL

CogniCrypt uses an allowlisting approach, where rules describe the secure usage of cryptographic APIs in the domain-specific language *CrySL* [Krü+19]. Currently, *CrySL* supports classes of the Java Cryptography Architecture (JCA), Java Secure Socket Extension (JSSE), Bouncy Castle (BC), and Tink APIs [Cogb]. Violating a rule causes one of two main error types: *typestate errors* and *constraint errors*. A rule can express the secure order of method calls, among other things. If there exists one typestate path for a *specified object*, an instance of a class for which a *CrySL* rule exists, that violates the specified order, *CogniCrypt* reports a *typestate error*. A *typestate error* can be an `IncompleteOperationError` if the secure order is not completed or a `TypestateError` for any other deviation from the secure call order.

CrySL rules comprise several sections, such as `CONSTRAINTS` for specifying constraints on parameters for different methods, `EVENTS` for calls and their parameters, and `ORDER` to specify the secure order of calls (`EVENTS`) in a regular expression. For

[1] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/javax/crypto/spec/PBEKeySpec.html` (last accessed: 2026-06-06)

instance, a *constraint error* describes a violation of a requirement expressed by a *CrySL* rule in the `CONSTRAINT` section, such as a secure encryption algorithm. Overall, five different kinds of *constraint errors* exist. A `ConstraintError` is reported if a constraint on any of the objects specified in the *CrySL* rule is violated. Some *specified objects* require a predicate that signifies a guarantee from another API, i.e., a predicate describes interactions between classes. The *specified object* that can ensure a predicate is an *ensuring object*. If a *specified object* requires a predicate that is not ensured by an *ensuring object*, this results in a `RequiredPredicateError`. If sensitive hard-coded information, insecure types, or forbidden methods were used, *CogniCrypt* reports a `HardCodedError`, `NeverTypeOfError`, or `ForbiddenMethodError`, respectively.

CogniCrypt

In this thesis, we primarily used the static analysis tool *CogniCrypt* for cryptographic misuse detection, which follows an allowlisting approach. In contrast to denylisting approaches such as *CryptoGuard* [Rah+19], which describe vulnerabilities, allowlisting approaches describe all secure API usages and report their violation. Violations of the defined rules are reported as misuses [Krü+19]. Previous studies reported a precision of 85% to 94% [Krü+19; HGN20]. Further, *CogniCrypt* supports the BC library next to the JCA.

All *CrySL* rules for one JCA provider, e.g., the JCA default implementation, form a ruleset that is consumed by *CogniCrypt* to configure its analysis. Internally, each rule is parsed into a finite-state machine (FSM) that encodes all specified elements of the rule, e.g., constraints or the secure call order for a given class. Using the analysis framework *Soot* [Val+10] and the data-flow framework IDE^{al} [SAB17], *CogniCrypt* performs a flow-, field-, and context-sensitive typestate analysis.

2.2.3 Effective False Positives

Past research has shown that developers consider false positives as the "Achilles heel" of static analyses [Joh+13; Aye+07]. Bessey et al. [Bes+10] reported that developers need low rates of false positives and well-explained warnings. Misunderstood warnings, whether true or false, can lead developers to perceive them as false positives. In practice, the definition of false positives can vary depending on the perspective, e.g., the designer of the static analysis tool or the developer using it: From a static analysis perspective, a false positive is a finding which is incorrectly

identified by the analysis. For developers, on the other hand, some findings cannot be fixed in the application, e.g., due to a broken standard. Sadowski et al. [Sad+15] introduce the term *effective false positives* to refer to reported misuses for which a user will not take further action.

Hence, in this thesis, we define the term *effective false positive* as a true positive reported by the static analysis that a developer does not remediate for any reason, e.g., not understanding the report, as discussed by Bessey et al. [Bes+10], or deeming the reported issue as not relevant. As an example, consider a use of *MD5* that is correctly flagged by the static analysis. However, the *MD5* usage at hand is in a non-security context, as external factors cannot influence the concrete call, and it is not essential to the software's security.

3 Large-Scale Tool Study of Usability Criteria Addressed by Static Analysis Tools

This chapter presents the first main contribution of this thesis, a large-scale study of usability criteria addressed by static analysis tools (*Contribution 1*). The results of this empirical study provide valuable insights into the state of the art of usability and usability features of static analysis tools. In the context of this thesis, these are pivotal foundations to build a usability-focused static analysis tool that helps developers reduce cryptographic API misuses.

To comprehensively assess the current state of the art, this chapter presents the first systematic usability evaluation of a wide range of static analysis tools. We derived a set of 36 relevant criteria from the scientific literature and gathered a collection of 46 static analysis tools complying with our inclusion and exclusion criteria — a representative set of mainly non-proprietary tools. Then, we evaluated how well these tools fulfill the aforementioned criteria.

The evaluation shows that more than half of the considered tools offer poor warning messages, while about three-quarters of the tools provide hardly any fix support. Furthermore, the integration of user knowledge is strongly neglected, which could be used for improved handling of false positives and tuning the results for the corresponding developer. Finally, issues regarding workflow integration and specialized user interfaces are further confirmed.

These findings should prove useful in guiding and focusing further research and development in the area of user experience for static code analyses.

3.1 Introduction

Static analysis tools play an important role in supporting developers in detecting and resolving code issues. However, as described in Section 2.1.1, research has identified many usability issues of static analysis tools that may hinder adoption by developers.

While the central task of static analysis is detecting respective code problems, the tool also has to inform the developer about these issues, support them in resolving issues, and offer all of its features in a usable way.

If the tool is not able to support the developer in these aspects, they will not be able to solve or even understand the issue — they might not even believe the tool and regard correct warnings as false positives [Bes+10], so-called *effective false positives* [Sad+15] (cf. Section 2.2.3). If it is too difficult to access the tool's features in a user-friendly way, the developer will simply not use it. Hence, previous research has shown that it is insufficient to only study the technical perspective of static analyses and that research should rather also focus on the users' perspective and support users accordingly [Joh+13; CB16; DWA20; SDM20]. Over the past decade, this perspective has been supported by several publications, pointing to specific flaws and weaknesses when it comes to the user experience with static analysis tools [Har+10; Tho+16; Vas+18; Sad+18; Imt+19].

The study in this chapter is the first to provide a *comprehensive* view of the current state of *Static Analysis Tools* from the user's perspective. While prior studies drew anecdotal evidence from surveys, interviews, or the authors' own experiences, this chapter takes a complementary view of reported issues by instead considering a broad range of existing tools and evaluating them directly. This chapter thus presents the first such direct assessment of user interactions offered by a large fraction of the static analysis tool landscape. Importantly, it paints a *current* picture of this landscape — after all, some of the previous studies are several years old, and many tools might well have evolved since then.

To evaluate the current state of static analysis tools from this perspective, we executed an assessment of current tools. We searched the current related literature in detail and extracted 36 relevant criteria, which we used for the evaluation. We discussed and defined these criteria with their assessment grades in advance. Furthermore, we collected 243 tools and evaluated a representative set of 46 tools that met our inclusion and exclusion criteria. Our evaluation yields central open challenges in the current state of static analysis tools: Overall, the tools present their warnings rather poorly, which causes a high risk that developers may not understand or even care about the warning. Furthermore, in most cases neither the warning messages nor any other feature in the tool supports the developer in fixing the issue. Next, current tools neglect the potential of asking for the developer's knowledge and integrating this information to provide a more tailored user experience. Last but not least, the tool integration into developers' workflow remains an open challenge in many tools.

To summarize, this chapter makes these original contributions:

- It presents a catalog of 36 usability-evaluation criteria drawn from the scientific literature.
- It evaluates the current state of static analysis tools by applying these criteria to 46 current static analysis tools.
- For each criterion, the chapter discusses why it is relevant, and what fraction of current tools offer features to fulfill the respective criterion.
- It reports open challenges and neglected aspects, which might be starting points for further research.

The remainder of the chapter is structured as follows: In Section 3.2, we describe our general methodology on how we collected tools, defined criteria, and executed the evaluation. Section 3.3 discusses our criteria in more detail and presents our evaluation results. Section 3.4 discusses potential threats to validity. Section 3.5 presents some related work. Finally, Section 3.6 concludes with our main findings.

3.2 Methodology

We first explain the most important parameters to our usability survey of static analysis tools: Which static analysis tools to evaluate? What evaluation criteria to use? And what exact evaluation procedure to use for each tool?

Collection of Tools While it is infeasible to compile a complete set of all existing static analysis tools, we aimed at gathering a representative set of tools. Starting in October of 2019, we therefore searched several prominent websites giving recommendations for which static analysis tool to use[1]. We found these lists in the scientific literature [SDM20] and by snowballing from these lists, as they also recommend further lists. This process resulted in a very substantial set of tools, even after removing duplicates.

[1] We used the following webpages (last accessed: 2026-06-06):
`https://samate.nist.gov/index.php/Source_Code_Security_Analyzers.html`
`https://security.web.cern.ch/security/recommendations/en/code_tools.shtml`
The CERN webpage on code tools was no longer available.
`http://projects.webappsec.org/w/page/61622133/StaticCodeAnalysisList`
`https://github.com/mre/awesome-static-analysis#multiple-languages-1`
`https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis`
`https://dwheeler.com/essays/static-analysis-tools.html`
`http://www.spinroot.com/static/`

As it would not be realistic to evaluate all these tools, and this would not give a realistic view of the current state of the art either, we decided to apply reasonable inclusion and exclusion criteria to the collected set.

First, we included only tools that consider code in the languages C, C++, Java, JavaScript, C#, and Python, since these appear to be the most used languages [TIO], also being more relevant in the related literature than the excluded languages. After applying this criterion, we obtained a set of 243 tools in total, which were considered in more detail in the next steps. Second, we included only such static analysis tools that are in some way giving warning messages and hinting at coding issues. This includes different analysis types, such as simple linters, bug finders, and more complex security checkers, but not libraries or model checkers. 51 tools do not fulfill this criterion. Third, we excluded four tools for which we were unable to find an executable (or source code), or even a related website. Fourth, we excluded most proprietary tools as we were unable to access them, which accounts for 81 tools. As an exception, to broaden our evaluation results, we contacted the leading companies of proprietary tools mentioned in the Gartner Magic Quadrant [THZ19]. In the end, though, only one of these tools was considered, as the remaining reported tools did not match our inclusion criteria or we were not able to access them after all. Fifth, we excluded 41 tools that were not maintained in the previous two years, as these tools would not represent the current state of the art. In this step, we excluded a tool when following the documentation did not lead to a successful installation and when we did not have any further ideas on how the installation might succeed. Lastly, we had to exclude 20 tools that we were unable to install as described below.

Table 3.1: Overview of tools considered, excluded, and included overall

	C and C++	Java	Python	C#	JavaScript	Multi-Language	Overall
All considered tools	69	52	28	18	23	53	243
Excluded: Proprietary	29	19	0	4	3	26	81
Excluded: Not maintained	15	8	2	4	7	5	41
Excluded: Unable to install	8	4	2	0	3	3	20
Excluded: Other reasons	12	8	7	4	5	19	55
Included	5	13	17	6	5	0	46

Table 3.1 gives an overview of the number of tools collected, excluded, and included overall. Full information on each considered tool is provided in our artifact (cf. Chapter 9).

Some of the considered lists recommending static analysis tools contain a Multi-Language section. If tools from this section have been included in the evaluation, they are mapped to the language we evaluate them in. Accordingly, all tools were mapped to only one programming language even though they might support several

languages. We did not consider excluded tools further or map them to a specific language. The miscellaneous exclusions row includes libraries and frameworks, model checkers, IDEs, tools without static analysis, and tools without any kind of warning messages. Overall, 46 tools are included in the evaluation. Among these, Java and Python account for the biggest share. In comparison to the number of included tools, the number of tools we have not been able to install appears somewhat high. This hints at first usability issues of poor documentation, which is not considered further in this study, but still is a major perceived pain point in this study.

Evaluation Criteria The main goal of our evaluation is to assess the usability of current static analysis tools. To conduct such an assessment objectively, one requires clear criteria. We considered different aspects to derive such criteria: we considered specific features, how the user might interact with the tool, what information is given to the user, and how this information is presented to the user. While some criteria are strongly connected to specific features, other criteria are rather abstract and might be fulfilled in different ways. Regarding the former, our approach is somewhat related to a feature analysis as in DESMET [Kit96], yet regarding the latter it extends further. We aim at defining clear criteria to objectively evaluate the tools' usability, but unlike DESMET are not constrained to the context of an organization and its culture. Furthermore, our main focus is not on the features themselves and on the process of deciding what tool to use, but rather on using these features as a device to evaluate the overall usability of static analysis tools.

Nachtigall et al. [NDB19] have grouped recurrent usability issues into six categories, which are connected to understandable *Warning Messages*, *Fix Support*, *False Positives*, *Integration of User Feedback*, *Workflow Integration*, and a specialized *User Interface*. To expand on this categorization, we searched the related literature and compiled a set of usability issues related to static analysis tools. For collecting criteria, we were inspired by the approach of systematic literature reviews [Kit+09] but adhered to it only in a lightweight form. Particularly, we started with central papers from this area and further used snowballing.

After collecting the usability issues reported by the related literature, we mapped the identified issues to these six categories. Furthermore, we discussed for each issue whether we found reasonable assessment grades. For instance, it would be desirable for warning messages to be clearly understandable [DWA20; Bes+10] or to use natural language [Joh+13]. However, it might be very difficult to objectively evaluate whether a tool fulfills these aspects, as this would require a dedicated

user study with every single tool. If we could not find objective definitions for a criterion and its assessment grades, we excluded it from our study. For all remaining criteria, we defined in advance what exactly has to be evaluated, and we predefined assessment grades. In most cases, these assessment grades are composed of *yes*, *partially*, and *no*, indicating to what extent the corresponding criterion is fulfilled. By April 2020, the evaluation criteria, their assessment grades, and the study protocol had been established. The next section explains the criteria in detail.

Setup and Evaluation We conducted the study between May 2020 and June 2021. Regarding the evaluation of one specific tool, we used the following protocol. First, we searched for the related website or GitHub repository. The websites we collected our tools from usually refer to some of these. On the resulting site, we checked the documentation on the setup of the tool and also on how the tool should be used. We tried to set up the tool on a standardized Windows or Linux VM, depending on which platform the tool supports. If the installation (following the documentation) did not succeed, we checked for alternative solutions until either the tool was successfully installed or we ran out of ideas on how to complete the installation. Due to the high workload involved, usually the installation was done by the first rater, and only if this did not succeed, the second rater also tried to install it, again according to the documentation and potential solution ideas. If both failed, the tool was excluded from the study. Since both raters are experts regarding static analysis tools, it is plausible that others would fail here, too, making this exclusion process appropriate. If the installation succeeded, the tool was evaluated using a multiple-raters approach [SDL80]: both raters evaluated the tool independently of each other, according to the predefined evaluation criteria. For this purpose, we explored the available documentation and attempted to find relevant features and information in the installed tool. Afterward, both evaluations were compared. For coinciding evaluations, we set this evaluation as the final evaluation. In case of disagreement, both raters discussed the source of disagreement until agreement on a final evaluation was reached. For every tool, we used real-world projects and took care to use a project where the respective tool detects many issues. We also used comparable input projects within the same programming languages and analysis scopes.

3.3 Results

In this section, we discuss the used criteria, tool evaluations, and resulting implications in more detail. Table 3.2 provides an overview of all included tools and a summarized view of the evaluation for each tool in each category. The table is divided into command line interface (CLI) tools and graphical user interface (GUI) tools. For this rough evaluation, it is easier to receive partial fulfillment in some cases. For instance, some basic criteria from the workflow integration and user interface categories are fulfilled by many tools. Full results are available in our online artifact. Since the main differences in the evaluations are primarily related to whether the tool is a CLI or a GUI tool, we will repeatedly compare these types of tools. As we found that there is no significant influence of the analyzed programming language, we will not further discuss the language in the following. In the remainder, we will focus on the evaluation criteria, discuss their relevance, and then present our evaluation results.

3.3.1 Warning Messages

Warning messages are the main approach of static analysis tools to point out potential code issues to developers. Therefore, they have to direct the developer's attention to the detected issue and give information on what might be wrong, why it should be fixed, and how it could be fixed. This information has to be presented in a clear way [Bes+10]. Johnson et al. show in their survey [Joh+13] that most of their participants criticize the poorly presented tool output and that it should be presented in a more user-friendly and intuitive way. To analyze this area in more detail, we gathered criteria from the literature and evaluated collected tools.

Internal Reasoning Barik et al. [Bar+14] show that warnings only present the final result of the analysis but do not give any insight into the underlying reasoning, so the developer has to mentally duplicate this process. Barik further argues that static analysis tools are able to computationally expose the internal reasoning [Bar16]. Rule graphs are one way to achieve this [DB20].

In our evaluation, only three tools give at least some insight into the reasoning process. This indicates that related concepts are not as highly prioritized by tool developers as they are by the scientific literature. While it might need quite some implementation effort and is not that relevant for simple linting tools, this aspect is quite relevant for tools that seek to signal complex findings in the code.

Table 3.2: Overview of all evaluated tools (CLI and GUI).

○ : virtually not fulfilled ◐ : somewhat fulfilled ● : clearly fulfilled	Warning Msg.	Fix Support	False Positives	User Feedback	Workflow Integr.	User Interface
Bandit [Ban]	◐	○	◐	◐	◐	○
Cpplint [Cpp]	○	○	○	○	○	○
CScout [CSc]	○	○	○	○	◐	◐
Dlint [Dli]	○	○	○	○	◐	○
Flawfinder [Fla]	◐	○	○	○	◐	◐
Hegel [Heg]	○	○	○	○	◐	○
InferSharp [Inf]	○	○	○	○	◐	◐
McCabe [McC]	○	○	○	○	◐	○
NodeJSScan [Nod]	◐	○	○	○	◐	○
Dependency Check [Owa]	◐	○	●	○	◐	○
Pycodestyle [Pyc]	○	○	○	○	◐	○
Pydocstyle [Pyd]	○	○	○	◐	◐	○
Pyflakes [Pyf]	○	○	○	○	◐	○
Pyre-Check [Pyra]	○	○	○	○	◐	○
Pytype [Pyt]	○	○	○	○	○	○
Radon [Rad]	○	○	○	○	◐	○
Semgrep [Sem]	○	○	○	◐	◐	○
Vulture [Vul]	○	○	●	○	◐	○
Wemake-Python-Style [wem]	○	○	○	◐	◐	○
Wily [Wil]	○	○	○	○	◐	◐
Xenon [Xen]	○	○	○	○	◐	○
Xo [xo]	○	○	○	○	◐	○
Checkstyle [Che]	○	○	○	●	◐	◐
CogniCrypt [Krü+19]	○	◐	◐	○	◐	◐
Commercial Tool	●	○	○	●	●	●
CppCheck [Mar]	◐	○	○	●	○	◐
DevSkim [Dev]	○	○	○	◐	◐	○
ErrorProne [Err]	○	◐	○	○	○	○
Eslint [ESL]	○	○	○	◐	◐	◐
Fb-Contrib [Fb-]	◐	○	◐	○	◐	◐
FindSecurityBugs [Art20]	◐	○	◐	○	◐	◐
Standard [Sta]	○	○	○	○	◐	◐
Mypy [Myp]	○	○	○	○	◐	◐
PMD [PMD]	○	○	○	●	◐	◐
Puma Scan [Pum]	◐	◐	◐	●	○	◐
PyDev [PyD]	○	●	○	○	◐	◐
Pylint [Pyl]	○	○	◐	◐	◐	◐
Pyright [Pyrb]	○	○	○	○	◐	◐
Reshift [Res]	◐	◐	○	○	◐	●
Roslynator [Ros]	◐	●	◐	●	◐	◐
Security Code Scan [Sec]	◐	◐	◐	●	○	◐
SonarLint [Sonc]	●	●	◐	●	○	◐
SonarQube [Sonb]	●	○	◐	◐	●	●
SpotBugs [Spo]	◐	○	◐	○	◐	◐
VisualCodeGrepper [Vis]	○	○	◐	◐	◐	◐
VSDiagnostics [VSD]	◐	◐	◐	◐	○	◐

Clickable Trace or Path Projection Johnson et al. mention the participants' demand for visual outputs supporting the warning [Joh+13]. Approaches for this would be

clickable traces of the relevant intermediate steps or path projections. Similar to internal reasoning, these approaches hint at the main points of the issue and how they are connected with each other. One general approach is presented by Phang et al. [Kho+08]. In their study with FindBugs and Fortify SCA, Ayewah et al. [AP08] point to the benefits of path projection for the understanding and evaluation of a reported bug but also point to its shortcomings in potentially confusing the users.

Our evaluation shows that most of the considered tools do not support any related features. Two tools fulfill this criterion, and three tools give at least some support in this regard, while 41 tools do not point to any intermediate steps in their warning messages. All CLI tools are evaluated negatively in this criterion. Again, this feature is not required for simple linting tools but is of high value in the evaluation of warnings of complex issues.

Explaining Consequences of an Issue Intuitively, if developers are not informed about the consequences of a detected issue, they will probably care less about it, as the importance of the problem may remain unclear. In their interview and observation-based study, Thomas et al. [Tho+16] evaluate their interactive analysis tool ASIDE and discuss the necessity of explaining the issue's consequences. Before approaching a detected issue, it appears to be highly relevant to evaluate whether a found issue is indeed causing problems or exploitable. If this appears not to be the case, developers tend to ignore the issue. Therefore, explaining the consequences of an issue is not only important for understanding the issue, but also for the evaluation of whether the developer considers fixing it.

In our evaluation, the kind and degree of the expected explanation highly depend on the analysis type. For linters considering code style, the consequences are rather trivial. Bug finders or security scanners require more explanation since the context usually is more complex. While the majority of the considered tools (33 of 46) still do not explain the consequences of detected issues, 13 tools mention potential consequences (five *partially*, eight *yes* evaluations). Comparing CLI tools with the GUI tools shows that the fraction of tools not explaining the consequences is higher in CLI tools (86.3% compared to 71.7%).

Explaining by Example Another approach in explaining a problem concept lies in illustrating it with examples. These examples might consider the formation of the issue, its type, consequences, or fix. Explaining the underlying issue with relevant background information using the example helps the developer to get a better understanding of the issue and how to fix it [Joh+13]. In the next step, Smith et

al. [SDM20] argue that providing any example is better than not doing it at all, but also that mismatched examples are another usability issue. When the hard-coded example differs too much from the original code, the developer has to figure out the transferable similarities between them.

In our evaluation, we mainly consider whether there is any example given for the explanation of the issue and the fix. We find that five tools offer reasonable and complete examples, two tools give at least some kind of example for illustration, whereas 39 tools do not use any kind of supporting example. This is worse for CLI tools, where only one tool offers a partial solution. While we agree with Smith et al. [SDM20] that tool developers should aim for matching examples, our evaluation shows that as a first step, even more examples should be included in warning messages.

Offer Further Information One pain point of insufficient warning messages often lies in the issue of presenting insufficient information [SDM20; Smi+18; Joh+13]. Smith et al. [SDM20] consider missing information regarding vulnerability prioritization and fix information, but also mention that the solution would not be to overload the notification with too much information. Johnson [Joh15] argues that offering insufficient information leads to the developer searching other resources, while processing too much information is too time-consuming. Since the context and the developer's expertise are relevant as well [Joh+16], solution approaches might need to offer additional information with more details, e.g., via links [Joh+13], or else adapt to the corresponding developer, e.g., using machine learning [Joh+15].

Almost one-third of the considered tools offer more information in warning messages (three *partially*, eleven *yes* evaluations). In many cases, a link to their website offers explanations about their different warning categories and more detailed information about the detected issue. Still, the ratio of tools offering more information is a little worse in CLI tools (one *partially*, three *yes* evaluations).

Warning Connection to Code As mentioned above, Smith et al. [SDM20] report issues related to reports that are disconnected from the code. Warnings often consist of mismatched examples or a main template text, where only names of the variables, methods, etc. are changed. As errors from the same warning category with the same warning message still might be unique, the developer still has to process the message further, in addition to the other challenges the developer faces while understanding the warning. Thomas et al. [Tho+16] find that developers wish to integrate the warnings within the code and contextualize them further. Other developers desire

the use of more natural language [Joh+13]. To pursue natural explanations, Barik et al. [Bar+18] analyzed the communication on Stack Overflow to find out how human developers explain code issues.

We consider a message as generic if the tool always states the same error message for a specific error or only adapts the names of parameters or methods. Accordingly, we only found three tools that really fulfill a closer connection to the codebase, while 43 tools are more generic. In many cases, the content of such messages is also very short, which makes it even more complicated to understand and apply the warning message to the codebase. In these cases, the developer is roughly informed about the issue and knows where to find the issue. However, such generic messages often lack information and context and hinder the developer in understanding what exactly is wrong with the code.

Unique Weakness Identifier If the developer would like to learn more about a specific code issue outside the static analysis tool, a unique weakness identifier is valuable as it helps the developer recognize it or search for information. Security vulnerabilities are usually categorized according to Common Weakness Enumeration (*CWE*) references [Mita]. In other contexts, such categorization is not that clear, for instance in code style linters or non-security bug finders. Still, such categorization according to globally unique identifiers would improve the search for further information for specific issues outside the used tool.

Our evaluation validates this description of unique identifiers: security static analysis tools (ten tools) categorize issues according to the CWE, while this is difficult to achieve in other contexts. Three tools find partial solutions for unique identifiers, which are not as well-defined as CWEs but still help with clear identifiers. Again, this aspect is not as often implemented in CLI tools as in other tools. This is probably connected to the higher amount of simpler tools (e.g., code style linters) in CLI tools than in more powerful and complex tools.

Error Information In our last aspect related to warning messages, we evaluated whether the tools answer some of the central information categories that are relevant to understand, evaluate, and fix the issue. Therefore, we consider the categories of the description, existence of a general warning categorization, localization of the found issue, and whether there is a severity evaluation. The latter is among the most relevant factors when selecting warnings [Vas+18].

Though the implementation of an error description in the warning message might appear trivial, we find that there are quite a few tools using very unclear warning messages. In these cases, there is a general warning text, but no description of what is wrong with the code or what the developer might want to change. Sometimes there is a description of a part, but no explanation of what is wrong with it (e.g., code style linting tools describing the length of a code line or outputting a number related to code complexity). In ten tools, we did not find a description of a code issue, and in eight cases, the description is insufficient. Again, CLI tools stand out with a higher fraction of no descriptions or partial descriptions. Only about 40% give a satisfying error description in *CLI* tools, while this fraction lies around 60% for *GUI* tools.

Next, we evaluate whether the tool associates a unique warning category with every warning message. Compared to the unique identifier, no global unique identifier is required, but a rough classification of the issue types the static analysis tool considers. Such categorization helps in our evaluation as it roughly hints at the error type and helps the developer to recognize and connect experienced warning messages over time. 32 of the considered tools apply clear error categories, as opposed to twelve tools not categorizing errors accordingly (two *partially* evaluations). The share of CLI tools fulfilling this criterion is somewhat smaller, but still more than half of the CLI tools use error categories (about 59% compared to about 69%).

In the next criterion, we evaluate whether the warning messages hint at the place of the issue. In case of issues covering several intermediate steps, we ignore intermediate steps, as this is already covered above. Instead, we only consider the main location of the error and evaluate its granularity, using the evaluation grades *line of code*, *method*, *class*, and *project*. 40 tools hint at the exact *line of code*. Since there are some issue types that are not related to one specific line but rather to methods or classes (e.g., dependency checker), there are good reasons for a rougher localization.

Last, we check severity evaluations of warning messages. Severity supports the developer in the evaluation of a notification and for the prioritization of fixing errors. The distribution of tools fulfilling this criterion is somewhat balanced since 21 tools do not use severity classifications, whereas 20 tools do (five *partially* evaluations). In CLI tools, the distribution is very different, where four tools fulfill this criterion, and 18 do not.

Summary Overall, our evaluation of the static analysis tools' warning messages very much confirms the usability issues presented in the literature. For most of

the categories, the majority of tools do not support the required features. Warning messages in CLI tools tend to be worse compared to GUI tools. We found too many tools with very short warning messages and hardly any documentation about detected issues and what might be wrong with the code. Then again, there are also good examples for all considered aspects. This leads to the assumption that there are known approaches to address most usability issues but only a few implementations of these approaches.

Observation 3.1 – Warning Messages

1. More than half of the tools have poor warning messages (30 of 46, see Table 3.2).
2. Three tools give good warning messages, which might inspire more tool developers.
3. Hardly any tools give information about why they evaluate something as an issue (giving traces or paths, internal reasoning).
4. Explaining the code issue with details exceeding the most basic aspects also remains a common problem.

3.3.2 Fix Support

The fix support category is related to warning messages, as a good explanation already leads to potential fix ideas. Heckman et al. [HW09] try to classify alert messages as actionable and non-actionable using machine learning techniques and continue to synthesize available research results about main approaches for actionable alert identification techniques in a systematic literature review [HW11]. Accordingly, static analysis tools should not only inform the user about potential issues but also add further information on how to fix the issue. Sadowski et al. [Sad+18] affirm the challenge of unactionable alerts: One of their main lessons learned is that tools should not just find bugs, but also fix them. Smith et al. [SDM20] evaluate four security-oriented static analysis tools and conclude that static analysis tools do not support the developer well enough to fix the issues, which even might lead to the abandonment of static analysis tools. Therefore, we sought to analyze the fix support features of the considered tools.

Quick Fix Regarding quick fixes, the scientific literature points to two main issues: the lack of implemented quick fixes and the lack of trust in existing quick fixes. Nguyen Quang Do et al. [DWA20] state that quick fixes are one of the most popular features of static analysis tools, based on their survey of developers. Johnson et al. [Joh+13] find the lack of or ineffectively implemented quick fixes as the most frequently mentioned difficulty in their study. This lack of effectively implemented quick fixes may diminish trust: some developers trust their own solutions more than solutions offered by the tool [Tho+16].

We considered it too difficult in our context to evaluate whether offered quick fixes are implemented too ineffectively. Hence, we only check whether the tool offers quick fixes at all. We find that 36 tools do not offer any fixes to the developer (see again Table 3.2). Only one of 22 CLI tools offers quick fixes. While this shows that the overall number of tools not offering quick fixes is too high, almost half of the GUI tools offer some partially or fully satisfying quick fixes. While it is technically challenging to offer quick fixes, especially for more complex issues, these numbers show that it is still possible to offer some help to the developer, but also that we can still do better by offering more fix support.

Alternative Solutions In many cases, there is not only one possible solution for a detected issue. Depending on the developer's preferences, the development contexts, and other aspects, one solution might be better fitting than other solutions. Therefore, understanding alternative fixes and approaches is one of the main questions developers ask while diagnosing potential security vulnerabilities [Smi+15; Tho+16]. Instead of quick fixes, Barik et al. [Bar+16] propose the idea of slow fixes, in which the developers are supported in exploring different solutions and balancing the benefits of manual and automated fixing.

In our evaluation, we find that only four tools consider alternative solutions. Two of these tools are evaluated as partially fulfilling, as they offer alternative solutions in some cases and only one option for other issues. None of the CLI tools consider alternative solutions. This shows that even though there is an interest from the developers to consider alternative solution approaches, this is implemented by hardly any tool developers.

Fix Preview Connected to the need for exploring the space of possible solutions is the question of whether offered fixes are applicable in the given code context. Due to the mentioned trust issues towards quick fixes, developers might be afraid that the fix might break the code in other parts [Tho+16]. Therefore, a fix preview

helps to evaluate whether the fix causes undesired side effects and to assess the application of the fix in the given context [Smi+15]. Johnson et al. [Joh+13] also support that developers would prefer fix previews instead of directly applying the fix.

Just like the previous criterion, none of the CLI tools offer a fix preview. While there are a few other tools implementing this feature (one *partially*, four *yes* evaluations), this only constitutes a small fraction of the overall considered tools (about 11%).

Fix Example Corresponding to the explanation of the issue in warning messages using examples, examples also might be used to explain and illustrate solution approaches. Similar to fix previews, Johnson et al. [Joh+13] report that some participants of their study would prefer the use of examples to get a better understanding of how to fix the problem. Hartmann et al. [Har+10] state the hypothesis that relevant solution examples help novices to interpret and correct error messages.

In our evaluation, we find that 21 of 22 CLI tools do not present any fix examples to the developer. Similar to *Fix Preview*, five of the remaining GUI tools provide fix examples to the developer, but this represents a small fraction as well.

Fix Tutorial From the perspective of a warning message, explaining all necessary steps the developer has to take to fix the issue appears to be relevant and useful. Nachtigall et al. [NDB19] present the idea of an interactive system between the static analysis tool and the user, in which the tool takes the role of an assistant and of a teacher. The teacher guides the developer through fixing all detected issues and gives all necessary information at every state of the fix. Johnson et al. [Joh+15] aim to model the user's knowledge and adapt the tool. The tool would present the required knowledge accordingly and might support the user with the next steps to fix an issue.

As before, only one CLI tool gives partially satisfying instructions on how to fix the reported issue. With regard to the remaining GUI tools, there are five more tools (one further *partially*, four *yes* evaluations) giving more detailed guidance on how to fix the issue. However, those tutorials are not comparable to, for instance, the assistant discussed before.

Summary The *Fix Support* category appears to be a much neglected area. This may be due to technical challenges in the implementation of corresponding features. Yet, the literature shows that giving more fix support is a highly relevant task for tool

developers. Only reporting potential issues is insufficient if the developer is not able to fix them afterward. Throughout our considered criteria, CLI tools offer almost no support. GUI tools reach somewhat better results. Nonetheless, the number of tools fulfilling these criteria is very low. Not every tool needs to fulfill every aspect since these approaches aim for the same goal. Overall, our evaluation exposes definitive challenges for future work.

Observation 3.2 – Fix Support

1. With 37 tools giving almost no fix support, this appears to be the most neglected category.
2. Three tools give sufficient fix support.
3. Only ten (partially) offer quick fixes. Even fewer tools support this with alternative solutions or previews.

3.3.3 False Positives

One major reason why developers do not use static analysis tools is the high number of false positives. Johnson et al. [Joh+13] discuss the consequences of having too many false positives. Accordingly, false positives outweigh true positives and lead to dissatisfaction. Christakis et al. [CB16] stress that high rates of false positives lead to disuse of the analysis. Furthermore, an analysis of questions on Stack Overflow related to static analysis tool alerts reports that false positives are among the most prevalent topics [Imt+19]. As the existence and occurrence of false positives mainly is a technical problem related to the implementation, and we focus on usability aspects, we consider how developers might deal with false positives.

Suppression of False Positives One mechanism to avoid some false positives lies in suppressing false positives. By telling the tool that a specific warning is a false positive, the tool might ignore reporting the same issue again or even use this information for a user feedback-based ranking of warnings [MS16]. The high relevance of suppression mechanisms for false positives is stressed in several publications [CB16; Joh+13; Aye+08].

In the evaluation of this criterion, we only consider explicit ways of reporting false positives to the tool. Using implicit mechanisms like code annotations is valid as well as desired by developers, but will be considered in the temporary suppression

criterion as code annotation is used in more contexts than for false positives. We find that overall twelve of the considered tools offer the possibility to explicitly report false positives, see again Table 3.2. Only two of these twelve tools are CLI tools.

Confidence Evaluation Static analysis tools often add further metrics to the warning message for evaluating whether it is a false positive, one of which is a confidence evaluation [DWA20]. Tools such as FindBugs associate a confidence factor with each warning [Aye+08], which allows the developer to filter by confidence and hence support the developer in focusing on relevant warnings.

In our evaluation, we find that six tools offer such confidence evaluation, including three CLI tools. Three of these six tools are extensions of FindBugs, which already offered confidence evaluation before being extended. This emphasizes how rarely this mechanism is considered overall.

Ask for User Knowledge Sometimes the analysis has incomplete knowledge about the context in which the analyzed code is executed and erroneously reports an issue due to the analysis' over-approximation. To avoid this, Zhu et al. [Zhu+15] propose to explicitly ask for the developer's knowledge in the context of access control. This approach might also be transferable to different contexts. Furthermore, asking for user knowledge might help in modeling the user's knowledge and in using this information to adapt the tool to the developer's preferences [Joh+15]. To support developers in focusing on important warnings, Bodden et al. [BLH08] employed machine learning to classify and reduce false positives reported by their static analysis tool. Tripp et al. [Tri+14] present a similar approach, where the tool applies feedback and statistical learning to improve reports. As mentioned above, a static analysis tool also might collect the user's feedback on whether a report is a false or true positive for future reports [MS16].

We evaluate eight tools to partially fulfill this criterion, while all remaining 38 tools offer no related features. Again, the evaluation is worse for CLI tools, where no tool asks for the user's knowledge. While this feature might be less prioritized than other features, our results indicate that this concept is still in its early stages and is hardly used in practice.

Summary Our observations in the category of false positives indicate that usability-related mechanisms are hardly used to avoid false positives. For each of the considered criteria, there are tools considering them, but overall these mechanisms

are implemented too rarely. Considering the high impact of false positives on the developers' satisfaction, this poses many tasks for future work.

Observation 3.3 – False Positives

1. Only twelve tools let the developers explicitly report a warning as a false positive.
2. Overall, 15 tools fulfill the considered criteria partially or better, while 31 are weak in this aspect.

3.3.4 User Feedback

The category of how false positives might be handled from the perspective of usability is related to the perspective of the user's feedback. Hence, some examples of how user feedback might be integrated in static analysis tools are mentioned above. In this section, we consider some further, more general use cases, in which the integration of the user's feedback is relevant. The more the analysis adapts to specific users, the more useful it will appear to them [Joh+15; Joh15].

Customizability of Analysis Rules Johnson et al. [Joh+13] point out that customizability, in general, is a reason why developers do not use static analysis tools, including analysis rules, but not restricted to them. Nguyen Quang Do et al. [DWA20] confirm this demand for the customization of analysis rules. Not all rules should be turned on by default, and the process of selecting rules should be made easy [CB16].

In our evaluation, we check whether the tools allow the developer to *turn specific analysis rules off*, *modify existing rules*, or *introduce their own rules*. These options are not mutually exclusive and allow for multiple selections. By doing this, we find that 19 tools do not offer any customization of the analysis rules. With 14 tools, the majority of these tools are CLI tools. The remaining 27 of all considered tools offer the option to switch off specific rules. 19 of these 27 tools also allow modifying existing rules, and 17 of them allow introducing their own rules. These fractions indicate a positive tendency towards allowing customizability of analysis rules.

Validation of True Positives As seen above, it is not only useful to report false positives to the tool, but also to validate true positives for user feedback-based

self-adaptive ranking [MS16]. Accordingly, warning types with a higher fraction of true positives and fewer false positives might be prioritized higher.

Regarding the 46 selected tools, only two tools offer explicit validation of true positives, none of which are CLI tools. In other cases, this might only be done implicitly by fixing the error or by assigning a responsible developer for fixing it. However, this information is not used for validation purposes in any further tools, according to the corresponding tools' documentation.

Temporary Suppression The relevance of suppression mechanisms for false positives is described above. Many developers prefer to do this via code annotations [CB16; Joh+13]. Code annotations allow for temporary suppression, which is relevant when the corresponding code part is still under development and warnings would rather be distracting than supporting [Joh+13]. The implemented code annotations usually do not refer to false positives, but just generally ignore a line of code or a whole code section.

In our evaluation, we find that 19 tools provide the described features, while 27 tools do not offer temporary suppression, see again Table 3.2. Eight of these 19 tools are CLI tools, which is a comparable share between CLI tools and GUI tools.

Filter Alerts An evaluation of Stack Overflow questions about static analysis tool alerts shows that ignoring/filtering alerts is the most discussed issue [Imt+19]. This shows that developers try to use these features but also that there are challenges in using these features.

In ten tools, we find the option to filter alerts. Four further tools partially support this feature, which means that these tools generally offer filters but are limited to very few filter options. In contrast to this overall result, 21 of 22 CLI tools do not offer alert filtering.

Summary Overall, we find different results in this category. On the one hand, there are hardly any tools offering implicit validation of true positives, which would allow for adapted error reports. On the other hand, the majority of tools allow customization of analysis rules, with different options. The fraction of tools supporting alert filters and temporary suppression is somewhere in the middle, neither too small nor high enough.

Observation 3.4 – User Feedback

1. Eight tools strongly support the integration of user knowledge, while ten tools partially fulfill it.
2. The majority of tools allows switching off rules, while 19 tools also allow further modification of the analysis rules.
3. 19 tools support temporary suppression of warnings.
4. Filtering alerts and validating true positives are implemented in a clear minority of tools.

3.3.5 Workflow Integration

The integration of the static analysis tool's support into the developer's natural workflow is one of the main challenges of usable tool support and one of the main reasons why many users are dissatisfied [Joh+13; Oye+18; BN18]. As workflow integration is a key aspect for the use of static analysis tools [Sad+15], Sadowski et al. [Sad+18] conclude that workflow integration should be pushed as early as possible.

Warning Prioritization The larger the analyzed codebase, the more warnings might be reported. In real-world projects, the number of warnings is often too high to process for a human developer. Hence, developers demand automatic warning prioritization by the tool [DWA20; CB16], so that they know which bug to select. Many tools fail in providing enough information in this aspect [SDM20], which again leads to the user's dissatisfaction.

Twelve tools report the detected issues in a prioritized order. Two more tools also consider prioritization but are rather rough by just offering a few priorities. The remaining 32 do not prioritize their warnings. Regarding CLI tools, three tools fulfill this criterion, while 19 do not. As warning prioritization requires some kind of evaluation of priorities, this requires more complex processing. Still, this should be worth the effort as it makes the static analysis tool effective [Joh+13].

IDE Integration IDEs or more powerful editors are the standard development environment. As this might be the closest connection to the developer's workflow,

static analysis tools should be integrated in IDEs to offer better workflow integration [DWA20; Sad+18].

Almost half of the considered tools (20 of 46) offer plugins for one or more IDEs in the respective programming language, see again Table 3.2. The one reported CLI tool fulfilling this criterion offers some plugins, which we were unable to install. Some IDEs offer unified tool integration (like IntelliJ [Jet]), while other IDEs allow more options in offering and accessing the tool's features (like Eclipse [Ecla]).

Standalone Tool In this criterion, we evaluate another type of tool's accessibility and check whether it is possible to use it as a standalone tool. Therefore, we inspect whether it is possible to install, configure, and execute the analysis on its own. We find that nearly all tools work as standalone tools as well. Only six tools do not fulfill this criterion, for instance, tools that can only be integrated with Maven [Apab] or other build tools.

Browser Access Furthermore, there are tools accessible in typical browsers, which include only services as well as locally hosted analysis servers, accessible via browser. Four of the evaluated tools offer this access type. While this option is not directly integrated into the workflow, this setting usually allows for more presentation possibilities of the analysis.

Disjoint Process of Understanding and Fixing Offering different access types is an advantage as the developer might choose according to their own preferences. However, options that are not directly available in the developers' coding environment lead to a disjoint process of understanding and fixing reported issues [Joh+13]. This means that the developer has to open tool reports to see and understand the warnings, while they have to switch to the coding environment afterward to see and work on the corresponding code. As this process is ineffective and disruptive, developers tend not to use such tools [Joh+13].

Overall, the evaluation of this criterion completely overlaps with the question of whether the tool supports IDE integration, including respective editors. Since all evaluated IDE integrations allow displaying the warning message and the corresponding code part at the same time, the process of understanding and fixing the issue is not disjoint. In all remaining tools, this process is disjoint as the warnings are reported outside the coding environment.

Runtime of the Analysis The longer the runtime of the analysis, the more difficult it is to integrate the analysis into the developer's workflow. Johnson et al. [Joh+13] report in their study that developers complain about analyses taking too much time, which disrupts the flow. Accordingly, analyzing a larger codebase is even more problematic as it requires more time. These problems might be slightly avoided by running the analysis at different stages of the software development lifecycle, like during nightly builds or at major project milestones [DWA20]. While these options are valid, they are also not integrated into the developer's workflow as well as running the analysis at coding time.

In our evaluation, we tried to use comparable projects within each considered language. For some tools, we were not able to run the analysis on the complete project but only on one directory or even one file, because of the otherwise too high runtime. While this makes comparison harder, it points to another usability issue. Independent of the exact size of the analyzed code, the resulting runtime still allows for an evaluation of whether a corresponding tool might be useful during coding time or only during nightly builds or at major project milestones. Around half of the tools required less than a *minute* for their analysis (22 tools). Such a runtime most likely allows execution during coding time. Twelve tools require *more than one minute*, nine tools *more than five minutes*, and one tool each for *more than ten*, *15*, and *30 minutes*. Five tools were not evaluated in this criterion as we were unable to execute the analysis on the whole project.

Feedback on Fixed Issue After spending some time fixing reported issues, the developer might require feedback about whether these issues are really solved. Giving specific feedback about fixed issues gives confirming and motivating information to the developer, which might keep them engaged and more satisfied with the tool's overall experience [DB18].

In our evaluation, we only find two tools giving explicit feedback about fixed issues. For all remaining tools, the developer has to remember the warning, rerun the analysis, and check whether the issue is still reported. Hence, these results show that this idea is hardly used in practice and rather less prioritized by tool developers, even though such feedback would support and motivate the developer's work.

Summary Concluding the category of the static analysis tool's *Workflow Integration*, our evaluation confirms several challenges from the literature. Offering different ways to access a tool is a good way to adapt to the developer's preferences. Our results show that a high number of tools are only available as CLI tools. As shown

through our whole evaluation, CLI tools are weaker than the other options in most usability-related aspects. It should be the goal to get as close to the developer's natural workflow as possible, which is reached through IDE integration. This is done by roughly half of the considered tools. Also, the aspects of warning prioritization and feedback on fixed issues pose open challenges for static analysis tool developers.

Observation 3.5 – Workflow Integration

1. 22 of 46 tools are only available via CLI, which is highly problematic for *workflow integration* without further integration into user interfaces.
2. 22 of the tools also need more than one minute for the analysis, which might already be too long for integration into the normal coding process.
3. Overall, we evaluate 36 tools to partially allow workflow integration, while only two tools stand out positively.

3.3.6 User Interface

In this final main category, we evaluate how the tool generally presents all of its interaction options to the developer. While the user interface is not the core of static analysis, it might still distract or support the developer. To support developers in their work with static analysis tools, there are different approaches for improving the user interface [MS16]. In the following, we evaluate central aspects of supportive user interfaces.

Highlighting in Code and Warning Icons Highlighting the error in the code view and adding warning icons are commonly known mechanisms to draw the developer's attention to code issues. While only highlighting code would not be enough, it is still useful for many developers, as are warning icons [Ngu+17]. Apart from error locations, highlighting and icons might also be used for other related code locations, such as intermediate steps of how the error evolves through the code, locations where the analysis might require the developer's knowledge, or wherever the tool might want to guide the developer's attention [Xie+11; KM04].

16 of the considered tools highlight errors in the code and twelve tools integrate warning icons, see again Table 3.2. At first sight, these fractions might appear small, but they are downgraded by the observation that no CLI tool supports either code

highlighting or warning icons. Hence, the fraction of GUI tools supporting code highlighting is two-thirds, while this fraction is a little lower for warning icons.

IDE or CLI Conformity In their handling of static analysis tools, developers have specific expectations based on experience and on other tools communicating in similar contexts [Joh+16]. Accordingly, developers expect the tool to behave similarly to how they know it. Therefore, we evaluate whether the considered tools report their findings in a similar way to other tools or compilers. While doing this, we also try to maintain the context of IDE or CLI presentation of warnings. We evaluate that overall eight tools deviate from the typical communication patterns, five of which are CLI tools.

Overview of Warnings In huge codebases with many detected issues, it is difficult for the developer to get an overview of the main results of the analysis. To support the developer in doing this, several static analysis tools process the overall results in dashboards or similar warning overviews [Sad+18]. Nguyen Quang Do et al. [DWA20] support the demand and value of dashboards and report that the developers often look at them to understand the project's health before considering the single warnings.

Eight of the evaluated tools offer a dashboard or a comparable overview. The remaining tools only present the list of all warnings to the developer, sometimes adding a few simple metrics. Though CLI tools are limited in their graphical expressiveness, still, five CLI tools offer comparable overviews.

Tracks Progress Many tools only report currently existing issues found in the codebase, but do not refer to older analysis runs to report fixed issues [SDM20]. Hence, developers demand features that keep track of the progress, which also give a clear view of what is achieved [DWA20].

In our study, we only find two tools offering such features, while this is not the case for the remaining 44 tools. In the latter case, the developer needs to manually keep track of the progress.

Search through Warnings When the list of reported issues is too long for developers to survey, they might want to search through the warnings. Using different metrics, the search might support the developer in finding relevant issues to fix, for instance, issues the developer has specific knowledge about. Based on comparable use cases,

developers confirm the demand for features for searching through warnings. We find ten tools offering searching features, none of which are CLI tools.

Summary Overall, our evaluation of user interface-related criteria yields mediocre to weak results. Most of the GUI tools guide the developer's attention using code highlighting and warning icons, while this is ignored by CLI tools. Most tools also present their results in typically used patterns. Still, the last three criteria also pose several tasks for future work since only a few tools offer sufficient warning overviews, progress tracking, and search functions.

Observation 3.6 – User Interface

1. Code highlighting and warning icons are completely neglected in CLI tools.
2. Searching through warnings and tracking progress are highly neglected across all tools (ten and two tools fulfill them, respectively).
3. 26 tools support at least a few of the considered criteria, while three tools get a good overall evaluation in this category.

3.4 Threats to Validity

We are aware, and sought to mitigate, the following threats to the validity of our study.

For tools supporting multiple operating systems, there is a bias towards the Windows version. Only if we were unable to install a tool on Windows did we try to install it on the Linux VM (where Linux versions were available). We see this bias as unproblematic, however, because Windows still has, by a large margin, the largest market share on desktop systems [sta].

Overall, the evaluation process of all tools took more than one year. We always tried to evaluate the most current version of the tool, depending on the time when we considered the respective tools. This might lead to the issue that some tools we considered earlier might have been updated in the meantime. If these updates affected features that have been evaluated in an earlier version, our evaluation might become slightly obsolete in the respective criteria. We see no way to avoid this.

Although the evaluation criteria and grades were established in advance of the study, the exact delineation between grade levels can be debatable in individual cases. Hence, it might happen that we assigned different grades to comparable fulfillment of a respective criterion in some cases. To mitigate this problem to the largest extent possible, we used a multiple-raters approach (see Table 3.2).

While we try to evaluate the current state of static analysis tools, it appears infeasible to collect a complete set of all relevant tools. As described above, we attempted to collect a representative set of tools in order to be able to obtain the most meaningful results possible. Nevertheless, our collection might miss a few relevant tools.

The same also holds for our selection of evaluated criteria. Literature surveys with a different methodology might result in further relevant aspects related to usability. When determining the list of criteria, we sought to find a good trade-off between feasibility and a meaningful representation of relevant usability aspects.

As described above, several tools have been excluded from the evaluation as we were unable to install the respective tool. We tried to install the tools with reasonable effort, depending on the documentation and our own knowledge. While we were unable to install these tools, this does not generally mean that it is impossible to install them.

3.5 Related Work

Much related work was already discussed above. Overall, many publications mention usability issues with static analysis tools, yet some seek to give a wider and more complete view of the usability of static analysis tools. We focus on such studies here.

Johnson et al. [Joh+13] research why static analysis tools are not widely used by developers and how this might be improved, also discussing further implications. Their results are based on interviews with developers. Christakis et al. [CB16] discuss what developers want and need from program analysis by surveying developers across Microsoft. They mainly focus on the following three aspects: 1) barriers and reasons why developers stop using analysis, 2) functionality the developers want in analysis, and 3) non-functional characteristics a program analyzer should have. Based on a survey within Software AG, a leading international software company, Nguyen Quang Do et al. [DWA20] analyze the analysis tools' integration in the development environment, the usage context of analysis tools, developers' strategies

in working with analysis tools, and features the analysis tool should provide. These publications are related in that they seek to find usability issues and reasons why developers would use or reject static analysis tools. This chapter, however, rather seeks to use criteria mentioned in these papers to evaluate the current state of usability in static analysis tools.

Nachtigall et al. [NDB19] summarize the main design challenges for building usable static analysis tools and very roughly evaluate 14 tools against the introduced six main challenges. While we revolve around their six main challenges, we split these categories up into more detailed criteria and also execute a more thorough evaluation of 46 tools. Smith et al. [SDM20] execute a heuristic walkthrough evaluation of four tools. In this approach, they familiarize themselves with the tools, like in a cognitive walkthrough. In the next step, they explore the system using a set of usability heuristics, like in heuristic evaluations. They reveal several usability issues and group them into six themes. While their work is restricted to four tools, we evaluate a larger set of tools.

3.6 Conclusion

This chapter presented a large-scale assessment of user interactions offered by static analysis tools, the first main contribution of this thesis. As the scientific literature points to many usability issues related to static analysis tools, we sought to evaluate to what extent these aspects generally apply to the current state of the art. Therefore, we defined a set of relevant criteria, collected relevant static analysis tools, and applied the criteria to the tools. Our tool-based perspective is complementary to the mainly survey- and interview-based view from previous work, yet empirically confirms many previous findings and suspicions.

Therefore, our results indicate substantial future work for static analysis tool builders but also reveal the need for further research in the area of static analysis. Overall, the areas of explaining *Warning Messages*, *Fix Support*, handling of *False Positives*, consideration of *User Feedback*, *Integration* into the developer's *Workflow*, and supporting *User Interfaces* are still much neglected. For CLI tools, these challenges are even more problematic in all categories. This leads to the developer's dissatisfaction and might make them abandon support from static analysis.

The results of this empirical study have identified key foundations for the goal of this thesis in helping Java developers reduce cryptographic API misuses. Having a

well-designed user interface that is integrated into the developer's workflow is essential; in this thesis, this is further addressed in Chapter 7 with *SecAI* (*Contribution 5*). The usability criteria, usability categories, and results of this study systematically guided the design and development of *SecAI*. All features of *SecAI* were systematically derived from the usability criteria and categories, and technically self-assessed following the approach of this study. The most neglected usability category in this study was *Fix Support*, which is addressed in Chapter 6 with *LLM-SAST*$_{Repair}$ (*Contribution 4*). Foundations to address the usability category *Warning Messages* with increased explainability and internal reasoning are laid in the next two chapters by investigating API misuses in general in Chapter 4 (*Contribution 2*) and cryptographic API misuses in Chapter 5 (*Contribution 3*). The usability criteria and categories identified in this chapter form the foundation to systematically design the final main contribution of this thesis, *SecAI* (*Contribution 5*, cf. Chapter 7), which combines all main contributions of this thesis in one tool.

4 FUM — A Framework for API Usage Constraint and Misuse Classification

In the previous chapter, fundamental insights into the usability of static analysis tools were discussed. One key usability category encompasses *Warning Messages* (cf. Section 3.3.1) and improving their explainability to better support developers. To increase the explainability of warning messages of static analysis tools for the specific domain of cryptographic API misuse detection, it is pivotal to understand API misuses in Java and their origins in general. APIs are the primary mechanism that developers use to obtain access to third-party algorithms and services. Unfortunately, APIs can be misused, which can have catastrophic consequences, especially if the APIs provide security-critical functionalities like cryptography. Understanding what API misuses are, and for what reasons they are caused, is important to prevent them, e.g., with API misuse detectors. However, definitions and terms for API misuses and related terms in literature vary and are diverse.

In this chapter, we contribute a systematic literature review of Java API misuses from which we derive tangible definitions and *FUM*, a novel *F*ramework for API *U*sage constraint and *M*isuse classification (*Contribution 2*, cf. Chapter 4). We apply *FUM* in a case study on *CogniCrypt* to systematically assess the capabilities of *CogniCrypt* and ways to improve it. The results of the second main contribution of understanding Java API misuses in general can also be applied to better explain cryptographic API misuses, e.g., by detecting misuses that propagate in *multi-object protocols*. This is essential to build explainable *Warning Messages* (cf. Section 3.3.1) that developers act on and thus reduce *effective false positives* (cf. Section 2.2.3).

4.1 Introduction

Software reuse is one of the fundamental principles of good software development [Hei+11]. To facilitate such reuse, developers can take advantage of previously

written functionality exposed as application programming interfaces (APIs). For example, consider Java, which comes bundled with the *Java Class Library (JCL)* [Orag] — a set of APIs that, for instance, provide basic functionality for the convenient use of data structures and allow comfortable communication with I/O devices. Besides language maintainers such as Oracle, which provide standard APIs, there is also the community of developers that supports the idea of sharing functionality, e.g., by providing APIs as free or open-source software; for instance, the Maven repository [Apab] consists of thousands of APIs provided by a huge community that a developer can benefit from using. On average, a Java project depends on 40 different libraries [MKP23]. Consequently, developers often are composers of an arrangement of APIs instead of implementing the desired functionality from scratch.

However, the use of APIs has its drawbacks. Recent studies show that developers often face several difficulties [SHA15; Nad+16; Laz+14; Isl20; Gu+19], such as the lack of appropriate documentation, the complexity of the API, or even an inadequate level of abstraction [Nad+16]. As a result, developers tend to incorrectly integrate APIs into their code, which leads to misbehavior, errors, or even program crashes. Research [Ama18; Ama+16; Li20; Luo+18] has shown that the misuse of APIs goes beyond specific and complex domains like cryptographic APIs; misuses occur across all API usages. Amann et al. [Ama+16] examined the prevalence of API misuses in several bug datasets [HJZ13; JJE14]. They found that 89 of all 1189 reviewed bugs were API misuses (7.49%); however, 69.5% of these caused the program to crash. Although the number of API misuses found in the datasets was small, the high probability of causing a crash underlines the need for API misuse detectors. Furthermore, Li [Li20] performed a systematic and extensive empirical study of API misuse mining on GitHub [Gitc]. They found that 50.6% of all bug-fixing commits between 2011 and 2018 were related to API misuses. Misuses of APIs like the JCA are quite common [Nad+16] and tend to have catastrophic effects because of their security-critical nature, even if they do not cause crashes. Moreover, even if high-quality documentation is provided, this may still not be enough for developers to avoid API misuses altogether [Ama+18]. To reduce API misuses, developers need more support, e.g., in the form of API misuse detectors [Krü+19; Ama+18; Rah+19; Ege+13; Shu+14; Ngu+17; MBB18; Pra+12; Ngu+15; MCJ17; HH06].

Using misuse detectors can be very beneficial for developers. However, they are only infrequently adopted in practice, because of a lack of analysis quality (e.g., precision and recall), usability issues [CB16], and the Achilles' heel of static analysis: false positives [Joh+13]. Among other things, a main contributing factor to this is the quality of reporting, which, in turn, originates from a lack of a standardized taxonomy of API misuse classifications, their severity, etc. The acuteness of this problem

was discussed by Robillard et al. [Rob+13], who surveyed a decade of studies on API misuses, where the authors state "[...] we observe a lack of uniformity in terminology and definitions for the properties inferred by various approaches". Although they observe the need for uniformity, their survey was aimed at understanding and categorizing the API usage inference techniques and not at standardizing the misuse classification terminologies themselves. The first approach for this was provided by Amann et al. [Ama+18] who studied API misuses in the wild, classified them, and introduced a taxonomy called the *API-Misuse Classification framework* (*MuC*). However, they derived *MuC* empirically from API misuses instead of building the taxonomy solely based on a theoretical foundation.

This chapter presents a literature survey on API misuses and classification taxonomies to understand the state of the art and to derive required definitions and an API misuse classification framework. Specifically, we exhaustively studied the previous work of Amann et al. [Ama+18] with *MuC*, refinements that Li [Li20] made to *MuC*, and the work by Monperrus et al. [Mon+12] who categorized the different types of API directives in API documentation. To derive a *concrete* classification framework, this work focuses on API misuses in Java programs. Java is one of the most popular languages [TIO], and many of its features are representative of those in other commonly used languages. Moreover, many previous studies on API misuses have been performed on Java APIs, which gives us the ideal starting point for both comparison and expansion.

To arrive at our taxonomy, we first conducted a scientific literature study, seeking to answer the following research question:

RQ1 *How is the term API misuse defined, delimited, and made tangible?*
To improve the development of analysis tools for API misuse detection, it is necessary to understand what API misuses are and how they are caused. This requires a theoretical foundation with clear definitions and an exploration of the sources of API misuses inside the program, i.e., to identify the nature of API misuses from the perspective of the API designer/expert.[1] We conducted a systematic literature survey to determine to what extent related work already provides an appropriate framework. We found that, while relevant related work exists, no existing work on the subject provides a classification framework that allows fine-grained classification based on definitions identifying the reasons for API misuses, namely API usage

[1] We focus on the issues originating from improper use of an API and not on social aspects such as developer skill and style.

constraints and API misuses themselves — thereby also associating classification types with the respective part of the method call.

We hence posed also the following research question:

RQ2 *What does an API classification framework need to look like that links API misuses to their causes and the respective part of the method call?*

The chapter overall is structured as follows. We first provide background on APIs and show an example of their (mis)use in Section 4.2. In Section 4.3, we provide an overview of previous works that have declared diverse definitions related to API misuses, elaborated on the root causes, and focused on classifying the different misuse types. As a result, in Section 4.4, we introduce our own definitions with respect to previous research, originating from the results of **RQ1**. Based on this theoretical foundation, we then address **RQ2** by contributing ***FUM***, a comprehensive "**F**ramework for API **U**sage constraint and **M**isuse classification" (cf. Section 4.5). We further clarify this in Section 4.6 by discussing the differences between *FUM* and other classification frameworks. Section 4.7 then presents a case study that applies *FUM* to the cryptographic API misuse detector *CogniCrypt*, which is considered state of the art in API misuse detection, has been shown to cover a majority of cryptographic API misuses, and has better precision than other cryptographic API misuse detectors [Gao+19; HGN20; Bon+21]. The study shows that *FUM* assists one in classifying API usage constraints and API misuses, and in improving API misuse detectors. Section 4.8 concludes.

4.2 Background

In modern software development, the reuse of functionality is a desired approach because the developer does not have to implement it entirely from scratch. In object-oriented languages such as Java, the functionality is usually implemented in multiple classes bundled and offered to the developer as a library. Classically, a library can be divided into two areas: the public interface and the private implementation [Mon+12]. The private implementation is the part that implements the functionality of the library. However, concrete implementation details are usually not exposed to the developer (i.e., to the library's user). In the following, we will not consider the private implementation but the public interface, as we focus on its misuses. The public interface exposes software elements (e.g., classes and methods) to the outside world, making the implemented functionality accessible. In the literature,

these public interfaces are known as application programming interfaces (APIs). Furthermore, we refer to an *API class* as a class from the API and an *API object* as its instantiation.

At first glance, the widespread use of an API (and its instantiated objects) seems simple because in most cases, only method calls are necessary to get the desired functionality [ZM19]. However, it can be much more complicated. For instance, developers must consider certain restrictions in environments (e.g., multi-threading) or be aware of specific properties (e.g., when performance plays an important role).

Consider, for instance, a correct application of the `java.io.FileReader` [Orac]. Whenever a file is successfully opened, the file's content can be `read()`, and following its usage, the resource needs to be released by invoking `close()`. Consequently, it is a misuse not to call `close()` at the end of every usage of `FileReader`.

To support developers, APIs are delivered with documentation, which can be very diverse in nature [Mon+12]. For example, parts of the documentation can be related to typical use case scenarios, code snippets, constraints, or even performance.

We now have a better understanding of the definition of an API. However, we do not know how it relates to API misuse, nor do we know the root cause of API misuse or what concrete API misuse types exist; we address this in the following sections.

4.3 Related Work — A Survey of Definitions and Perspectives

This section addresses **RQ1**, providing an overview of several studies focused on different aspects of API documentation, misuses, and even beyond. We performed a systematic literature review [Kit04]. As part of our literature survey, we used keywords "API misuses", "misuse detection", "library misuse" among other related terms in Google Scholar and IEEE Xplore from November 2020 until February 2021. To complete the results of the literature research process, we have snowballed relevant results backward and forward. Overall, we considered 69 publications in discussion with two or more co-authors to filter out outcomes. Furthermore, we also actively engaged in discussions with the authors of previous studies. The most relevant publications are presented in this chapter.

Whenever an API is used, developers need comprehensive and explanatory documentation. Past research focused on the different kinds of API documentation. We consider previous work by its range of different (implicit) types of API documentation and its noncompliance. For example, Robillard and Maalej [MR13] focused on every part of API documentation, whereas Lv et al. [Lv+20] only considered those parts of documentation for which noncompliance would result in errors or misbehavior. Further, we also show that almost every study provides its own definitions (denoted in bold letters).

4.3.1 API Documentation Types and Definitions

Dekel and Herbsleb [DH09] researched how developers' awareness of *usage directives* could be increased. They defined **usage directives** as parts of API documentation that capture nontrivial, infrequent, and possibly unexpected information, introducing ten different types of usage directives, e.g., *Restrictions*, which address the context in which the API method should (not) be called. They developed *eMoose*, an Eclipse [Ecla] plugin to highlight *usage directives* in the IDE.

Bruch et al. [BMM10] investigated parts of API documentation focusing on extensibility. They defined parts of API documentation related to extensibility and inheritance as **subclassing directives**, e.g., the subclassing directive *subclasses may extend this method* requires subclasses to call the super method. In total, they introduced four types of subclassing directives.

Based on both studies, Monperrus et al. [Mon+12] derived a classification framework of API directive types. They defined **API directives** as natural-language statements of API documentation that describe how to use an API correctly and optimally. Their classification framework comprises 26 different API directive types. Moreover, they performed an empirical study on the variety of API directive types in the wild, based on the documentation of three Java libraries, namely the *Java Class Library* [Orag], *JFace* [Eclb], and the *Apache Commons Collections* [Apaa]. Analyzing 4,561 API elements (i.e., documentation of interfaces, packages, classes, etc.) in total, they showed the prevalence of each type, respectively.

Robillard and Maalej [MR13] empirically elaborated a comprehensive taxonomy of **API knowledge types**, which are patterns of knowledge classifying a specific part of API documentation, e.g., the type *Directive* specifies what developers are (not) allowed to do with the API. Further, the *Quality* type describes non-functional requirements like performance implications. In total, they introduced twelve of these types.

4.3.2 API Usage Constraint Types and Definitions

In this section, we present studies on API documentation subsets that a developer must comply with to avoid encountering misbehavior, errors, or vulnerabilities.

Li et al. [Li+18] narrowed down the set of API documentation parts to **API caveats**, which are directives where noncompliance would likely incur unexpected program behaviors or errors. Moreover, they introduced a classification framework of syntactic patterns that comprises ten types of API caveats. Lv et al. [Lv+20] considered only the parts of API documentation for which noncompliance would result in severe consequences, so-called **integration assumptions (IAs)**. Each IA has its constraint related to pre-conditions (e.g., parameter length limit), post-conditions, or the invocation context. Similarly, Nguyen et al. [NVN19] classified constraint types that lead to serious programming errors but without providing a concrete definition. They categorized such constraints as temporal order (e.g., API method calls are expected in a particular order), pre-conditions and post-conditions, argument value, and finally, exception (e.g., handling of certain exceptions). They proposed a novel approach called *Statistical Approach for API Misuses* (SAM) that tries to detect noncompliance.

Saied et al. [SSD15] investigated so-called **API usage constraints** and found that API usage constraints are (and should be) captured in the respective API documentation. They introduced four types, namely *Nullness not allowed* (i.e., passing a *null* value results in failures), *Nullness allowed* (i.e., passing a *null* value has a certain semantics), *Range limitation* (i.e., restricted numeric value), and *Type restriction* (i.e., restricted parameter type). Except for *Nullness allowed*, all introduced types restrict the API usage, and noncompliance would result in runtime failures. Saied et al. showed that such API usage constraints are frequently present in code but not always documented.

Addressing this problem, Amann [Ama18] defined **API usage constraints** to be (implicit) constraints not enforced by the compiler, such as correct typing, but constraints enforced by the API, for which noncompliance would result in runtime errors. Based on the comprehensive classification framework of API directives of Monperrus et al. [Mon+12], they provided an overview of API usage constraint categories.

In contrast, Ren et al.'s definition also comprises parts of the API documentation in which noncompliance does not necessarily result in errors or misbehavior. They [Ren+20] defined **API usage directives** as "[...] contracts, constraints, and guidelines that specify what developers are allowed/not allowed to do with the API".

In addition, several (mostly empirical) studies [SHA15; Cer+18; BBA09; BKA11; Bod09] investigate only certain constraints, mostly referred to as **object protocols**, which according to Beckman et al. [BKA11] are "[...] dictating the ordering of method calls on objects of a particular class". An example of a predefined method call order is the `java.io.FileReader` [Orac] where `read()` is only allowed to be called if the resource was opened successfully.

4.3.3 API Misuse Types and Definitions

So far, we have only considered studies investigating the spectrum of API directives and API usage constraints. As shown in Figure 4.1, API usage constraints are a subset of API directives for which violation results in misbehavior, errors, or vulnerabilities at runtime (cf. Section 4.3.2). The dashed line splits API directives and API usage constraints as they can either be implicit or documented. Since the designer of an API is not given language constructs provided by the programming language to force the API user to comply with API usage constraints [Ama18], they can be (unintentionally) violated. Such a violation is mostly referred to as **API misuse** in the literature. We will introduce precise definitions in Section 4.4.

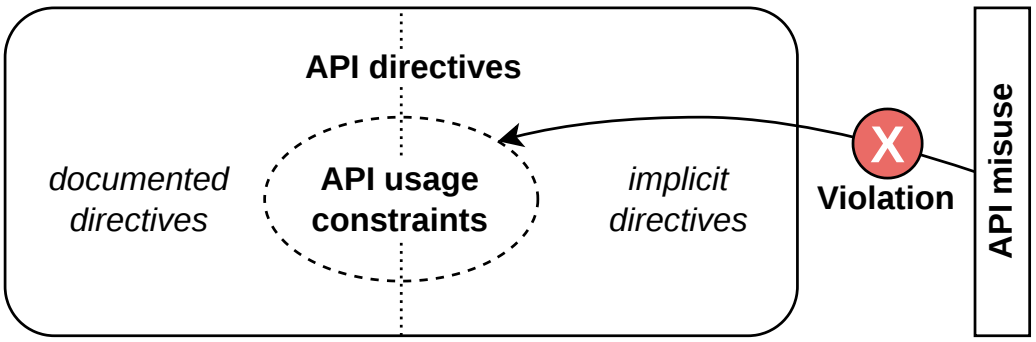


Figure 4.1: Summary of different terminologies around the term API misuses we observed in the literature.

The classification, detection, and mitigation of API misuses have been studied in various specific fields of research – for instance, cryptography [Nad+16; Laz+14; Krü20; Ege+13], machine learning [Isl20], stream APIs [Kha+20], and even biometric APIs [JCX19], as well as in the overall analysis of API misuses [Ama18; Li20; Sca+19; KS14; Kec+19; HP04]. Further, API misuses are not specific to just one programming language like Java. There are also studies focusing on API misuses

in different languages, e.g., C [Gu+19] or Python [Isl20]. Thus, API misuse is a more general problem for which we observed different kinds of definitions in the literature.

Luo et al. [Luo+18] investigated the continuous and rapid development of the *Android Platform APIs* [Goo]. They found that Android applications (apps) often cannot keep up with the rapid development of the underlying *Android Platform APIs*. Therefore, apps can run into compatibility, security, or reliability problems. They define an **API misuse** as the violation of an API's contract, resulting from an update to the underlying API, but improperly maintained API calls.

Khatchadourian et al. [Kha+20] studied how Stream APIs are misused. They defined bugs related to Stream APIs as **Stream Misuses**, where the term *bug* is not further specified. From 22 projects using streams, they identified different classifications of Stream Misuses by analyzing Git [Gita] commits. They derived a classification framework comprising 15 different categories.

Especially in the domain of cryptography, researchers are interested in API misuses [Nad+16; Krü20; Ege+13; Laz+14; Cha+16]. Nadi et al. [Nad+16] found that cryptographic APIs' low level of abstraction leads to incorrect usage. As a result, APIs become too complex to use to achieve a specific goal. For example, there is the need to compose several method calls over different API objects to encrypt a file correctly and securely. Egele et al. [Ege+13] conducted an empirical study on Android apps. They found that 88% of all apps have at least one mistake using cryptographic APIs. They showed that an API misuse does not necessarily result in errors or crashes but in sensitive vulnerabilities causing possible security breaches. However, they did not provide any precise definition of an API misuse.

According to a study by Amann [Ama18] in 2018, they were the first to systematically define the problem space of API misuses and empirically investigate the prevalence of API misuses in bug datasets. They introduced a comprehensive classification framework of 16 different API misuse categories: *MuC* [Ama+18]. They defined an **API misuse** as a violation of an (implicit) API usage constraint (cf. Section 4.3.2). Besides summarizing the findings of Monperrus et al. [Mon+12], Amann based their framework on the results from investigating about 1,200 bug reports from bug datasets like *BugClassify* [HJZ13] or *Defect4J* [JJE14] and Monperrus et al.'s work. They identified a total of 164 API misuses. In addition, they provided their own dataset with pre-classified API misuses based on *MuC*, the so-called *MuBench Dataset* [MuB]. Together with *MuBench* [Ama+16] — a benchmark to assess API misuse detectors — they provide a versatile toolbox in the field of API misuse and detection.

Since *MuBench* is based on API misuses from bug datasets, it might be limited. Thus, Li [Li20] conducted the first large-scale empirical study of API misuses in 2020. They defined "[...] **API misuses** as code edit operations related to some APIs in a bug fix". Although their definition is in stark contrast to Amann's, they followed up on *MuC*, refined the classification framework by restructuring and introducing new categories. Moreover, they mined all bug-fixing commits of Java projects from GitHub [Gitc] between 2011 and 2018. From over three million investigated changes (3,197,593) in the respective commits, Li found 50.6% were involved in API misuses, demonstrating that developers heavily struggle with API misuses in modern software development.

4.4 Definitions

As discussed, several studies provide fundamental theory. They already present comprehensive classification frameworks of the different types of usage constraints and misuses [Ama18; Mon+12; Li20; BKA11]. However, we observed that the definitions surrounding the term API misuse are scattered across several studies in different versions, sometimes targeting different aspects of an API misuse. In fact, researchers classify API misuse types by *(i)* the overall definition of an API misuse in the specific domain and *(ii)* by the needs of their work. This section provides two reasons why we study the term API misuse and its root causes again.

Reason 1 — Dispersed Theory Knowledge across Studies

Scattered across several studies, different approaches have already been introduced to classify usage constraints. However, some work focuses on specific areas, such as object protocols [SHA15; Cer+18; BBA09; BKA11; BHL07], while others try to classify — sometimes roughly — the spectrum of API directives, usage constraints, or API misuses [Mon+12; Ama18; Li20; NVN19; SSD15; DH09; SHA15; ZM19; BBA09; Luo+18; Kha+20; Lv+20; Li+18; MR13; Sca+19; KS14; Kec+19; HP04; Rob+13] (cf. Section 4.3). The most comprehensive summary of relevant misuse types so far is provided by *MuC* [Ama18]. However, the high abstraction level they chose misses several important pieces of information. For instance, Monperrus et al. [Mon+12] mentioned the *Method Parameter Type Directive*, which mandates the allowed type of method parameters. For example, the `compareTo(Date)` method of `java.sql.Timestamp` [Oram] states: "[...] Compares this `Timestamp` object to the given `Date`, which must be a `Timestamp` object." Although mentioned by Amann,

this type is not part of their API misuse derivation. Our approach ensures such root causes of API misuses are not missed. Essentially, we define all necessary terms surrounding API misuses (**RQ1**) based on our survey. This allows us to collect previous contributions to provide one comprehensive framework (**RQ2**).

Reason 2 — Localization of Usage Constraints in Method Calls

We show that usage constraints can be associated with specific parts of a method call. For instance, usage constraints of type *Post-Call Directive* [Mon+12] categorize requirements for method calls that must later be invoked on the returned object. Thus, this usage constraint type only targets the return value of a method call. We can also observe usage constraint types associated with the invoked method itself, the passed arguments, and the context in which the API is used. Such localization helps to better understand the causes and types of API misuses. There is no previous taxonomy of API usage constraint types based on their location in API method calls.

4.4.1 Definition of "API."

We provided a definition for APIs in natural language based on Monperrus et al.'s work [Mon+12] (cf. Section 4.2), where we can basically divide APIs into two areas in terms of their application domain: *(i)* APIs with a focus on a particular domain. For example, cryptographic APIs implement cryptographic algorithms and provide basic functionality for data security. *(ii)* Other APIs are domain-independent. Consider, for instance, `java.lang.String` [Oral]. It provides overall functionality to work with strings that are used domain-independently. Note that this differentiation is important because oftentimes domain-specific APIs have more usage constraints and disclose very specific usage constraint types, which we will see for the case study (cf. Section 4.7) for the domain of cryptography. This perspective on APIs leads to the following definition:

Definition 4.4.1. *A **domain-specific API** offers functionality tailored to a specific domain. Its domain determines the achievable goal and application rules of a domain-specific API. In contrast, **non-specific APIs** are not tailored to a domain, nor do they determine a specific goal associated with their use.*

4.4.2 Definition of "API Directive."

As previous studies have shown [Nad+16], developers struggle to understand the intended usage of an API. Therefore, the API documentation is a core requirement that provides code examples, conceptual overviews, definitions of terms, programming guidelines, known bugs, and documented workarounds [Kra99]. In fact, API documentation is manifold in its nature [MR13; Mon+12]. Essential for developers are contracts enforced by the API. Such contracts, also called *API directives*, describe what developers are (not) allowed to do [MR13]. Monperrus et al. [Mon+12] described them as "[...] natural-language statements that make developers aware of constraints and guidelines related to the usage of an API". They consider API directives to be captured in the respective API documentation. However, documentation does not necessarily need to be of high quality, and thus it may be incomplete or contradictory [Rob+13; NVN19; SSD15]. It is not uncommon for documentation to be completely unavailable. Furthermore, some APIs take domain knowledge for granted [Nad+16] and miss documenting such domain-specific constraints. Therefore, we extend the definition of API directives not only to the underlying API documentation but also to *implicit* statements. This conceptual expansion to Monperrus et al.'s [Mon+12] work leads to the following definition:

Definition 4.4.2. *An **API directive** is a natural-language statement related to guidelines or constraints that describes how to use an API correctly and optimally. It can be part of the underlying documentation of an API. However, an API directive can also be implicit, for example, because of incomplete documentation or expected domain-specific knowledge.*

4.4.3 Definition of "API Usage Constraint."

API directives can also be guidelines (e.g., hints for performance improvements). Hence, we can further distinguish between different types of API usage constraints. A constraint is a contract that narrows down the actual use of an API. For example, an API may require a method to be called only under certain conditions. In the case of `java.util.Iterator` [Oraf], the method `next()` may only be called if the iterator contains at least one element [BHL07]. At compile time, an API designer cannot force a developer to adhere to these constraints because the programming language does not provide any language constructs to ensure compliance (e.g., correct typing enforced by the compiler [Ama18]). Overall, this implies the following definition of an API usage constraint based on the definition of Amann [Ama18]:

Definition 4.4.3. *An **API usage constraint** is an API directive that restricts the actual use of an API. These restrictions are not enforced by the programming language itself, such as correct typing. Because API usage constraints are API directives, they are imposed by the API designer/expert.*

Note that an API usage constraint is tailored to the perspective of an API designer/expert. They are responsible for the API design, which includes imposing the constraints on it. This perspective is important as API usage constraints are API directives and therefore not imposed by any API user.

4.4.4 Definition of "API Misuse."

API usage constraints can be violated, e.g., because of a developer's lack of domain knowledge [Nad+16] or improper documentation [Rob+13]. This harms the program's further execution as it leads to misbehavior of the API, errors, crashes, or even worse, security vulnerabilities. Therefore, we extend the definition of Amann [Ama18] to the following:

Definition 4.4.4. *An **API misuse** is the violation of one or more API usage constraints. Such violation leads to misbehavior of the API, e.g., errors, crashes, or vulnerabilities.*

An example of an API misuse is not releasing a recently opened resource by calling `close()` after the end of using `java.io.FileReader` [Orac] (cf. Section 4.2).

In conclusion, the definitions presented in this section have answered the first research question (**RQ1**) by discussing the relevant aspects surrounding API misuse and deriving definitions concerning previously conducted studies. The provided definitions narrow down the term API misuse, as it is a violation of an API usage constraint. An API usage constraint is imposed by an API designer/expert and does not necessarily need to be captured in the respective documentation. Thus, API usage constraints may also be part of domain-specific knowledge that is taken for granted by the API (designer).

4.5 Classification of API Usage Constraints — *FUM*

We next provide ***FUM***, a comprehensive "**F**ramework for API **U**sage constraint and **M**isuse classification". To accurately assess and understand the causes and types of

API misuses, one needs to elaborate on the different types of API usage constraints. We consider an API usage constraint to be a concrete constraint imposed by the API. Accordingly, an **API usage constraint type** (*FUM* type) is a set of API usage constraints that share the same kind of constraints.

Following our definitions (cf. Definitions 4.4.3 and 4.4.4), the only cause of an API misuse is a violation of the API usage constraint(s). Therefore, *FUM* is based on API usage constraint types instead of API misuse types like other approaches [Ama18; Li20]. Moreover, we localize each API usage constraint type by the parts of an API method call (i.e., the API usage).

Usage Constraint Types by Parts of an API Method Call

As our study revealed that Monperrus et al. [Mon+12] so far introduced the most comprehensive and fine-grained collection of API directive kinds and an empirical study on their prevalence in API documentation, we extend their contribution. We consider all their mentioned API directive kinds, which actually are API usage constraint types (cf. Definition 4.4.3). We then enrich their findings by contributions of previous other studies [Ama18; Li20; BKA11; NVN19; Rob+13; Cer+18; SSD15; ZM19; DH09; SHA15; BBA09; Luo+18; Kha+20]. As a result, we introduce six new or more fine-grained types to the API usage constraint types, i.e., *High-Level Constraints*, *Post-Null-Check*, *Controlling Method Call*, *Threading*, *Argument State*, and finally, *Pre-Null-Check* (shown in italics in Figure 4.2).

Since interaction with APIs is due to method invocations [ZM19], we assign each API usage constraint type to the different parts of an API method call. We provide an overview of API usage constraint types and their localization in Figure 4.2. Furthermore, we consider an API method call to be a method invocation either on the API class or on the respective instantiated object (i.e., the API object), which allows us to clearly separate and localize API usage constraints based on the use of APIs. We consider an API method call to have three relevant parts: the *return value*, the *method call* itself, and the *passed arguments*. There are also API usage constraint types associated with multiple parts of the API method call (uncolored dashed boxes).

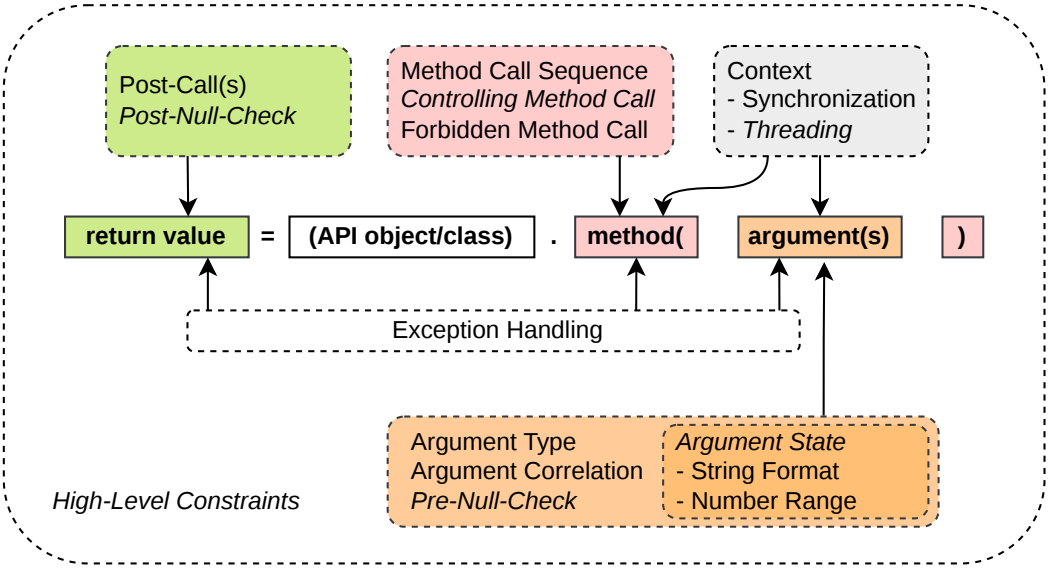


Figure 4.2: *FUM* — Overview of API usage constraint types associated with parts of an API method call. Types shown in italics are additionally added to the work of Monperrus et al. [Mon+12]. Dashed colored boxes are specific to a single API method call part. Uncolored dashed boxes are API usage constraint types spanning over multiple parts of an API method call.

4.5.1 *FUM* Types Associated With the "Return Value."

Here, we discuss the usage constraint types associated with the return value of an API method; we call this main type *return value*. An API can impose constraints even beyond the method call itself (e.g., requirements in the form of methods that need to be called on the return value). We can observe two types of API usage constraints associated with the *return value* (cf. green boxes in Figure 4.2).

Post-Call(s) constraints mandate calling methods on the *return value* [Mon+12; NVN19; Lv+20]. For instance, the static method `AlgorithmParameters.getInstance(algorithm)` [Oraa] requires initializing the *return value* via the call of `init()` [Mon+12]. Note that this API usage constraint type is a specialization of *Method Call Sequence*, but with the fact that the required calls do not necessarily refer to the same API class. Rather, the *return value* may be typed differently. In total, 0.9% of all analyzed API elements contain such usage constraints [Mon+12].

Post-Null-Check comprises (implicit) usage constraints that require a *null* check on a *return value* to avoid runtime errors [Ama18; Li20; NVN19]. We can observe such constraints whenever a method returns either a specified value or `null`. For instance, the API class `java.io.InputStreamReader` [Orae] exposes the method `getEncoding()` which returns the name of the character encoding being used by the stream or `null` if the stream is closed. However, since the return value is a string,

the developer may continue to work on this return value; there is a need to check for `null` beforehand. This usage constraint type is not mentioned by Monperrus et al. [Mon+12]. Note that this kind of usage constraint is distinguishable from *Post-Call(s)* because no method call is used to check for `null`.

4.5.2 *FUM* Types Associated With the "Method Call."

We introduce all usage constraint types associated with the method call itself (cf. red boxes in Figure 4.2) as *method call*. Some APIs restrict interactions with themselves, or methods need to be called in a specific sequence to achieve the desired result. There are also cases where methods are not supposed to be called, although they are public.

Overall, we can observe mainly two reasons for constraints on the invocation of a method. Firstly, the context in which an API is used requires or restricts specific usage (cf. Section 4.5.4). Secondly, the state of the API object determines the allowed, required, or forbidden usage of itself [Mon+12].

Method Call Sequence constraints are usage constraints enforcing the requirement to call methods in a particular sequence [Mon+12; Cer+18]. A method call sequence can be considered as a usage protocol [BKA11; DH09; SHA15; Rob+13; Bod09]. Thus, the correct usage of method calls can be modeled with a finite-state machine [BBA09; Rob+13; ZM19]. Furthermore, the term typestate is also often used in the literature, i.e., the typestate defines the permitted operations on the API object based on its state [Krü20]. In total, up to 12.2% of all analyzed API elements contain such usage constraints [Mon+12].

Controlling Method Call classifies constraints on a method call that is only allowed to be called if a condition is met beforehand [Ama18; BKA11; NVN19; BBA09; Kha+20]. Unlike *Method Call Sequence*, this usage constraint type aims to use method calls in conditional statements. Thus, this type of *method call* controls and safeguards the further program flow depending on its return value. The return value can give information about the state of the API object and information on whether a method, including its argument constellation, is allowed to be called. For instance, consider the `java.util.Iterator` [Oraf]. To retrieve the `next()` element, there is a need to check whether the `Iterator` contains at least one element — to check the respective state of the API object that allows the call of `next()`. Only if calling `hasNext()` returns `true` are we allowed to use `next()`. Such controlling method calls can be seen as an interface for the developer to react to a specific state. In fact, this is an extension of *Method Call Sequence*, as it can be seen as a sequence of method calls; however, further operations on the API object are dependent on

the return value. Note that this usage constraint is not explicitly mentioned by Monperrus et al. [Mon+12].

Forbidden Method Call specifies that certain methods are not allowed to be called [Mon+12; Ama18] and mostly is a special case of *Method Call Sequence*. There exist API methods that may not be called in an API object's whole lifetime, e.g., deprecated methods which are still callable for backward compatibility [Luo+18]. Consider, for example, the static class `java.net.URLDecoder` [Oran] where the method `decode(String)` is deprecated; instead, an encoding scheme should be passed as the second parameter of `decode(String, String)`. Note that the most similar type of Monperrus et al. [Mon+12] is *Method Call Visibility Directive* addressing constraints on the calling context's method call. Hence, we can only estimate the prevalence of *Forbidden Method Call* constraints with 4.2% [Mon+12] or less.

4.5.3 *FUM* Types Associated With "Passed Arguments."

We introduce usage constraint types related to passed arguments of an API method call (orange-colored box in Figure 4.2) as *passed arguments*.

Argument State constraints require the passed argument objects to be in a specific state or to have been correctly generated through a predefined protocol. Such predefined protocols specifying the composition of several API objects are also called *multi-object protocols* [Rob+13; SHA15]. To analyze such protocols, which can span multiple objects, Bodden et al. [BHL07] used *tracematches* to specify *traces* expressing a protocol that are matched against the source code. We can typically observe this type of usage constraint in the context of cryptography, where several API objects need to be correctly composed [Krü20]. Besides, we can observe special cases not perfectly fitting into this definition, e.g., if constraints are enforced on primitive types and string literals. We therefore further distinguish between two special cases, i.e., *String Format* and *Number Range*. Since we tightly refer to Monperrus et al., we only distinguish between those. In fact, those special cases can be extended to other primitive types as well.

String Format constraints specify a restriction for the format of the passed string [Mon+12]. There are APIs that only allow a certain value from a set of strings. For instance, the method `getBytes(charsetName)` provided by `java.lang.String` [Oral] needs the argument to be a valid string from a set of supported charsets (e.g., UTF-8). In total, 2.7% of all analyzed API elements contain such usage constraints [Mon+12].

Number Range constraints specify that only a set or range of numbers are allowed to be passed [Mon+12; SSD15]. In total, 2.7% of all analyzed API elements contain

such usage constraints [Mon+12].

Pre-Null-Check constraints state the requirement to check for `null` before the argument is passed to the method [Mon+12; Ama18; Li20; NVN19; SSD15] and are rather implicit like *Post-Null-Check* (cf. Section 4.5.1). There are cases where the documentation explicitly restricts passing `null`, e.g., the API class `Collating Iterator` [Apac] exposes the method `addIterator(iterator)`, for which the documentation states "[...] the iterator to add the collection must not be null". In total, 13% of all analyzed API elements have at least one such usage constraint [Mon+12].

Method Parameter Type constraints restrict the passed argument's type [Mon+12; SSD15]. Although this is contrary to object-oriented programming principles (i.e., generalization [Tai96]), there are cases where this particular usage constraint exists [Mon+12]. For instance, the `java.sql.Timestamp` [Oram] API class offers the method `compareTo(date)` where the parameter `date` is of type `java.util.Date` [Orab]. However, the documentation states "*Compares this Timestamp object to the given Date, which must be a Timestamp object*" [Mon+12]. In total, 1.9% of all analyzed API elements contain such a usage constraint [Mon+12].

Method Parameter Correlation restricts the value of passed arguments because of their inter-dependency [Mon+12]. For instance, the method `setKeyEntry(alias, key, password, chain)` provided by `java.security.KeyStore` [Oraj] requires passing a `key` depending on the provided certificate `chain`. In total, 1.9% of all analyzed API elements contain such usage constraints [Mon+12].

4.5.4 *FUM* Types With Multiple API Method Call Associations

In addition to API usage constraint types only assigned to one specific part of an API method call, there are also usage constraint types assigned to multiple parts; we furthermore introduce *Exception Handling*, *Context*, and *High-Level Constraints* (uncolored and light-gray dashed boxes in Figure 4.2). Even though this category sounds similar to the *multi-object variant of the sequential pattern* discussed by Robillard et al. [Rob+13], their perspective is from that of an API misuse detector, and ours is from the API usage constraints themselves. Furthermore, our classification here involves more fine-grained information like exceptions, context, threading, etc. The usage constraint type *Context* is further divided into two subtypes, i.e., *Synchronization* and *Threading*.

Exception Handling constraints impose requirements at the exception-handling level. *Exception Handling* constraints describe the situations in which possibly thrown exceptions must be considered and reacted to precisely [Ama18; Li20; Kha+20]. This type of usage constraint can be associated with all parts of an API usage (cf.

white-colored box labeled with *Exception Handling* in Figure 4.2). In total, 4.1% of all analyzed API elements contain such a usage constraint [Mon+12].

Context constraints are related to language constructs that must surround the API usage, excluding constructs that are already covered by *Exception Handling* and *Controlling Method Call*. We observed a similar usage constraint type mentioned in previous studies [Mon+12; Ama18; DH09], but without explicitly limiting the constraint to surrounding language constructs. The `java.lang.Object` [Orak] is a good example that requires using its exposed method `wait()` only within loops. The most similar API directive mentioned by Monperrus et al. [Mon+12] is again the *Method Call Visibility Directive* we already introduced in the *Forbidden Method Call* constraint type (cf. Section 4.5.2). Similarly, we can only estimate the prevalence as 4.2% [Mon+12] or less. We further subdivide *Context* into two more usage constraint types, namely, *Threading* and *Synchronization* (cf. gray-colored box in Figure 4.2).

Threading addresses usage constraints of using an API only on specific threads. Consider, e.g., the `javax.swing` [Orai] API package, where the documentation states: "[all] `Swing` components and related classes[...] must be accessed on the event dispatching thread" [Orai]. The most similar API directive mentioned by Monperrus et al. [Mon+12] is again the *Method Call Visibility Directive* as for *Forbidden Method Call* and *Context* with an estimated accumulated prevalence of 4.2% [Mon+12] or less.

Synchronization comprises requirements taking care of concurrent access to the API object in a multi-threaded environment (i.e., thread-safety) [Ama18; Mon+12; Kha+20]. For instance, if at least one thread structurally modifies an instantiation of `java.util.HashMap` [Orad], there is a need to synchronize the API object externally (e.g., by retrieving a lock). This type also includes requirements for proper assurance of thread-safe use, e.g., a usage constraint of this type is violated if the lock is obtained twice in a nested manner (deadlock). In total, 3.9% of all analyzed API elements contain such a usage constraint [Mon+12].

High-Level Constraints address usage constraints that cannot be covered by the previously presented types in this section. They can cover all areas of an API usage (cf. large dashed box in Figure 4.2). For example, the operating system on which the program is executed may be important — Windows machines provide different algorithms for the secure generation of random numbers than Linux machines [Krü20]. Therefore, this may also affect the interaction with the given API that encapsulates and provides such functionality. Note that none of the studies we considered explicitly mentioned this type of usage constraint, nor did Monperrus et al. [Mon+12].

4.5.5 *FUM*— Limitations

The work of Monperrus et al. [Mon+12] and Bruch et al. [BMM10] also considered types of usage constraints related to the extension and inheritance of APIs. Monperrus et al. list these usage constraint types under the main category *Subclassing Directive*. For instance, there is the type *Call Contract Subclassing Directive*, which states that whenever a defined method is overridden, there is a need to call other predefined methods rather than just calling the `super()` method. However, *FUM* (cf. Figure 4.2 in Section 4.5) only comprises usage constraint types that restrict an API's usage (i.e., not an extension or inheritance) for simplicity. Furthermore, our survey and therefore *FUM* are based on the Java language (cf. Section 4.1).

4.5.6 Overlapping Characteristics of *FUM* Types

Although we provide a more fine-grained classification framework than recent studies, in addition to an association of usage constraint types related to the actual use of an API, there are cases where the types are not orthogonal to each other. For example, this is the case when we consider *Pre-Null-Check* (cf. Section 4.5.3) and *Exception Handling* (cf. Section 4.5.4) constraints. An API can indicate that a `NullPointerException` is thrown when *null* is passed to a method call. This can be seen as a usage constraint of *Pre-Null-Check* (i.e., checking the argument for not being *null* before it gets passed to the method call). Conversely, it can also be seen as a usage constraint of *Exception Handling* because if it is not checked in advance, the developer must consider the possible thrown exception.

4.6 Discussing Differences of Classifications

The most similar classification framework compared to *FUM* is *MuC* by Amann [Ama18] and its refined version provided by Li [Li20] (cf. Section 4.3.3). Both studies introduced an exhaustive classification framework of API misuse types and indicated the prevalence of each elaborated type. Amann's study is based in part on the findings of Monperrus et al. [Mon+12], who examined the different types of API directives, including API usage constraints. Since Li used the work of Amann as a foundation, they also implicitly used the findings from Monperrus et al. as we did for *FUM*. Although *FUM* works on the level of API usage constraint types, we can compare *FUM* to theirs, as a violation of an API usage constraint is an API misuse (cf. Definition 4.4.4); hence, *FUM* also can be used to classify API misuses analogously.

In this section, we show that *FUM* is at least as expressive as theirs. Moreover, *FUM* provides a more fine-grained classification of API usage constraint types — and API misuse types. For the sake of clarity, we annotate Amann's introduced types with 'A', Li's with 'L', and ours with '*FUM*', respectively.

API-Misuse Classification (MuC) [Ama18] *MuC* encompasses four main categories: Method Calls, Condition, Exception Handling, and Iteration. Although every main category has its subcategories, we only introduce subcategories' details when (i) there are clear differences between the violation of our introduced *FUM* types or (ii) this subcategory does not refer to API misuse based on our definitions.

The category **Method Calls**$^{\mathbf{A}}$ is covered by the violation of *Method Call Sequence*^{FUM}, as its constraints define the allowed and required method calls based on the API object's state.

Their category **Conditions**$^{\mathbf{A}}$ is covered by the violation of a combination of *FUM* subcategories, namely *Pre-Null-Check*^{FUM}, *Post-Null-Check*^{FUM}, constraints of type *Method Call Sequence*^{FUM}, and *Controlling Method Call*^{FUM}, as well as all the subtypes listed in *Passed Arguments*^{FUM} (excluding *Pre-Null-Check*^{FUM}) and *Argument Type*^{FUM}. Thus, our separation allows us to assess API misuses in a more fine-grained way. Further, *Redundant Value and State Condition*^{A} aims at the use of unnecessary conditional checks before using an API method. However, according to Definition 4.4.4, they are not considered API misuses.

The subcategory *Missing Synchronization Conditions*^{A} classifies API misuses that imply an improper use of the API object regarding synchronization — for example, in a multi-threaded environment where the API object is used without first obtaining a lock. This covers exactly violations of *Synchronization*^{FUM} usage constraints. In contrast, *Redundant Synchronization Conditions*^{A} classifies API misuses for two typical cases. First, situations in which a lock is obtained unnecessarily, and second, situations where a lock is obtained twice in a nested manner, leading immediately to a deadlock. According to our definition, the first case is no API misuse if and only if it does not negatively affect the program (cf. Definition 4.4.4). The second case is covered by the violation of *Synchronization*^{FUM} constraints.

Finally, their subcategory *Missing Context Conditions*^{A} aims at API misuses related to threading. For example, GUI components from Swing [Orai] need to be accessed on the event dispatching thread; if not, then it is accordingly an API misuse. These kinds of scenarios are covered by the violation of *Threading*^{FUM} constraints. Moreover, we provide a more fine-grained view as our *FUM* type *Context*^{FUM} also comprises

constraints related to language constructs that need to surround the API usage. The other subcategory, *Redundant Context Conditions*^{A}, classifies API misuses related to threading whose usages are merely redundant and therefore not needed, which is covered in *FUM* as the violation of *Threading*^{FUM}.

The category **Iteration**$^{\mathbf{A}}$ is covered by the violation of *Controlling Method Call*^{FUM}, and finally, the **Exception Handling**$^{\mathbf{A}}$ categories are covered by the violation of either *Exception Handling*^{FUM} or *Method Call Sequence*^{FUM}.

Refinement of MuC by Li [Li20] Following the study by Amann [Ama18], Li [Li20] did a more exhaustive and wide-ranging empirical study on API misuses in the wild in 2020. They restructured and refined Amann's *MuC* and introduced new subtypes in addition to *Method Calls*^{A}, namely, *Replaced Arguments*^{L}, *Replaced Name*^{L}, *Replaced Name and Arguments*^{L}, and finally, *Replaced Receiver*^{L}. *FUM* covers them with *Method Calls*^{FUM} and *Passed Arguments*^{FUM}.

Summary *FUM* provides at least as much expressiveness as *MuC* [Ama18] and Li's [Li20] refined framework. Moreover, we provide a more natural view of API misuses by classifying usage constraint types instead of classifying API misuse types. We also provide a more fine-grained overview of usage constraints. *FUM* includes the (new) API usage constraint types *Method Parameter Type*^{FUM}, *Method Parameter Correlation*^{FUM}, and *High-Level Constraints*^{FUM}. In addition, *FUM* has split individual types to better locate them based on API method calls. For example, *Missing Null Check*^{A} is further distinguished into *Post-Null-Check*^{FUM} and *Pre-Null-Check*^{FUM}, which comes closer to the actual use of an API method call. As a result, this answers our second research question (**RQ2**) for a classification framework.

4.7 Case Study: *CogniCrypt*

To assess the extent to which *FUM* aids in determining and guiding the improvement of an API misuse detector's capabilities, we performed a case study consisting of two parts: First, we evaluate a domain-specific static analysis tool that, due to the domain-specificity, we expect to provide only partial coverage of *FUM* types. Second, we seek to assess to what extent *FUM* can then be used to systematically extend the coverage of the analysis tool to further *FUM* types.

As a study subject, we used *CogniCrypt* [Krü+19], a state-of-the-art static analysis tool for misuse detection of cryptographic APIs (cf. Section 2.2.2). *CogniCrypt* follows an allowlisting approach, consuming rules of correct usages of cryptographic APIs defined with the domain-specific language *CrySL* [Krü+19] (cf. Section 2.2.2). We sought to determine which API usage constraints (*FUM* types) *CogniCrypt* can detect, and based on the assessment of the coverage, we used *FUM* to identify weaknesses and to propose extensions to improve detection coverage. Specifically, we inspected which *FUM* types are specifiable with *CogniCrypt* [Krü+19] (version 1.0.0) and *CrySL* [Krü+19] (version 2.0.0).

FUM enabled us to classify the *FUM* types of code samples of API misuses we collected from the *MuBench Dataset* [Ama+16], as well as Li's [Li20] five most common API misuse patterns for each of their categories. In total, we were able to extract 88 usage constraint examples that are also API misuses based on Definition 4.4.4 (cf. Section 4.4.4). Our dataset consists of general API misuse cases as we assess a domain-specific tool's capabilities and then provide targeted improvements. The distribution of the examples for individual *FUM* types is shown in Table 4.1, and the protocols of our evaluation per code sample can be found in the artifact [Sch+22b] we provide.

We examined whether the usage constraints could be specified via *CrySL* rules. Upon inspection, only 27 of 88 samples were specifiable using *CrySL*. Due to bugs in *CrySL* and *CogniCrypt* we uncovered and reported [Sch+22b], fewer code samples were detectable with our rule specifications. Table 4.1 shows the simplified results where we only discuss whether code samples of API usage constraints were specifiable with *CrySL*. Overall, the coverage of *FUM* types (cf. column *Currently*) is limited. However, this was to be expected since *CrySL* and *CogniCrypt* are designed specifically for the domain of *cryptographic* APIs (cf. Definition 4.4.1: domain-specific API), and the scope of *FUM* reaches far beyond just this domain.

As the next step of our case study, we assume that it is a sensible goal to adapt the domain of *CrySL* and *CogniCrypt* to general API misuse detection. Whether such a broadened domain makes sense with regard to usability or performance is not the focus of this evaluation. To see to what extent one could mitigate the identified weaknesses of *CrySL* and *CogniCrypt*, we derive theoretical improvements (in the form of language extensions) to *CrySL* detailed in the provided artifact [Sch+22b]. We suggest adding two sections to *CrySL*, namely `EXCEPTIONS` and `GUARDS`, together with minor language improvements. These extensions are intended to express additional *FUM* types that cannot be properly specified in the current version of *CrySL*, in particular *Pre-Null-Check*, *Post-Null-Check*, *Exception Handling*, and *Controlling*

Method Call constraints theoretically. For instance, `EXCEPTIONS` would allow specifications to state how API-specific exceptions should be handled, while `GUARDS` would allow rules to encode conditions under which certain API calls are permitted or required. We focused on those *FUM* types that had the highest prevalence in our test samples but which were not already fully covered. However, our derived code samples did not contain a sufficient number of examples to derive a proper improvement suggestion for the remaining types of *Multiple Assignments* and *Post-Call(s)*. It seems plausible that, when implementing the suggested extensions, the number of specifiable samples would increase from 27 to 82 out of 88 (about 93% in total). We discussed our findings with the developers of *CrySL* and *CogniCrypt*, who will take them into consideration for future development. The case study revealed that *FUM* can be practically used to assess the current development state of API misuse detectors and that *FUM* can further be used to identify weaknesses and derive useful improvement suggestions.

Table 4.1: Simplified results of the evaluation of *CrySL*'s coverage of *FUM* types contrasted with proposed improvements. **Currently** shows *CrySL*'s coverage. **Improvements** shows the coverage once proposed improvements are applied. Dots symbolize the feasibility of specifying *FUM* types and respectively detecting their violations, i.e., API misuses. ○ denotes no API usage constraints are specifiable for this type, ● denotes all API usage constraints are specifiable. The values show the number of examples per *FUM* type.

Category	#	Constraint Type	Currently	Improvements
Return Value		Post-Call(s)	0 ○	0 ○
	16	Post-Null-Check	0 ○	16 ●
Passed Arguments	1	Argument State	1 ●	1 ●
	3	String Format	3 ●	3 ●
	1	Number Range	1 ●	1 ●
	0	Argument Correlation	0 ●	0 ●
	0	Argument Type	0 ●	0 ●
	5	Pre-Null-Check	0 ○	5 ●
Method Call	19	Method Call Sequence	19 ●	19 ●
	20	Controlling Method Call	0 ○	20 ●
	3	Forbidden Method Call	3 ●	3 ●
Multiple Assignments	14	Exception Handling	0 ○	14 ●
	1	Context	0 ○	0 ○
	4	Synchronization	0 ○	0 ○
	1	Threading	0 ○	0 ○
	0	High-Level Constraints	0 ○	0 ○
Summary	88		27 ●	82 ●

4.8 Conclusion

This chapter presented the second main contribution of this thesis, *FUM* (*Contribution 2*), which comprises three elements: *(i)* an overview of research on API misuses for the language Java, *(ii)* unified definitions around the term API misuse, and *(iii)* *FUM* — a framework for API usage constraint and misuse classification which is based on the localization of usage constraints on a method-call level.

While classification frameworks already exist specifically for evaluating the capabilities of API misuse detectors (e.g., *MuC* [Ama+18]), *FUM* focuses on an earlier level — API usage constraints instead of API misuses — because only the violations of API usage constraints are API misuses (cf. Definition 4.4.4).

We have shown that, compared to *MuC*, *FUM* has three advantages: (1) *FUM* is more fine-grained with new API usage constraint types, (2) *FUM* allows the localization of API usage constraints and API misuses as each type is associated with the different parts of an API usage (i.e., an API method call), and (3) *FUM* was designed to be comprehensible and exhaustive for API designers/experts providing a more intuitive view of API misuses.

Furthermore, a case study with the API misuse detector *CogniCrypt* [Krü+19] showed that *FUM* can be practically used to assess the current development state of such detectors and that *FUM* can effectively assist the identification of weaknesses and guide useful improvements. The contributions discussed in this chapter lay the foundations to create warning messages with increased explainability as described in Section 3.3.1. While this chapter investigated Java API misuses in general, the following Chapter 5 shifts the focus to cryptographic API misuses specifically. Section 5.1 investigates cryptographic API misuses in open-source business applications, discusses *effective false positives* (cf. Section 2.2.3), and provides a risk model that aids in addressing the usability categories (*Contribution 1*, cf. Chapter 3): *Warning Messages* (cf. Section 3.3.1) and *User Interface* (cf. Section 3.3.6). Section 5.2 introduces an adaptation of *CogniCrypt's* analysis to detect and reason about error chains caused by *multi-object protocols* (cf. Section 4.5.3), which also helps in addressing the usability category *Warning Messages* (cf. Section 3.3.1). Furthermore, in Section 5.3, *FUM* was used in the ground truth file of the domain-specific benchmark *CamBench* (*Contribution 3*, cf. Section 5.3) to describe the type of API usage constraint and misuse per file.

5 Cryptographic API Misuses

In the previous chapter, we investigated and defined Java API misuses and proposed a framework for their classification, *FUM* (cf. Chapter 4). In this chapter, we specifically explore the domain of *cryptographic* API misuses. A better understanding of cryptographic API misuses is important to achieve the goal of this thesis in improving developer support for reducing cryptographic API misuses, e.g., by increasing the explainability of *Warning Messages* (cf. Section 3.3.1).

This chapter combines three complementary foundations for building tools for cryptographic API misuse detection, which are finally applied in this thesis in Chapter 7 by building *SecAI* (*Contribution 5*). First, this chapter presents two co-authored publications that present empirical studies foundational for developing usability features in the final main contribution of this thesis, *SecAI* (*Contribution 5*). This chapter concludes in Section 5.3, introducing our third main contribution of this thesis, *CamBench* (*Contribution 3*), a "**C**ryptographic **A**PI **M**isuse detection tool **Bench**mark suite". Since this chapter combines multiple publications, they are discussed in separate sections.

Section 5.1 introduces a risk model that is used for the *Severity Score* feature (cf. Section 7.4.7) in *SecAI* and discusses findings of the empirical study of cryptographic API misuses with regard to *effective false positives*. Section 5.2 proposes an adaptation of *CogniCrypt's* analysis algorithm to detect error chains which can be explained by the concept of *multi-object protocols* (cf. Section 4.5.3) for which *SecAI* provides a visualization (cf. Section 7.4.3). Further, an expert interview with five experts indicates that this adaptation can help developers in the remediation of the identified error chains.

Lastly, Section 5.3 presents the third main contribution of this thesis, *CamBench* (*Contribution 3*), a benchmark that makes use of *FUM* (*Contribution 2*, cf. Chapter 4) for classification in its ground truth files and is used for the evaluation of SAST tools and for feature development in *SecAI*, including evaluation of the automatic cryptographic API misuse repair *LLM-SAST*$_{Repair}$ (*Contribution 4*, cf. Chapter 6) and for training our model for false positive prediction, *FP*$_{Predictor}$ (cf. Section 7.4.8). Both are integrated as features into *SecAI* (*Contribution 5*, cf. Chapter 7, Sections 7.4.5, 7.4.6 and 7.4.8).

5.1 To Fix or Not to Fix: A Critical Study of Cryptographic API Misuses in the Wild

This section presents a shortened version of the publication of Wickert et al. [Wic+22], which I contributed to during the research of this thesis. It aligns with the research of this thesis in understanding cryptographic API misuses and creating better tooling support for developers, e.g., by providing important information to developers in *Warning Messages* (cf. Section 3.3.1). In this section, we focus on two important elements of this empirical study of cryptographic API misuses in open-source business applications; first, the proposed risk model which is later used in the *Severity Score* feature (cf. Section 7.4.7) of *SecAI* (*Contribution 5*, cf. Chapter 7) and second, further insights into *effective false positives* which we aim to reduce by proposing *SecAI* in Chapter 7.

Prior studies have revealed that 87% to 95% of the Android apps using cryptographic APIs have a misuse that may cause security vulnerabilities [Cha+16; Krü+19]. As previous studies did not conduct a qualitative examination of the validity and severity of the findings, our objective was to understand the findings in more depth. We analyzed a set of 936 open-source Java applications for cryptographic misuses. Our study reveals that 88.10% of the analyzed applications fail to use cryptographic APIs securely. Through our manual analysis of a random sample, we gained new insights into *effective false positives*. For example, every fourth misuse of the frequently misused JCA class `MessageDigest` is an *effective false positive* due to its occurrence in a non-security context — technically, the static analysis is correct in reporting the warning; however, a developer might not address it. As we wanted to gain deeper insights into the security implications of these misuses, we created an extensive risk model of vulnerabilities for cryptographic API misuses. Our model includes previously undiscussed attacks in the context of cryptographic APIs such as DoS attacks. This model reveals that nearly half of the misuses are of high severity, e.g., hard-coded credentials and potential vulnerabilities allowing Man-in-the-Middle attacks.

5.1.1 Introduction

This section contributes to closing this gap by designing and conducting a study focusing on qualitative examination of reported cryptographic API misuses. With regard to false positives, we are particularly interested in reports that are specific to their concrete usage. An API may be used in different – non-cryptographic –

contexts, and the same constraints do not apply in all contexts. For example, the most misused JCA class `MessageDigest` from previous studies [Gao+19; Krü+19; Rah+19] may be used to compute hashes independent of a security context, e.g., to compute the hash of a file. However, a static analysis cannot know this and has to be conservative. Thus, it may produce false positives and draw an inaccurate picture of the security of our software. With regard to qualitatively judging the severity of findings, we formulate and use a novel, comprehensive risk model. Specifically, we designed and conducted a study to answer the following research questions:

RQ1: *What are common (effective) false positives arising from cryptographic API misuses, e.g., due to a non-security context?*

RQ2: *How severe are the vulnerabilities introduced by cryptographic API misuses in applications?*

We studied cryptographic API misuses of two Java cryptographic API providers – the Java Cryptography Architecture (JCA) and Bouncy Castle (BC) – in open-source Java projects from GitHub [Gitc] using *CogniCrypt* [Krü+19]. We only included projects that are mainly developed and maintained by professionals to address the generalizability problem of open-source studies [Spi+20]. Altogether, we collected 936 Java projects, of which 210 use a cryptographic API and 88.10% have at least one cryptographic API misuse.

To qualitatively analyze the cryptographic API misuses, we randomly picked 157 cryptographic API misuses for review. We observed that one-fourth of the cryptographic API misuses of the class `MessageDigest` – the most misused class according to previous studies [Krü+19; Rah+19; Gao+19] and the second most in our study – occur in a non-security context. Thus, we can consider these cryptographic API misuses as *effective false positives* [Sad+15] (cf. Section 2.2.3) that may not or cannot be fixed by developers.

To judge the severity of all findings, we defined a risk model that connects cryptographic API misuses reported by *CogniCrypt* to security vulnerabilities. This model subsumes the existing vulnerability model for cryptographic API misuses by Rahaman et al. [Rah+19] and includes new threats, e.g., *Denial of Service* (DoS) attacks and *Chosen-Ciphertext Attacks* (*CCA*). Overall, our model marks 42.78% of the cryptographic API misuses as high severity.

In summary, this section makes the following contributions and is structured as follows: *(i)* We studied cryptographic API misuses found in a large, representative

dataset of applications [Spi+20] (cf. Section 5.1.3). *(ii)* We provide empirical evidence that the reported cryptographic API misuses contain a significant number of *effective false positives* (cf. Section 5.1.4). *(iii)* We created a novel, comprehensive risk model mapping cryptographic API misuses to vulnerabilities, introducing previously undiscussed threats like DoS attacks and CCAs (cf. Sections 5.1.2 and 5.1.5). Section 5.1.6 discusses related work and Section 5.1.7 concludes.

5.1.2 Risk Model of Vulnerabilities Introduced by Cryptographic API Misuses

To reason about the potential impact of the reported cryptographic API misuses, we contribute a risk model extending existing models [Rah+19] with more vulnerabilities caused by API misuses. Specifically, we derive our model from a study of *CrySL* [Krü+19] rules written by cryptography experts, the respective APIs, and the cryptographic API misuses observed in our study. In combination with standard attacks and cryptographic API misuses from previous work, our model covers a wide variety of cryptographic API misuses and their attack potential. We list the vulnerabilities, the attack types, the respective severity, the novelty, and the affected error types in Table 5.1. In contrast to more complex existing risk models, such as the *Common Vulnerability Scoring System SIG* [FIR23] and the *Common Weakness Scoring System (CWSS™)* [Mitb], our risk model is tailored specifically to the domain of the study in this section, cryptographic API misuses.

1. **Predictability Through Initialization:** Applications can become insecure if sensitive information like a key is predictable [Ege+13; Rah+19; Laz+14]. As an instance of this vulnerability, we consider predictable and constant cryptographic keys, e.g., a hard-coded key or a predictable password used to derive a key from. Furthermore, a cryptographic secure random number in Java requires a non-predictable seed as well as the usage of a dedicated class, e.g., `java.security.SecureRandom`. If the application code fails to fulfill these requirements, the cryptographic operation becomes predictable. While Rahaman et al. [Rah+19] separated predictable keys and PRNGs, our model combines them, as both attack types are due to predictability.

2. **Predictability Through Usage:** In contrast to attacks of the type *Predictability Through Initialization*, the improper usage of an API can render the respective computation predictable. An example of this issue is the usage of cryptographic APIs which rely on data to be processed, e.g., `MessageDigest`, and

Table 5.1: A risk model of vulnerabilities which can be detected with *CogniCrypt* [Krü+19]. For Novel ● marks if this vulnerability is discussed by Rahaman et al. [Rah+19] and ○ if not. For C: Constraint Error, IO: Incomplete Operation Error, RP: Required Predicate Error, NT: Never Type of Error, FM: Forbidden Method Error, and TS: Type State Error ● marks if this vulnerability can be caused by the respective error type and ○ if not.

Vulnerabilities	Attack Type	Severity	Novel	C	IO	RP	NT	FM	TS
Predictable/constant cryptographic keys	Predictability Through Initialization	H	●	●	○	●	○	○	●
Predictable/constant passwords for PBE		H	●	○	○	○	○	●	○
Predictable/constant passwords for KeyStore		H	●	●	○	○	○	○	○
Cryptographically insecure PRNGs		M	●	○	○	●	○	○	○
Missed to finish cryptographic function	Predictability Through Usage	H	○	○	●	○	○	○	●
Missed to pass data		M	○	○	●	○	○	○	●
Insecure TrustManager	MitM Attacks on SSL/TLS	H	●	○	○	●	○	○	○
Insecure SSL/TLS standard		H	○	●	●	○	○	●	●
Static Salts in PBE	Chosen-Plaintext Attack (CPA)	M	●	○	○	●	○	○	○
ECB mode in symmetric cipher		M	●	●	○	○	○	○	○
Static IVs in CBC mode symmetric ciphers		M	●	○	○	●	○	○	○
Padding Oracle	Chosen-Ciphertext Attack (CCA)	M	○	●	○	○	○	○	○
Fewer than 10,000 iterations for PBE	Bruteforce Attacks	L	●	●	○	○	○	○	○
64-bit block ciphers		L	●	●	○	○	○	○	○
64-bit authentication tag GCM		L	○	●	○	○	○	○	○
Insecure cryptographic ciphers		L	●	●	○	●	○	○	○
Insecure cryptographic signature		L	○	●	○	○	○	○	○
Insecure cryptographic MAC		L	○	●	○	○	○	○	○
Insecure cryptographic hash		H	●	●	○	●	○	○	○
Usage of String	Credential Dumping	L	○	○	○	○	●	○	○
Missed to clear password		L	○	○	●	○	○	○	○
Trigger Exception	DoS Attacks	M	○	●	○	○	○	○	●

fail to process the required data, thus essentially resulting in a predictable computation.

3. **MitM attacks on SSL/TLS:** The improper usage of SSL/TLS can enable an attacker to launch a Man-in-the-Middle (MitM) attack to gain sensitive information [Fah+12]. This includes improper configuration of connections, e.g., incorrect verification of protocols, as well as insecure cryptographic protocols, e.g., TLS 1.1, which are vulnerable to attacks like the POODLE attack.

4. **Chosen-Plaintext-Attack (CPA):** An encryption algorithm should be provably secure against chosen-plaintext-attacks (CPA) [Ege+13]. An example is a static *Initialization Vector* (IV) for the *Cipher Block Chaining* (CBC) mode.

5. **Chosen-Ciphertext-Attack (CCA):** While an encryption scheme should be provably secure against CPA, it should also be safe against chosen-ciphertext-attacks (CCA). Concretely, we cover improper padding schemes, like PKCS5

and PKCS7 in combination with the block cipher mode CBC [Vau02; KR03], and the usage of plain RSA [Ble98], as these are all vulnerable to CCA. Note that a padding attack requires an interaction between the attacker and the program that responds to messages from the attacker. Thus, data at rest is not vulnerable.

6. **Bruteforce Attacks:** Some cryptographic algorithms are vulnerable to repeated, extensive computations, e.g., a feasible collision computation for MD5 and SHA-1 [Ste+17]. Further, primitives like DES [Rah+19] with a block size of 64 bits or a 64-bit authentication tag for GCM [Fer] are vulnerable to brute-force attacks to break the encryption.

7. **Credential dumping:** In Java, string values are immutable and therefore cannot be cleared or overwritten from memory, except when the garbage collector runs, which is out of the developer's control. Thus, the JCA enforces the usage of byte arrays, e.g., `PBEKeySpec`, to handle sensitive information. However, developers may use strings to store this information and only convert it to byte arrays before calling the cryptographic API. Furthermore, classes like `PBEKeySpec` provide the method `clearPassword` to clear the internal copy of the password. However, usage of this class without calling this method renders an application vulnerable to credential dumping.

8. **DoS Attacks:** A Denial-of-Service (DoS) attack limits the availability of a service, e.g., by deliberately overloading the memory or consuming an extensive number of CPU cycles. One possibility to achieve this is by triggering an unintended behavior such as uncaught exceptions [Wu+17]. Such exceptions can be raised when contradicting the API contract, e.g., by missing the initialization of a cipher object[1]. Depending on the program, such an exception can prevent further correct control flow, can cause a program crash, or, if triggered in rapid succession, it can spam the log file, increase CPU usage, or cause IO congestion. A recent example is CVE-2021-23372, where an exception causes the application to crash.

We categorize the severity into high, medium, and low. Our prioritization is based on factors like whether the vulnerability can be exploited remotely, how difficult an attack is to perform, and if the attack leads to a direct gain or needs the combination with other flaws to be of use. Attacks such as MitM, which can be triggered remotely and provide direct benefits for an attacker [Rah+19], are ranked as high severity. We mark vulnerabilities that lead to compromises of secrets for active, dedicated

[1] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/javax/crypto/Cipher.html` (last accessed: 2026-06-06)

attackers as medium severity. These vulnerabilities are of great help in undermining the security of the systems [WA19]. Examples of this are CPA and CCA attacks. An example of a vulnerability with low severity is credentials passed as a string, as these require prior access to the system or running process to be extracted. Thus, the attacker must have exploited other vulnerabilities before to gain access to the system.

5.1.3 Empirical Study

In this section, we describe the setup of our study and the insights gained from it. First, the dataset used for our analysis is described. Then, our research questions are answered in Section 5.1.4 and Section 5.1.5, respectively. We published an artifact (cf. Chapter 9) including our script for the risk model.

Dataset

The dataset created by Spinellis et al. [Spi+20] addresses the generalizability problem of open-source studies and consists of 17,264 GitHub [Gitc] projects that are mainly developed or guided by enterprise employees and span multiple languages. *CogniCrypt* works on Java binaries; hence, we filtered out the non-Java projects and attempted to compile the rest automatically for Maven [Apab]/Gradle [Gra] build instructions. This resulted in 936 Java enterprise-driven, open-source binaries, which serve as the basis of our study. We applied the *CogniCrypt* version 2.7.2[2] with the corresponding ruleset for JCA and BC to them[3]. For each successfully analyzed application in our dataset, *CogniCrypt* created a report including details of the cryptographic objects analyzed and the identified misuses.

5.1.4 Manual Analysis of Reports (RQ1)

To gain a more in-depth understanding of common (effective) false positives, we randomly sampled 157 (Confidence: 99%, margin of error: 10%) cryptographic API misuses. The first four authors of this paper [Wic+22] independently analyzed the sampled cryptographic API misuses, with at least two reviews per cryptographic API misuse. In case of disagreement, the reviewers in question resolved them with

[2] `https://github.com/CROSSINGTUD/CryptoAnalysis/releases/tag/2.7.2` (last accessed: 2026-06-06)

[3] Previous studies used older *CogniCrypt* versions [Krü+19; Gao+19; Rah+19].

further code review until they reached a conclusion. The focus of our analysis is the precision of the report as well as the context of the cryptographic API misuse to answer our first research question.

Undecided Reports. We explicitly marked cryptographic API misuses as *undecided* if it was impossible to judge their validity due to cryptic or confusing error reports. A common reason was that the reported insecure usage could not be identified at the location of the report, nor in the respective callers. In total, we observed 31 cryptographic API misuses which fall into this category. All of these cryptographic API misuses occur for the more complex error types, namely *Incomplete Operation*, *Required Predicate*, and *Type State*. We assume that these error reports will be of little help to the user attempting to make decisions based on them. For the remainder of the discussion, we will focus on the remaining 126 cryptographic API misuses.

Effective False Positives. In total, we observed 33 false positives and 93 true positives (roughly 74%). During our analysis, we observed that some cryptographic API misuse reports can be seen as *effective false positives* (cf. Section 2.2.3). Thus, the user of the analysis would not take any action, as while they may acknowledge that it is theoretically a correct finding, it is invalid or irrelevant for the specific application.

Concretely, we identified 9 out of the 126 cryptographic API misuses from a non-security context. One example is the usage of *MD5* in deeplearning4j[4]. Here, the *MD5*-digest is used to verify that the input array was not modified during execution. Another example is in the project UAVStack[5], where *MD5*-hashes of a file are used to provide content-aware detection of changes.

The number of *effective false positive* cryptographic API misuses due to a non-security context may appear low at first sight. However, one has to consider this number in relation to the overall number of reports for the specific kind, e.g., class `MessageDigest`. It turns out that 22.86% of the `MessageDigest` misuses in our randomly selected sample fall into the category of *effective false positives* due to non-security context. If we consider (a) that roughly 25% of the reported `MessageDigest` misuses are potentially *effective false positives* and (b) that this class causes by far the most of the cryptographic API misuses reported by previous studies [Gao+19; Krü+19; Rah+19], and the second most frequent in our study, it becomes clear that previous reports may contain a significant number of *effective false positives*. Further,

[4]An open-source deep learning library for JVM-based languages (Stars: 12.2k, Forks: 4.9k), `https://github.com/deeplearning4j/deeplearning4j` (last accessed: 2026-06-06)

[5]Graphical tool to monitor and analyze programs running JVM (Stars: 667, Forks: 278), `https://github.com/uavorg/uavstack` (last accessed: 2026-06-06)

this may significantly decrease the acceptance of these tools by developers [Joh+13; Sad+15].

Besides the context, we observed that the 22 true positives, which use a string instead of a byte-array, may result in *effective false positives*. While it is possible to read user input as a byte-array, some credentials are passed by API design as a string instead of the byte-array. Thus, the developer has no effective and easy way to fix the cryptographic API misuse by hand.

Our analysis revealed another potential source for *effective false positives* not considered by previous studies. Some cryptographic API misuses may be intentionally introduced in the projects contained in our dataset. For example, the static analysis tool SonarSource[6] includes tests that do not follow the typical naming scheme for tests, both for correct API usages and for insecure cryptographic API misuses. These cryptographic API misuses, in total 13 in our sample, within the tests are intended.

Observation 5.1 – RQ1

In a non-security context, the usage of strings and "intentional cryptographic API misuses" are common *effective false positives* (cf. Section 2.2.3).

5.1.5 Vulnerabilities (RQ2)

Based upon our vulnerability model introduced in Section 5.1.2, we will discuss the most common vulnerabilities and their severity in this section. Most of the cryptographic API misuses are due to the attack types *Predictability Through Initialization* (876), *Bruteforce Attacks* (400), and *Predictability Through Usage* (343). In total, we identified 1,153 high-severity, 643 medium-severity, and 334 low-severity cryptographic API misuses. We present the number of cryptographic API misuses, their severity, and their relationship to the respective attack types in Figure 5.1.

High-severity vulnerabilities. Vulnerabilities belonging to the attack type *Predictability Through Initialization* are caused by 691 cryptographic API misuses spread over 108 projects. They are due to an insecurely generated key caused by a *Required Predicate* error. Another source of insecurely generated keys is insecure key generation parameters, e.g., *HmacSHA1* or *DES*. These were reported for 84 cryptographic API misuses within 29 projects.

[6] `https://github.com/SonarSource/sonar-java` (Stars: 755, Forks: 501, last accessed: 2026-06-06)

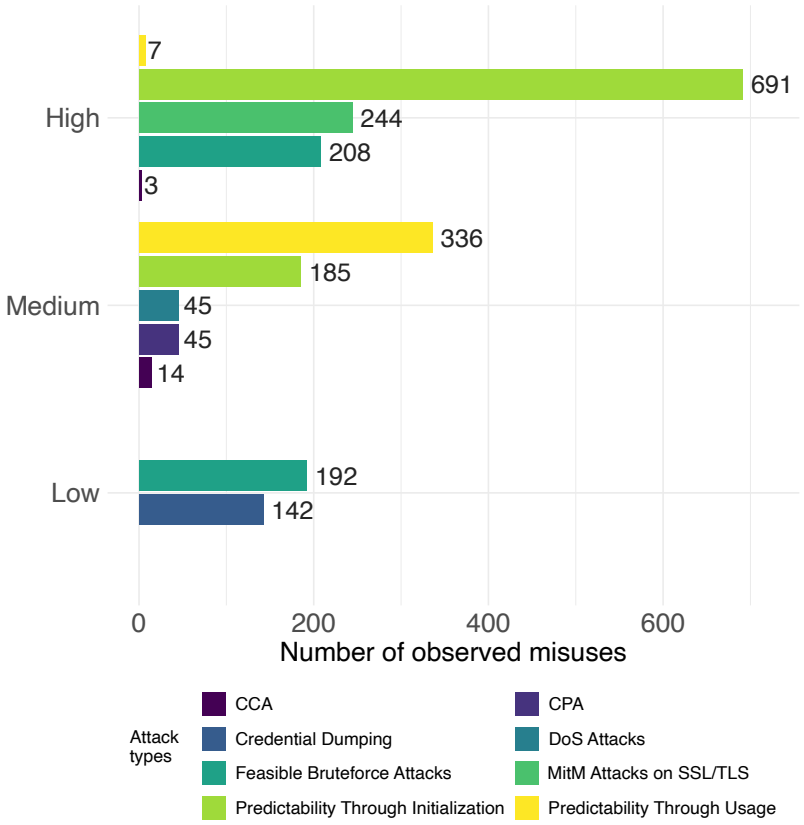


Figure 5.1: Number of cryptographic API misuses by severity for the attack types.

Medium-severity vulnerabilities. Our analysis detected 336 cryptographic API misuses that can result in *Predictability Through Usage* by observing the cryptographic function calls, their call order, and inputs. For example, the analysis identifies at least one path that consumes the cryptographic object without adding any data, e.g., instantiating a message digest without providing an input.

Low-severity vulnerabilities. Most (192) of the low-severity vulnerabilities are of attack type *brute-force* for ciphers, e.g., using an insecure signature algorithm or a 64-bit block cipher.

Observation 5.2 – RQ2

Nearly half the cryptographic API misuses (42.78%) are of high severity and should be prioritized while fixing cryptographic API misuses.

5.1.6 Related Work

Previous studies on cryptographic API misuses have primarily focused on Android applications and identified that 88% to 99% of the applications using cryptographic APIs have at least one cryptographic API misuse [Ege+13; MBB18; Krü+19; Rah+19; Pic+21]. Besides Android applications, Krüger et al. [Krü+19] inspected Maven [Apab] artifacts, mostly Java libraries. To demonstrate the scalability of their tool *CryptoGuard*, Rahaman et al. [Rah+19] analyzed 46 Apache projects in addition to their set of Android applications. All studies focused on precision from the tool's perspective, rather than understanding false positives from the developer's or security analyzer's perspective. An exception is an exploratory study [HGN20] that opened issues for some identified cryptographic API misuses. The results obtained mostly agree with the feedback we received from the disclosure as well as with our manual analysis. In contrast, we do not rely only on the judgment of developers by manually analyzing the cryptographic API misuses.

5.1.7 Conclusion

In this section, we presented a study of Java cryptographic API misuses that focuses on understanding *(effective) false positives* [Sad+15] and the severity of misuses. To understand common *effective false positives* that arise from misuses, we manually reviewed a random sample of misuses and identified several *effective false positives* such as the ones happening in a non-security context. We provide an extensive risk model covering previously undiscussed vulnerabilities such as DoS attacks in the context of cryptographic API misuses. Our risk model marks nearly half of the cryptographic API misuses as high-severity.

Overall, our study reveals several directions for future work. Further, our manual analysis reveals several sources of *effective false positives* that can be addressed in existing static analysis tools.

The results regarding *effective false positives* and the risk model discussed in this section are foundational to developing the final main contribution of this thesis, *SecAI* (cf. Chapter 7), to help developers reduce Java cryptographic API misuses. The risk model is used for the *Severity Score* feature (cf. Section 7.4.7) of *SecAI*, which is intended to aid developers in deciding which warning message to address first. The deepened understanding of *effective false positives* can be used to increase the explainability of *Warning Messages* (cf. Section 3.3.1) in *SecAI*.

In the next section, we present an adaptation of *CogniCrypt* that was motivated by the observation in the manual evaluation process of warning messages that reported an insecure line of code that was caused by another cryptographic API misuse at an earlier location in the code. We have already described the reason for this observation of such error chains in Section 4.5.3, which are *multi-object protocols* that are prevalent in cryptography [Sch+22a]. The adaptation of *CogniCrypt* proposed in the next section is used to study the occurrence of error chains in real-world projects and to conduct expert interviews with five experts indicating usability improvements (cf. Section 5.2). Further, the adaptation was integrated into the final contribution of this thesis, *SecAI* (*Contribution 5*, cf. Chapter 7), and received a user interface in Section 7.4.3.

5.2 Supporting Error Chains in Static Analysis for Precise Evaluation Results and Enhanced Usability

During the manual evaluation in the empirical study of cryptographic API misuses discussed in the previous section, we observed *interconnected* warning reports where misuses spanned over multiple lines in the code. When developers have to identify such connections themselves, this requires additional effort and may lead to *effective false positives* where actual true positives are not resolved. More precise and better explained reporting of static analysis tools can be achieved by supporting the detection of error chains and therefore is important to achieve the goal of this thesis in helping Java developers to reduce cryptographic API misuses. Hence, we supported the effort in the publication of Wickert et al. [Wic+24] in adapting *CogniCrypt* to support the detection of error chains, called $CogniCrypt_{SUBS}$.

The idea of $CogniCrypt_{SUBS}$ is not to necessarily remove warning reports, but to provide useful additional information in the reports for remediation of cryptographic API misuses. In our adaptation, we make use of *CogniCrypt's predicate mechanism* to provide improved reporting by reasoning about *interconnected* warning reports and jointly reporting them. We present a shortened version of this publication with a focus on the adaptation of the analysis algorithm itself, a study on the prevalence of error chains in real-world projects, and an expert interview with five experts indicating usability improvements that also include *effective false positives*.

In the context of this thesis, $CogniCrypt_{SUBS}$ is used to develop the final contribution of this thesis, *SecAI* (*Contribution 5*, cf. Chapter 7). The precise localization of errors (and their connections) increases the explainability of *Warning Messages*

(cf. Chapter 3), and *SecAI* provides a visualization for error chains in Section 7.4.3. In the visualization feature in *SecAI*, we make use of *CogniCrypt$_{SUBS}$'s* ability to reason about error chains and display them in an interactive graph view.

5.2.1 Introduction

Avizienis et al. [Avi+04] summarized established dependability concepts of faults and errors for security. In particular, they point out that one *fault* can cause another *error* and these errors can propagate. The propagation process transforms one *error* into multiple *errors* that can occur across different components. This behavior of API usage is also known as *multi-object protocol* (cf. Section 4.5.3), which is prevalent in cryptographic APIs [Sch+22a; SHA15]. Further, Lipp et al. [LBP22] revealed that static analyses rather report the *manifestation* of an error than its *root* cause (*fault*). They argue that this decision improves the precision of analyses, as not every *fault* manifests as a *vulnerability*. An empirical study [Wic+22] (cf. Section 5.1) revealed that 20% of the analyzed cryptographic misuses could not be resolved, as the *root* cause did not match the reported location. Concretely, while a misuse was reported, the required fix was unclear from the report. Summarizing, especially in cryptography, errors can propagate. Therefore, an error can depend on other errors, thus resulting in error chains. Such a *dependent error* in an error chain is a *subsequent error* to its *preceding errors* in the error chain.

Typically, studies and tools do not address error chains. In this section, we aim to close this research gap. We focus on the domain of cryptographic API misuses in Java as a well-researched area with many detection tools that cannot (yet) provide this granularity of information. Concretely, we want to understand to what extent error chains occur in the wild, if analyses that present the complete error chain are usable in practice concerning the runtime, and if the presentation of an error chain can improve the usability of a security-focused static analysis. To achieve this, we will introduce *CogniCrypt$_{SUBS}$*, which is our adaptation of an existing static analysis to allow the detection of error chains. We will answer the following two novel research questions:

RQ1 *What is the distribution of dependent errors for real-world applications?*

RQ2 *To what extent do experts consider CogniCrypt$_{SUBS}$ as usable compared to an error-chain-unaware version of CogniCrypt?*

By answering these research questions, we aim to provide new insights into error chains in practice. Understanding error chains for cryptographic API misuses in Java can support us in understanding empirical studies in more depth. In particular, the insights can help us to understand the real-world prevalence of error chains. Further, we gain new insights into the challenges of expressing the dependency between a root error and its subsequent errors for static analyses. In addition, our expert interview sheds light on the possible usability implications of reporting error chains.

To address this problem, we designed the first adaptation of an existing static analysis algorithm to distinguish between root and subsequent errors. We adapted the algorithm of an existing and established security-focused static analysis, namely *CogniCrypt* [Krü+19]. *CogniCrypt* was used in several empirical studies [Krü+19; Wic+22; HGN20; Gao+19] and can be used to analyze further APIs beyond cryptography. Further, we quantitatively evaluated real-world repositories with our adaptation, called $CogniCrypt_{SUBS}$, measured the runtime overhead, and conducted a user study.

The empirical evaluation for **RQ1** revealed that at least half of the projects with at least one cryptographic API contain error chains. The expert interview for **RQ2** indicates that $CogniCrypt_{SUBS}$ helped the participants identify the error chain more quickly and reduced the required interaction time with the tool.

Overall, this section presents the following contributions:

- We present an algorithm for static analyses that can distinguish between a root and subsequent errors (cf. Section 5.2.4).
- We improve an existing open-source analysis to integrate the suggested changes (cf. Section 5.2.4).
- We show that error chains occur in the wild and affect every second analyzed project.
- We report on an expert interview that revealed that the participants appreciate the structured reporting of root and subsequent errors and require fewer executions of the analysis to resolve all potential problems.

In the remainder of the section, we first introduce cryptographic API misuse basics with an example, followed by a description of the terminology to present error chains (cf. Section 5.2.2). We describe the required adaptations to an existing algorithm in Section 5.2.4. Section 5.2.5 briefly describes the motivation of our research

questions, methodology, and findings. We cover related work in Section 5.2.6 and limitations and future work in Section 5.2.7. The section ends with a conclusion in Section 5.2.8 and information about our artifacts in Chapter 9.

5.2.2 Cryptographic Misuse Example

Listing 5.1: A simplified example of an insecure encryption observed in a real-world project. ⚠ represents a reported cryptographic API misuse. Using `DESKeySpec` in Line 17 is insecure, too, but it is reported in Line 19.

```
15  public class FileEncrypt {
16  private static Key generateKey(String password) throws Exception {
17    DESKeySpec dks = new DESKeySpec(password.getBytes("utf8"));
18  ⚠     SecretKeyFactory keyFactory = SecretKeyFactory.getInstance("DES");
19  ⚠     return keyFactory.generateSecret(dks);
20  }
21
22  public static String encryptFile(String password, String srcFile, String destFile) {
23  ⚠     IvParameterSpec iv = new IvParameterSpec("123456".getBytes("utf-8"));
24  ⚠     Cipher cipher = Cipher.getInstance("DES/CBC/PKCS5Padding");
25  ⚠⚠    cipher.init(Cipher.ENCRYPT_MODE, generateKey(password), iv);
26    InputStream is = new FileInputStream(srcFile);
27  ⚠     CipherInputStream cis = new CipherInputStream(is, cipher);
28    // write cis into destFile and return
29  }
30 }
```

To implement our static analysis adaptation *CogniCrypt*$_{SUBS}$ to detect error chains, we extend the static analysis *CogniCrypt* [Krü+19]. The necessary information to understand our approach (cf. Section 5.2.4) was introduced in Section 2.2. In Listing 5.1, we present an example of an insecure file encryption in Java. Our example has two methods, `generateKey` (cf. Line 16) and `encryptFile` (cf. Line 22). The method `generateKey` generates a secret key using the JCA classes `DESKeySpec` and `SecretKeyFactory` from the given password. This method is called by the method `encryptFile` in Line 25. The method `encryptFile` generates an initialization vector (cf. Line 23), generates and initializes a `Cipher` object in Line 24 and Line 25, respectively, and passes the `Cipher` object to a `CipherInputStream` in Line 27.

Listing 5.1 exhibits seven different cryptographic misuses detected by *CogniCrypt*. The first and second misuses on Line 18 and Line 19, respectively, occur because the insecure algorithm `DES` is passed to the object `keyFactory` and the class `DESKeySpec` is used. These two misuses result in another report for the second parameter of the `Cipher.init` method in Line 25, which results in an insecure cipher input stream object in Line 27. Further, the misuse in Line 23 is due to the static initialization vector, causing another misuse for the third argument of the `Cipher.init` call in

Line 25. Finally, the `Cipher` object is initialized with an insecure algorithm in Line 24 that also renders the `Cipher` object insecure.

Terminology

Furthermore, Listing 5.1 illustrates the concept of *ensuring* and *specified objects*. The rule for the JCA class `CipherInputStream`[7] requires that the argument (cf. Line 27) of the class `Cipher` is generated securely. Thus, the object `cis` requires a security guarantee from the class `Cipher`. The object `cis` is the *specified object* and the object `cipher` is the *ensuring object*. As the *ensuring object* is insecure due to the use of DES[8] as the encryption algorithm (Line 24), the predicate is not ensured, resulting in a `RequiredPredicateError` (cf. Line 27).

5.2.3 Analysis Algorithm

The static analysis *CogniCrypt* uses allowlisting rules to identify cryptographic API misuses [Krü+19] (cf. Section 2.2.2). We present a shortened and simplified version of the (final) algorithm in Listing 5.2. The analysis first parses the rules and builds a FSM for each rule (cf. Line 32). Each FSM encodes the call order for one class that is encoded in the respective *CrySL* rule. Afterward, the analysis generates a call graph with the analysis framework *Soot* [Val+10] (cf. Line 33). This call graph serves as a starting point to identify allocation sites (API usages) that match the initial edge of all FSMs (cf. Line 35). The identified allocation sites are used as initial seeds for the analysis.

For each identified seed, *CogniCrypt* uses IDE^{al} [SAB17] and the matching FSM to perform a flow-, field-, and context-sensitive typestate analysis. To retrieve the parameter values of methods that are invoked on the seed object, *CogniCrypt* executes a backward pointer analysis [Spä+16]. Both the results of the typestate and backward analysis are stored in the respective seed. A seed uses this information to report errors in three stages. First, the results of the typestate analysis are used to report *typestate errors* (cf. Section 2.2.2, cf. Line 40). Second, a seed validates the specified constraints and reports *constraint errors* (cf. Section 2.2.2, cf. Line 41).

[7] The respective *CrySL* rule is available online and the discussed requirement is specified in Lines 30 and 31 (tag 3.0.2-jca): `https://github.com/CROSSINGTUD/Crypto-API-Rules/blob/master/JavaCryptographicArchitecture/src/CipherInputStream.crysl` (last accessed: 2026-06-06)

[8] The respective *CrySL* rule is available online and the secure values for the argument are specified in Lines 88 to 118 (tag 3.0.2-jca): `https://github.com/CROSSINGTUD/Crypto-API-Rules/blob/master/JavaCryptographicArchitecture/src/Cipher.crysl` (last accessed: 2026-06-06)

Third, a seed triggers the propagation of security guarantees, such as that a key is generated securely (cf. Line 39).

The aim of the propagation is to track security guarantees across multiple objects. Each security guarantee is a *predicate*. If a seed has no violated constraint and all required predicates (cf. Line 46), such as a salt for a password-based encryption must be randomized securely, are ensured, it ensures its predicate (cf. Line 47). The *ensured predicate* is passed to a handler that passes the newly *ensured predicate* further to all objects with a *CrySL* rule that use the object of the *ensured predicate* (cf. Line 51). The propagation repeats when a seed receives a new *ensured predicate* and terminates when all seeds have finished ensuring any predicate (cf. Line 39).

Listing 5.2: Simplified pseudo-code of the adapted analysis algorithm.

```
31 // Initialize analysis
32 fsms := parseRules(cryslRules)
33 cfg := generateCallGraph(program)
34 // Pre-analysis to identify cryptographic objects
35 for edge.allocation(fsms.classes) in cfg {
36   objects.add(edge)
37 }
38 // Analyze cryptographic objects traces
39 for seed in objects {
40   identifyTypeStateErrors(seed)
41   identifyConstraintErrors(seed)
42 }
43 // Propagate until all seeds updated
44 for handler.hasUpdates() {
45   for p in handler.updatedSeeds().preds() {
46     if p.noViolatedConstraints() && p.requiredPredicateError() {
47       ensuredPredicate(p)
48     } else {
49       unEnsuredPredicate(p)
50     }
51     handler.passToUsingObjects(p)
52   }
53   // Handle and map subsequent errors
54   for rpe in requiredPredicateErrors {
55     if rpe.isSubsequent() {
56       precedingErrs := rpe.getUnEnsuredPred()
57       identifyMatching(precedingErrs)
58     }
59   }
60 }
```

5.2.4 Design

In this section, we first discuss relevant terminology (cf. Section 5.2.4) using the example in Listing 5.1. We then explain our approach and necessary algorithm adaptations to detect error chains (cf. Section 5.2.4).

Terminology

To securely encrypt the file, the code must be modified at four locations, and yet, analyses, such as *CogniCrypt*, tend to report up to seven misuses (cf. Listing 5.1). All reports in this example are true positives. However, the misuses actually form a tree of misuses (cf. Figure 5.2). We refer to errors with *preceding errors* as *subsequent errors*. A *subsequent error* is caused by another misuse that is the respective *preceding error*. For instance, the misuse reported in Line 25 is a *subsequent error* caused by the insecure initialization vector from the *preceding error* in Line 23. If a developer fixes a *preceding error*, the *subsequent error* is fixed, too (if all *preceding errors* are resolved). As a *preceding error* can be a *subsequent error* too, we use the term *root error* to describe any *preceding error* that is not a *subsequent error*.

In Figure 5.2, the `ConstraintError` in the right top (cf. Line 18) is a *root error* and is a directly *preceding error* for the `RequiredPredicateError` in the middle (cf. Line 25). This error is the *subsequent error* (cf. Line 25) of the *root error* (cf. Line 18) and it as well is a *preceding error* of the `RequiredPredicateError` (cf. Line 27) at the bottom of Figure 5.2. This error is the *subsequent error* (cf. Line 27) of the two *preceding errors* (cf. Line 18 and Line 25).

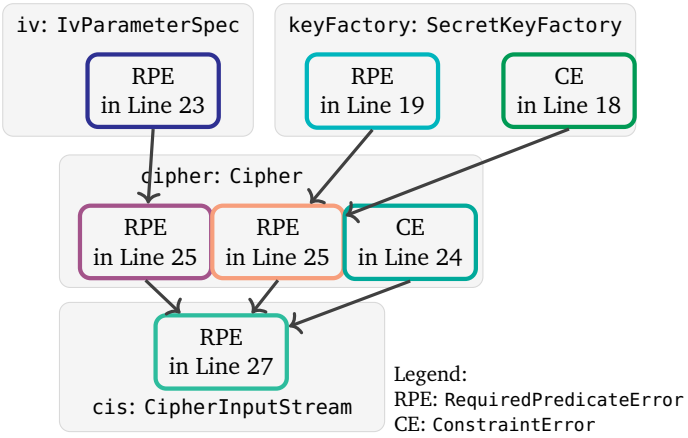


Figure 5.2: Dependent error tree of an error reported for `CipherInputStream` in the code of Listing 5.1. The gray boxes mark different objects, and the different box colors help to differentiate misuse locations.

Typically, *subsequent errors* can be fixed by the developer by fixing the line of code of the *root error*. If a static analysis tool reports a *subsequent error* without the context of its (*preceding error(s)* and) *root error*, a developer needs to identify the cause (*root error*) themselves. This may lead them to discard fixing the error, resulting in an *effective false positive* [GE09; Sad+15; Wic+22], and ultimately may hinder the adoption of such tools [Joh+13; CB16]. The current implementation

of *CogniCrypt* — and other analyses in the domain — only reports errors unsorted without reporting their connection. With $CogniCrypt_{SUBS}$, we therefore enhance the analysis and reporting of *CogniCrypt* to reason about error chains.

Unensured Predicate Approach Implementation

In the discussed example in Listing 5.1 and Figure 5.2, we could see that error chains occur when passed arguments do not hold a required property. Such properties can be described as *predicates* (cf. Section 2.2.2). Since *CogniCrypt* already provides a mechanism to identify *predicates* (cf. Section 5.2.3), we decided to adapt its static analysis algorithm, and we call it the *Unensured Predicate Approach*.

Observation 5.3 – Unensured Predicate

An *unensured predicate* is generated if an *ensuring object* fails to validate all its specifications (i.e., no *predicate* was generated).

Our approach is based on the concept of tracking *unensured predicates* — whenever a predicate failed to be ensured, we collect an *unensured predicate* instead of an *ensured predicate* (cf. Line 49). The main design decision for our implementation is that *unensured predicates* should be propagated in the same way as *ensured predicates*. In this way, *subsequent errors* can be clearly linked to their *preceding error*. Consequently, we enriched the existing propagation mechanism of *CogniCrypt* to integrate the *unensured predicate* propagation. Our adaptation (Listing 5.2) of the *CogniCrypt* algorithm (cf. Section 5.2.3) has three main changes: the generation of *unensured predicates* (cf. Line 49), the detection of *subsequent errors* (cf. Line 55), and the mapping of *subsequent* and *preceding errors* (cf. Line 56 and Line 57).

To ensure that the algorithm propagates *unensured predicates* the same way as it propagates *ensured predicates*, we handle *unensured predicates*[9] and *ensured predicates*[10] as subclasses of `AbstractPredicate`[11]. A seed generates a predicate either as ensured or unensured (cf. Line 46). An *unensured predicate* is generated when a seed does not ensure a predicate (cf. Line 49). The generated *unensured predicate* is passed further to any object that might require the *unensured predicate* to be ensured

[9] `https://github.com/CROSSINGTUD/CryptoAnalysis/blob/develop/CryptoAnalysis/src/main/java/crypto/predicates/UnEnsuredPredicate.java` (last accessed: 2026-06-06)

[10] `https://github.com/CROSSINGTUD/CryptoAnalysis/blob/develop/CryptoAnalysis/src/main/java/crypto/predicates/EnsuredPredicate.java` (last accessed: 2026-06-06)

[11] `https://github.com/CROSSINGTUD/CryptoAnalysis/blob/develop/CryptoAnalysis/src/main/java/crypto/predicates/AbstractPredicate.java` (last accessed: 2026-06-06)

(cf. Line 51). Therefore, a seed receives *unensured predicates* by the same logic as *ensured predicates*.

To connect the different errors to generate an error chain, we rely on our hypothesis that subsequent errors are caused by unfulfilled predicates. Thus, the seed checks for each determined and reported *required-predicate error* if this error is a subsequent error (cf. Line 54 and Line 55). The error is subsequent if the seed received an *unensured predicate* (cf. Line 56) that includes the reason for the *missing ensured predicate* (cf. Line 57). Consider the example in Figure 5.3. The seed for `Cipher` has a `RequiredPredicateError`. Thus, the seed (cf. Line 56) receives the *unensured predicate* that references the seed for `SecretKeyFactory`. As this seed has a `ConstraintError`, the algorithm can determine that this `ConstraintError` is the *root error* of the discussed `RequiredPredicateError`.

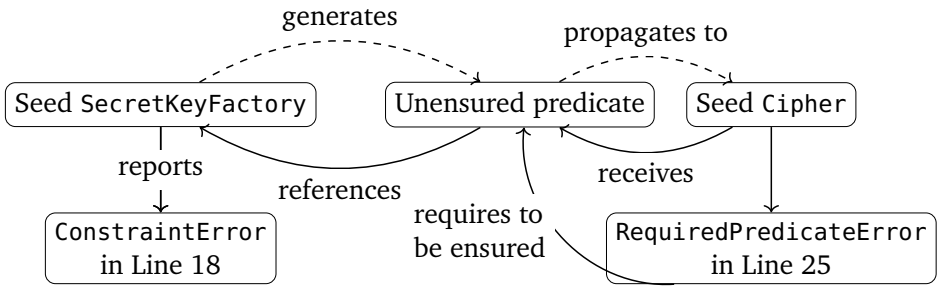


Figure 5.3: Example reference between the `SecretKeyFactory` and `Cipher` seed for Listing 5.1.

Note that our implementation of the *Unensured Predicate Approach* enabled us to implement another improvement for a well-known over-approximation causing false positives. For instance, a `Cipher` object uses an `IvParameterSpec` object and requires a securely randomized byte array generated by an object of `SecureRandom` as `IV` only in the case of encryption and not for decryption. *CogniCrypt* falsely reports a `RequiredPredicateError` for a static `IV` for decryption, and our change in *CogniCrypt*$_{SUBS}$ and the *CrySL* ruleset no longer reports this false positive. For this change, we leveraged an unused feature of *CrySL's* grammar that can express implications, and adapted the *CrySL* rules to include the implications where required. Due to our changes with the *unensured predicate* approach, we can track the implications. We call this change that avoids instances of the above-mentioned false positives backward error tracking (BET).

Overall, we make use of *CrySL's ENSURES* and *REQUIRES* sections as well as its ability to express implications to *(i)* reason about error chains and to *(ii)* reduce false positives. We evaluate the usability implications of reporting error chains in an expert interview in Section 5.2.5.

5.2.5 Evaluation

This section presents the results of our empirical study, benchmark, and expert interview. For each research question, we describe the motivation and methodology used to answer the research question and discuss the obtained results.

Large-Scale Study of JCA Misuses in the Wild (RQ1)

Previous work has pointed out that static analyses tend to report the location where a bug or vulnerability manifests, rather than where it is required to be fixed [LBP22]. For Listing 5.1, consider the insecure initialization vector. The insecurity manifests in Line 25 and should be fixed in Line 23. For cryptographic misuses, previous work showed that a missing distinction between the dependencies of errors can draw imprecise results of API misuses in the wild [Wic+22]. Further, this may hinder the adoption of static analyses for this critical field, as this may cause *effective false positives* [Sad+15; Wic+22] (cf. Sections 2.2.3 and 5.1.4). *Effective false positives* are true positives reported by the analysis that are perceived as false positives and not fixed by developers because, e.g., developers do not understand the report explanation [Sad+15]. Hence, the motivation of this research question is to understand the dependency between errors in the wild for cryptographic API misuses. In particular, we were interested in whether dependent errors are common in real-world projects, how errors can depend on each other, and which cryptographic classes are most commonly affected.

Methodology

To analyze real-world projects, we mined popular projects on GitHub [Gitc]. We used a repository miner[12] to collect popular and active Java projects (cf. Section 5.2.1). Concretely, all projects should be Java projects, have at least 100 stars (popular), at least one commit between 02/22/2022 and 08/22/2022 (active at the time of mining), and be no fork (no duplicates). These filters resulted in 4,022 distinct repositories.

To analyze the projects with *CogniCrypt*$_{SUBS}$, we require binaries. Therefore, we first filtered for projects containing a JCA class, namely `java.security`, `javax.crypto`, or `java.*`, since our target domain is cryptography, and identified 2,054 projects that may use the JCA. Second, we tried to compile the remaining 2,054 projects

[12] `https://github.com/marvinvo/Repository_Miner` (last accessed: 2026-06-06)

automatically with Maven [Apab] and Gradle [Gra] with a time limit of 20 minutes. We succeeded for 634 repositories. To ensure that the projects use JCA, we executed a lightweight *OPAL* [Hel+20] analysis that identifies if any class of the JCA is used in the binaries. This step reduced the number of projects that we analyzed with *CogniCrypt*$_{SUBS}$ to 505 repositories with 833 modules in total. We use the term modules to refer to the number of `bin` folders that we obtained through the compilation process.

With *CogniCrypt*$_{SUBS}$, we analyzed 505 Java repositories to identify error chains in the wild. We analyzed each repository with 3 GB memory, 2 MB stack size, and a timeout of 20 minutes for each project. With these restrictions, we received reports for 471 repositories and 783 modules. The repositories have 100 to 60,637 stars, and the median date of the latest commit was 15 days before we started the study.

Findings

To answer **RQ1**, we first focus on the quantitative results obtained through our analysis. Second, we discuss one large example as a more qualitative inspection.

Quantitative Results *CogniCrypt*$_{SUBS}$ reports at least one error for 306 of the 471 analyzed repositories. In total, *CogniCrypt*$_{SUBS}$ reports 3,964 errors, with an average of 8.41 errors per repository and 5.06 per module. `RequiredPredicateErrors`, `ConstraintErrors`, and `IncompleteOperationErrors` are the most common errors that *CogniCrypt*$_{SUBS}$ observes in our dataset. Table 5.2 presents the number of errors for each of the different error types defined by *CogniCrypt* [Krü+19] (cf. Section 2.2.2). In our dataset, error reports for the JCA classes `Cipher`, `MessageDigest`, and `SSLContext` cause most errors. The usage of the insecure algorithm `MD5` or `SHA1` in `MessageDigest` is the most commonly reported misuse, with in total 297 errors in 213 modules and 179 repositories.

Table 5.2: Overview of the number of identified cryptographic API misuses.

Error Type	Errors	Root Errors
`RequiredPredicateError`	1617	353
`ConstraintError`	810	111
`IncompleteOperationError`	732	1
`TypestateError`	185	8
`HardCodedError`	183	127
`NeverTypeOfError`	176	117
`ForbiddenMethodError`	17	-

In 155 (50.7%) repositories of the 306 repositories containing at least one error, *CogniCrypt$_{SUBS}$* reports at least one dependent error. In our dataset, 717 errors are marked as root errors, and 353 `RequiredPredicateErrors` are the most prevalent error type. Overall, only 9 root errors are typestate errors. Therefore, root errors caused by a *specified object* that does not reach a predicate-ensuring state are edge cases in our dataset. The remaining 98.7% of the root errors are constraint errors (cf. Section 2.2). The number of root errors caused for each error type is presented in Table 5.2.

In Listing 5.1, we showed that one subsequent error may have multiple preceding errors. The results of *CogniCrypt$_{SUBS}$* confirm this in practice. On average, each root error has 1.1 directly following subsequent errors. Each subsequent error has on average 1.54 preceding errors. For 134 subsequent error trees, the depth is equal to or higher than 3. On average, a dependent error tree is a directed-acyclic graph with 3.83 errors, and the largest in our dataset contains 22 errors.

A few JCA and JSSE classes cause dependent errors. Overall, *CogniCrypt$_{SUBS}$* reported dependent error pairs for 24 distinct JCA classes. Most of the root errors are reported for the JCA classes `SecretKeySpec` (240), `KeyStore` (150), and `KeyManagerFactory` (96) that either serve as a key store or wrapper. Thus, insecure keys cause most of the dependent errors. As a result, the three JCA classes with most of the subsequent errors require a key for their task. Namely, the JCA classes `Cipher`, `SSLContext`, and `Mac` with 219, 79, and 78 reported subsequent errors, respectively. Further, three JCA classes, namely `AlgorithmParameterGenerator`, `IvParameterSpec`, and `GCMParameterSpec`, cause only root errors in our study.

We present an overview of the number and dependencies between these classes in Figure 5.4. For instance, the example in Listing 5.1 and Figure 5.2 has a *root error* in Line 23 for the class `IvParameterSpec` that propagates as a *subsequent error* in Line 25 for the class `Cipher` that also propagates as a *subsequent error* in Line 27 for the class `CipherInputStream`. The same error chain can be seen in Figure 5.4 with 32 *root errors* for `IvParameterSpec` that has a directed edge to the class `Cipher`, showing that all 32 *root errors* propagate to `Cipher`. In total, the class `Cipher` had 228 *subsequent errors*, of which 16 were propagated to the class `CipherInputStream`, resulting in 7 *subsequent errors*. The lower number of *subsequent errors* is expected, as shown in Figure 5.2; multiple (3) *subsequent errors* in the class `Cipher` can result in fewer (1) *subsequent errors* in the class `CipherInputStream`.

Note that we removed self-loops from the classes `SecureRandom` and `KeyPair` caused by requirements within the class. For example, an instance of `SecureRandom` can

be initialized with a seed. This seed needs to be securely randomized, and this requirement can be ensured by another call sequence of `SecureRandom`.

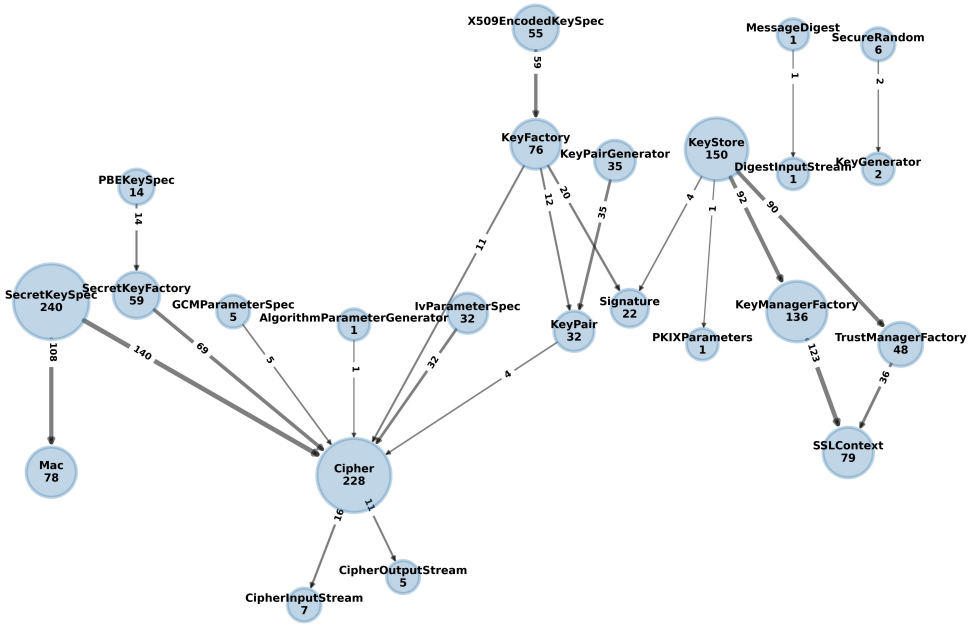


Figure 5.4: JCA classes and their number of dependent errors. Each node presents the number of (dependent) errors for the JCA class. A directed edge shows the number of preceding errors that cause subsequent errors in another class.

Observation 5.4 – RQ1

In the wild, every second project with cryptographic misuses has dependent errors.

Case Study In this case study, we investigate one of the largest chains of subsequent errors found in our study. To calculate the length, we computed each error's preceding and subsequent error tree and appended the subsequent tree to the preceding error tree. This step results in a graph that we call the *dependent error tree* of an error. The case study discusses one of the errors with the most edges in its dependent error tree.

We already discussed this case as an example in Figure 5.2 (cf. Section 5.2.4), which presents the dependent error tree for insecure file encryption. Listing 5.1 presents a minified example resulting in this dependency tree. We observed this tree in a Java project with 699 stars, and the tree is caused by three mistakes to securely encrypt a file that we discussed in more depth in Section 5.2.4. First, the initialization vector is insecure and results in three `RequiredPredicateErrors` as

shown in the left part of Figure 5.2. Second, the key is insecure and causes the subtree that originates from the `RequiredPredicateError` and `ConstraintError` of the JCA class `SecretKeyFactory`. Third, the encryption algorithm is insecure and causes the `ConstraintError` in the `Cipher` object. Therefore, *CogniCrypt*$_{SUBS}$ correctly reported this dependent error tree. We also did runtime measurements showing that subsequent error detection causes a median runtime overhead of 1%, and with all improvements activated, of 4% (more details cf. [Wic+24]).

Expert Interview (RQ2)

In the previous research questions, we focused on subsequent errors that exist in the wild and the runtime overhead caused by tracking subsequent errors. Our results suggest that subsequent errors exist in the wild, and the runtime overhead for their detection is low. Thus, we were interested in understanding if information about subsequent errors provides additional value. Besides empirical analyses, reports may also be interesting for users of an analysis. Specifically, we were interested in whether grouping and matching subsequent errors to root errors can lead to an improved user experience.

Methodology

For this research question, we performed an expert interview with a prototype we built that presents root errors and matches subsequent errors grouped below the root error. The prototype differs from the current implementation of *CogniCrypt*$_{SUBS}$, which only reports references between errors, such that the output is formatted to present a root error and the subsequent errors beneath it. During our in-the-wild study (cf. Section 5.2.5), we observed that errors sometimes occur in tree structures. Presenting this information is interesting but goes beyond the question of whether users may benefit from reports of root errors, so we used a simple presentation. Nonetheless, we believe that future work on the usable presentation of subsequent errors is worth exploring.

We recruited five experts for the expert interview. The experts were doctoral- or post-doctoral researchers who are experienced in four different areas. Two researchers are experts in static analyses, one researcher focuses on software tools and domain-specific languages, another is an expert in distributed programming languages, and the last one focuses on IT security and distributed systems. Our experts have, on average, 12.6 years of experience with Java, and everyone has developed at least

one project in Java. Three participants developed several big projects, and of these, one thinks they are an expert. Of our experts, two hesitated to call themselves a Java expert, as they did not keep track of the latest Java features. All our experts had at least one course about cryptography at university, and three have deeper knowledge because they either used cryptography before, have an IT-Security degree, or even taught cryptographic concepts. Further, three of our participants have prior experience with the Java standard library for cryptography that is used in our code examples.

For the interviews, we used semi-structured interviews. Specifically, we briefly introduced our research and collected the profile of the participants. After the introduction, we asked each participant to solve two different code examples. For each code example, the participants used either the version of *CogniCrypt* that supports subsequent errors or not. Due to the number of participants, we could not draw statistical conclusions and did not mix tasks and tools. Thus, the first task should be solved with *CogniCrypt* and the second task with our prototype. For the tasks, we asked the participants to follow the think-aloud approach [Jas+04; VBS+94], and requested the participants to inspect the reports of the analysis. Thus, we avoided experienced developers resolving the bugs independently of the analysis. We allowed our participants to ask clarification questions or use the internet. After each task, we asked the participants questions on the usability of the respective analysis tool. In the end, we resolved the differences between both rounds and asked final questions. Overall, the interviews had a duration of 40 to 60 minutes. We provided the questions in our artifact (cf. Chapter 9).

We executed the expert interviews with a remote-desktop tool to access a virtual machine with Eclipse [Ecla] installed as an IDE. The participants could execute the tool by running a script from the integrated terminal. To review the actions and their think-aloud, we asked them in advance for consent to record the screen of the VM and audio. Further, we asked each participant for their explicit confirmation when we used direct quotes to ensure that we preserve the intended meaning.

The cryptographic API misuses of both code examples can be fixed by altering one line of code, and are as similar as possible regarding the overall number of errors and root errors. Both examples cover the transmission of a secret with AES encryption. We created the examples based on our experience in the domain with a focus on tasks that can be solved in a reasonable time. The concrete root errors differ between the two examples to avoid a learning effect between them. The first example spans over four class files with four root errors within three class files. Due to subsequent errors, *CogniCrypt* reports nine misuses. The second example spans over three class

files that are slightly larger than the class files of the first example. Overall, the second example has three root errors in two class files resulting in a total of ten misuses. The tasks are in our artifact (cf. Chapter 9).

Findings

For the first code example, the participants started to resolve the reports by *CogniCrypt* by applying two main approaches. Three participants started with the error reports for the main class and used them as an entry point for fixing the example. In contrast, two participants started to resolve the errors in the order that *CogniCrypt* reported them. One participant commented on each error report before they started resolving the error, and thus realized quickly that some errors must be related somehow. Further, some participants used the call-hierarchy function of *Eclipse* [Ecla] to identify the insecure parameter for subsequent errors. In addition, the participants skipped error messages they did not understand or referred to the exact fix location. During the fixing process, these errors were often revealed as subsequent errors. Thus, the skipping tactic proved to be successful for this task.

Error chains caused confusion of participants when a root error was fixed. In particular, when a participant resolved a root error, the number of misuses decreased by more than one. Due to their experience, the participants realized that the errors must be linked to each other or are not relevant anymore. For example, participant four (E4) said: *"No violations found... huh... I only fixed two bugs and three are gone. That's always cool."*. However, in some cases, the participants did not always execute the analysis after resolving a root error and were confused about a subsequent error. For example, participant five (E5) said: *"It is not clear to me how it detects if the key is generated properly... Maybe it detects this from here [pointing to generateKey() method] then it is potentially already fixed. I need to run the analysis again."*

For task 2, the chosen approaches to solve the misuses were more similar among the participants. This time, all participants started with the first reported error that was in the main class. Therefore, the participants may not have changed their approach to resolve the reports. Overall, all participants ran the analysis tool less often than during the first task. Most participants presumed without any explanation from us that they do not need to address subsequent errors, such as participant one (E1): *"Subsequent errors... aha... so this is the Cause-Effect-Chain I'm guessing. Maybe I'll just ignore those and see what happens"*.

The expert interview revealed that reporting subsequent errors to users may have a positive impact on the user experience. All participants agreed that *CogniCrypt*

and our prototype supported them in solving the tasks, such as participant five (E5): *"[...]the first tool without the grouping, without the prioritizing each of the error messages had to be taken into consideration to see if they are relevant just to find they have no real meaning because the problem doesn't occur at this location. With the second tool, I immediately found the right errors. This, of course, is much more pleasant to use [...]."* Further, all experts agreed that during the second task it was clearer which error should be resolved first. However, one expert credited this to the experience gained during the previous task. All five participants would pick our prototype for use in a security-related project. In addition, two experts are affirmative that subsequent error detection makes the process of solving cryptographic misuses easier. The three other experts had mixed feelings, as the change '*only*' helps to identify the location, while the fix is equally challenging for both tools. Three experts agree, one expert is unsure if the presentation of the tool's output increases the acceptance, and one expert disagrees, as they argue that for security no error should be ignored.

Observation 5.5 – RQ2

The expert interviews indicate that reporting *root errors* supports users in fixing the relevant errors first.

5.2.6 Related Work

Dependent Errors Previous work on fault localization sheds light on the limitations of the assumption that bugs are localized and isolated. Multiple studies have indicated that bugs are not localized [HG09; Luc+12; DW09; Won+16]. Specifically, faults can spread over many methods or files [HG09; Luc+12] and even interfere with each other [DW09]. Thus, their work confirms that bugs can depend more on each other than usually presented and reported by static analyses.

Lipp et al. [LBP22] discussed that vulnerability reports may include the root cause, while static analyses tend to report the manifestation location of a vulnerability. To evaluate static analysis tools against vulnerability reports, Lipp et al. [LBP22] analyzed for one application if the root cause and manifestation location are in the same function. Their work focused solely on one subsequent error, and their empirical study focused on one application. In contrast, we consider error chains and evaluated 471 applications in our empirical study of error chains.

Besides empirical studies, Avizienis et al. [Avi+04] describe a taxonomy of security errors. The taxonomy describes that a root cause can cause an error that

is propagated within a program and between different components. The propagation of errors that may cause a fault or vulnerability is called *chain of threats*. While this work describes the propagation in theory, our work showed that the error propagation can cause a tree in practice rather than a chain for cryptographic misuses.

Cryptographic Misuses in the Wild The majority of empirical studies of cryptographic misuses in the wild draw an alarming picture of today's software security. For Android applications, studies report that 88% to 99.7% of the applications contain at least one misuse [Ege+13; Krü+19; Pic+21]. An empirical study draws a similar insight for iOS applications [FMS18]. Another study [Li+22] revealed that even Go programs using an API that avoids some misuses by design [fLP13] struggle with cryptographic misuses. Within all analyzed applications, 83% had at least one cryptographic misuse [Li+22]. For Python applications, a study revealed that 52% of the applications using a cryptographic API have a misuse [Wic+21]. For Maven [Apab], a study revealed that 63% of the artifacts have at least one cryptographic API misuse [Krü+19]. In contrast to most studies, we "only" identified that 65% of the analyzed applications have a misuse. Thus, our results are similar to the one by Krüger et al. [Krü+19] for Maven artifacts. However, no previous work inspected the dependencies between cryptographic misuses.

Usability of Security Warnings Previous work has shown that a high false positive rate hinders the adoption of a static analysis [Joh+13; CB16; Sad+15]. Our expert interview revealed that *CogniCrypt*$_{SUBS}$ may help to reduce confusing reports, which may otherwise remain unresolved [GE09] or be considered effective false positives [Sad+15]. Further, Johnson et al. [Joh+13] reported that poorly presented output has a negative impact on the usability of a tool. However, some participants think that a more intuitive presentation can minimize the effect. Our changes are a step in this direction, according to our experts. Further, the different configurations of *CogniCrypt*$_{SUBS}$ allow developers to balance between speed and quality [CB16].

5.2.7 Limitations & Future Work

Our approach relies on the static analysis *CogniCrypt* and the *CrySL* ruleset. Thus, the precision and recall of our approach depend on the accuracy of *CogniCrypt*, the expressiveness of *CrySL*, and the used ruleset. While we tested the correctness of our implementation with test cases, we acknowledge that a more in-depth analysis

will be valuable to assess the accuracy. Further, our analysis focuses on the relevant area of cryptographic API misuses due to the well-maintained *CrySL* ruleset. Thus, analyses of further APIs are still missing. Future work could focus on other APIs and understand error chains for these APIs by leveraging our work. Specifically, due to the separation of the analysis logic and the rules, *only* new *CrySL* rules for the APIs of interest need to be specified. Thus, our work may be transferred to other domains.

Recruiting participants for expert interviews or user studies is challenging. For our expert interview (Section 5.2.5), we only managed to recruit five experts, and two of them may have heard about the initial high-level ideas of this research. Another limitation of the small number of participants is that all participants had the same task and tool order. Therefore, the findings may have a carryover effect between both tasks. We tried to minimize the effect through the design of the tasks, and yet the interview may have a bias. As a consequence, the interviews do not provide statistical insights. Further, for our interviews, we built a prototype that has not yet implemented a structured presentation for error chains in the way they occur in the wild. We acknowledge that implementing and evaluating a structured presentation and a statistically relevant user study is an interesting future research topic.

The phenomenon we observed in Figure 5.3 is influenced by the interconnected architecture of cryptographic APIs in Java. Thus, this observation may not generalize to domains in which API constraints are restricted to one class. Further, for other security-related API misuses, presenting the root error may be relevant, too. Consider a taint analysis that can detect SQL injections. While a SQL injection manifests at the vulnerable call to the database, the dangerous input is the root cause that may occur at a different location. As the analysis can reason about this input, it should be possible for the analysis to report subsequent errors.

5.2.8 Conclusion

Many applications contain error chains that were so far unstudied. To study these, we introduced *CogniCrypt*$_{SUBS}$, which utilizes an existing static analysis and extends its algorithm to identify dependencies between errors. Specifically, we can detect a root error and all its subsequent errors. With *CogniCrypt*$_{SUBS}$ we have shown that for cryptographic API misuses error chains occur frequently in the wild. We further find that dependencies between cryptographic API misuses can be expressed as trees. *CogniCrypt*$_{SUBS}$ has achieved a negligible median runtime overhead of 1.16%. Finally, we conducted an expert interview to evaluate qualitatively and show that a

structured presentation of root errors and dependent subsequent errors is helpful to fix misuses and improves the usability of the analysis. Overall, we show that error chains are relevant in real-world projects, and we provide a solution, *CogniCrypt*$_{SUBS}$, to detect those that can be used for APIs beyond cryptography. Hereafter, when we refer to *CogniCrypt*, this includes the adaptations introduced with *CogniCrypt*$_{SUBS}$ as later versions of *CogniCrypt* use them by default.

The results presented in this section show that adapting the analysis algorithm of a static analysis tool may be necessary to build a new usability feature. However, the presentation of error trees evaluated in the expert interviews was limited to a simple CLI version. As discussed in Chapter 3, visualizations in dedicated user interfaces typically perform better in supporting developers. In this thesis, we therefore address this in our final contribution, *SecAI* (*Contribution 5*, cf. Chapter 7), which uses the improved version of *CogniCrypt*. For error trees, we develop an interactive visualization feature (cf. Section 7.4.3) in *SecAI*.

In the next section, we introduce our third main contribution of this thesis, *CamBench* (*Contribution 3*, cf. Section 5.3), which we use to develop and evaluate prototypes: *FP*$_{Predictor}$ (cf. Section 7.4.8) for false positive prediction, and *LLM-SAST*$_{Repair}$ (*Contribution 4*, cf. Chapter 6) for automatically repairing cryptographic API misuses. Both are integrated into our last main contribution of this thesis, *SecAI* (*Contribution 5*, cf. Chapter 7), a new usability-focused user interface for *CogniCrypt*.

5.3 *CamBench* — Cryptographic API Misuse Detection Tool Benchmark Suite

In the previous sections, we first investigated cryptographic API misuses in open-source business applications and introduced a risk model to classify their severity in Section 5.1. Further, the study showed that warning reports, even though they are true positives, can be perceived as *effective false positives* by developers. To address the observation of error chains we made in the manual evaluation of the study and to increase the explainability of *Warning Messages* (cf. Section 3.3.1), we adapted the analysis of *CogniCrypt* to support the detection of error chains in Section 5.2. In an expert interview, we showed usability implications of representing error chains to aid developers in the remediation of cryptographic API misuses.

The results presented in these sections show that the static analysis tool and its capabilities directly impact how well developers can be supported. Therefore, in

this section, we propose the creation of a benchmark, *CamBench* (*Contribution 3*), "**C**ryptographic **A**PI **M**isuse Detection Tool **Bench**mark Suite", to evaluate detection tools for cryptographic API misuse detection in Java, because there is no established reference benchmark for a fair and comprehensive comparison and evaluation of these tools. While there are benchmarks, they often only address a subset of the domain or were only used to evaluate a subset of existing misuse detection tools.

CamBench is the third main contribution of this thesis and is also used in the evaluation of our fourth contribution, *LLM-SAST*$_{Repair}$ (*Contribution 4*, cf. Chapter 6) to automatically repair cryptographic API misuses and for the training of our model *FP*$_{Predictor}$ (cf. Section 7.4.8), which is integrated as a usability feature into our last main contribution of this thesis, *SecAI* (*Contribution 5*, cf. Chapter 7). In this section, we first present parts of the registered report accepted at the MSR 2022 Registered Reports Track as an In-Principal Acceptance (IPA). Then, we introduce *CamBench*$_{CAP}$ (*Contribution 3*), the implementation of the synthetic micro-benchmark to evaluate analysis capabilities and sensitivities, in Section 5.3.8.

5.3.1 Introduction

Cryptography is widely used in today's software to ensure the confidentiality of users' data. Concretely, through cryptography, everyone can securely use online banking, purchase a book from the local book shop from their computer, and share sensitive information with their co-workers while working remotely without the fear that the software leaks their data. Unfortunately, previous research results show that developers struggle with the secure usage of cryptographic APIs and often add vulnerabilities to their code [Nad+16; Gor+18; Laz+14; Wic+21]. Most of these vulnerabilities are caused by developers who use a cryptographic API in a way that is considered insecure by experts, e.g., choosing an outdated cryptographic hash algorithm like *SHA-1* [Laz+14]. We use the term cryptographic API misuse to describe these programming flaws. Further, we focus on Java cryptographic APIs to provide concrete results. To support developers and security auditors in identifying these misuses, many cryptographic API misuse detectors such as *CryptoRex* [Zha+19], *CryptoLint* [Ege+13], *CogniCrypt* [Krü+19], and *CryptoGuard* [Rah+19] have been developed.

While several tools demonstrate the practical importance and their capabilities to identify issues in real-world code, a fair comparison of the analysis capabilities is difficult. First, all in-the-wild studies were conducted on different applications and domains. For example, *CryptoLint* [Ege+13] was evaluated on 11,748 Android

apps in 2012 while *CogniCrypt* [Krü+19] analyzed 10,000 apps in 2017. While both tools demonstrated their capabilities for Android apps, it is hard to judge how they compare, as they analyzed different apps and the apps evolved over time. Further, other domains of Java software are only analyzed by one tool, e.g., Apache applications, as representatives for large Java projects [Rah+19], or Maven artifacts [Apab], representatives for Java libraries [Krü+19]. Thus, a fair comparison of these analyses and their capabilities is challenging.

An accepted standard in other fields to overcome the above-described problem is benchmarks. While it is time-consuming to create a proper benchmark, the advantages are significant. A benchmark can set a common understanding within the community of the capabilities of the respective solutions, e.g., the strength of a misuse detector. If a benchmark is used for several tools, which should be the aim, a benchmark enables a fair comparison of the respective solutions in the field. Further, a proper benchmark can drive improvements of each tool and form a community to improve the state of the art significantly. Overall, an established benchmark drives the development and research in the community of its domain [Bla+06].

Unfortunately, for the domain of cryptographic API misuse detection, there is still no standard benchmark. While we, as a community, have a few custom benchmarks [ARY19; Bra+17; Wic+19; MR17], all of them have significant limitations. Some benchmarks are developed for a specific problem, e.g., parametric-based misuses [Wic+19], and are not suitable outside of this problem space. Other benchmarks come with a significant bias, as their creation was along with the development of the respective detection tool [ARY19], e.g., test cases were created for the 16 threat model rules of the detection tool *CryptoGuard* [Rah+19]. Therefore, in the evaluation in comparison to other detection tools, only a subset of 6 common rules was used. In 2023, Torres et al. [Tor+23] corrected the ground truth of this benchmark [ARY19].

To solve the above-described problem, we aim to establish ***CamBench***, a fair "**C**ryptographic **A**PI **M**isuse Detection Tool **Bench**mark Suite" to compare cryptographic API misuse detectors. Thus, in this section, we analyze the state of the art of benchmarking cryptographic API misuse detection tools, and how these benchmarks adhere to proper methods for benchmarking. Further, we focus on understanding and deriving specific challenges that the domain of cryptographic API misuse detection tools poses to the design of benchmarks, e.g., the evolution of security characteristics. Based on this, we aim to create a fair benchmark to compare cryptographic API misuse detectors.

Our benchmark suite consists of two benchmarks and one heuristic to compare cryptographic API misuse detectors. The benchmarks are $CamBench_{REAL}$, a collection of real-world applications, and $CamBench_{CAP}$, a collection of synthetic test cases to evaluate the analysis capabilities. The API coverage heuristic, $CamBench_{COV}$, provides detailed information about the strengths and limitations of a misuse detector with respect to certain API classes and is used in the creation of $CamBench_{REAL}$ to ensure that all relevant JCA classes are represented. This property is important as an analysis that misses a class may miss a severe vulnerability in an application due to the respective class. Thus, an assessment of this property besides precision and recall is important.

The first version of *CamBench* focuses on usages of the standard cryptographic library in Java, the Java Cryptography Architecture (JCA), and static analyses. To support future extensions, e.g., on different APIs and dynamic analyses, one design principle of the benchmark is extensibility. We open-source all our research artifacts along with our tooling on GitHub[13]. This way, we enable the extension of *CamBench* and provide tooling to simplify the creation of future benchmarks in related domains. Further, our tooling and our theoretical work can reduce the effort to create fair benchmarks for arbitrary APIs, e.g., misuse of the *Iterator* API.

5.3.2 Benchmarks Covering Cryptography

To understand the differences, strengths, and weaknesses of the existing benchmarks for cryptographic API misuse detectors, we present the state of the art of benchmarks for cryptographic API misuse detection tools. We performed a literature survey and derived all discussed benchmarks by collecting all existing cryptographic benchmarks we were aware of and extended them through a forward- and backward search of the related publications. An overview of all identified benchmarks and misuse detectors that were successfully evaluated with them is presented in Table 5.3. Note that we did not list any analysis for *MuBench* as the API misuse detection tools used for evaluation did not find any cryptographic API misuse [Ama+18]. Similarly, we also omit the misuse detection tools that mark the secure and insecure test cases of the *Ghera Benchmark* as non-security critical (cf. Ranganath et al. [RM20]) for readability.

CryptoAPI-Bench [ARY19] is the most recent benchmark and consists of synthetic examples to identify misuses. In total, 171 examples were created along 16 different

[13] `https://github.com/CROSSINGTUD/CamBench/releases/tag/CamBench` (last accessed: 2026-06-06)

Table 5.3: An overview of all identified benchmarks and the cryptographic API misuse detectors analyzed ("*" marks non-academic static analyses.): A: *Amandroid* [WRO14], AC: *AppCritique* * [All], CC: *CogniCrypt* [Krü+19], CG: *CryptoGuard* [Rah+19], CO: *Coverity* * [Syn], DN: *DevKnox* * [XYSEC], FD: *FixDroid* [Ngu+17], FS: *FindSecBugs* * [Art20], J: *Julia* * [Spo16], MA: *Marvin-SA* * [RH], MF: *MobSF* * [Abr+], P: *PMD* * [PMD], SB: *SpotBugs* * [Spo], SQ: *SonarQube* * [Sonb], V: *VisualCodeGrepper* * [Vis], X: *Xanitizer* * [RIG], Y: *Yasca* * [SR], and Z: *OWASP ZAP* * [ZAP].
● denotes a tool was evaluated with a benchmark; other is denoted by ○.

Benchmark	A	AC	CC	CG	CO	DN	FD	FS	J	MA	MF	P	SB	SQ	V	X	Y	Z	Source
CryptoAPI-Bench [ARY19]	○	○	●	●	●	○	○	○	○	○	○	○	●	○	○	○	○	○	[ARY19]
Braga Benchmark [Bra+17]	○	○	○	○	○	○	○	●	○	○	○	○	○	●	●	●	●	○	[Bra+17]
Parametric Crypto Misuse Benchmark [Wic+19]	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	[Wic+19]
MuBench [Ama+16]	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	[Ama+16]
Ghera Benchmark [MR17]	●	●	○	○	○	●	●	○	○	●	●	○	○	○	○	○	○	○	[RM20]
OWASP Benchmark [Wic]	○	○	○	○	○	○	○	●	●	○	○	●	○	●	○	○	○	●	[BFS17]

vulnerabilities for basic (40) and advanced (131) cases for static analyses. The advanced tests focus on inter-procedurality, field-sensitivity, and path-sensitivity. Thus, it challenges the static analysis for several different analysis capabilities. However, the benchmark comprises the 16 vulnerability rules detected by *CryptoGuard* [Rah+19] — in Section 5.1.2, we adapted the risk model to cover 22 vulnerabilities. In 2023, Torres et al. [Tor+23] investigated benchmarks for cryptographic API misuse detection, too, and corrected the ground truth of *CryptoAPI-Bench* [ARY19].

While *CryptoAPI-Bench* [ARY19] focuses on analysis capabilities, the *Braga Benchmark* [Bra+17] focuses on providing developers with the best analysis tool for their team. A team is characterized by their experience (novice or knowledgeable) and their support for cryptographic tasks (unsupported or supported). In addition to the team's characteristics, the test cases of the benchmark are grouped along three complexity classes of cryptographic API misuses. The complexity is represented as the abstraction level required to identify and respectively fix the misuse. For example, identifying an outdated hashing algorithm is considered as low complexity, while the reuse of a nonce is marked as a high complexity problem caused by an insecure system design or architecture. Thus, this benchmark sheds light on different complexities to cover different teams' knowledge.

In contrast to the previous two benchmarks, the *Parametric Crypto Misuse Benchmark* [Wic+19] derived misuse cases from in-the-wild code. All misuses were collected from top Java GitHub projects and cover cryptographic misuses caused

by passing an insecure parameter to a function. Thus, it covers one of the major problems, incorrect configurations via parameters, observed in several in-the-wild studies [Krü+19; Ege+13] while ignoring more complex misuses, e.g., incorrect call order or misuses caused by architecture flaws. While the previously discussed benchmarks created the benchmarking infrastructure, the *Parametric Crypto Misuse Benchmark* extended an existing and well-established benchmark for API misuses, *MuBench* [Ama+16], which is a benchmark for general API misuses, including several cryptographic API misuses in Java.

In the publication from 2016, the benchmark includes 18 misuses manually identified from bug fixes on GitHub and SourceForge. All of these cases, except one fix, influence the behavior of the program, such as a crash due to invalid input. Thus, the misuses cover spurious behavior rather than insecure algorithm choices. Later, the benchmark was extended with an additional 31 misuses by mining projects via *BOA* [Dye+13], which use the *javax.crypto.Cipher* or the entire *javax.crypto* package, and manually verifying the security of the code [Ama+18].

The *Ghera Benchmark* [MR17] focuses on general vulnerabilities in Android applications, which are a subset of Java applications, and thus covers a few cryptographic API misuse cases as well. In total, this benchmark includes five instances of cryptographic vulnerabilities and eight vulnerabilities caused by networking. While it may be counter-intuitive, the latter eight vulnerabilities may be covered by cryptographic API misuse detectors as well when covering SSL and TLS-related misuses, e.g., the usage of TLS 1.0. While the number is low, all cases are executable Android applications with a vulnerability along with an exploit of the respective vulnerability.

While the previous benchmarks are developed in academia, the *OWASP Benchmark* [Wic] is developed by industry. Similar to our motivation, the underlying motivation for the *OWASP Benchmark* was to measure the strengths and weaknesses of different static analyses to detect vulnerabilities such as cryptographic API misuses. To achieve this aim, version 1.2 consists, similarly to *Ghera Benchmark*, of test cases with real-world web applications that can be exploited. The exploits are categorized along with the different vulnerability types of OWASP Top-10 web vulnerabilities [Sto+21], such as command injection and weak cryptography. In total, 246 tests for vulnerabilities are in the *OWASP Benchmark*.

The overlap of the static analyses, e.g., *CogniCrypt* [Krü+19], *CryptoGuard* [Rah+19], and *FindSecBugs* [Art20], analyzed on the aforementioned benchmarks is rather small, as the motivation and domain for which all these benchmarks were created differ. We illustrate the benchmarked static analyses for each previously discussed

benchmark in Table 5.3. Only two tools, namely *FindSecBugs* (three) and *SonarQube* (two), were evaluated on more than one benchmark. The remaining 16 analyses were evaluated only on one benchmark. For example, the *Ghera Benchmark* evaluated analyses for Android vulnerabilities, such as *Amandroid* [WRO14]. On the other hand, tools focused on cryptographic API misuses, such as *CryptoGuard* [Rah+19], are evaluated on the cryptography-specific *CryptoAPI-Bench*.

5.3.3 Methodology

We presented existing benchmarks for Java cryptographic API misuse detection tools in Section 5.3.2. In this section, we discuss the theoretical foundations we derived from our literature survey for our benchmark generation methodology and design. As of now, our results indicate that there is no widely accepted or established benchmark for the evaluation of cryptographic API misuse detection tools. Existing benchmarks have only been used to compare subsets of cryptographic API misuse detection tools (cf. Table 5.3).

First, we discuss what the proper methodology for benchmark generation in general is, and how this can be applied to one domain, namely cryptographic API misuse detection tool evaluation in Section 5.3.3. Second, we analyze the special characteristics of this domain that need to be addressed by *CamBench* or, in general, a benchmark for this domain in Section 5.3.3. We conclude the theoretical foundations by deriving the domain-specific requirements for *CamBench* and developing the domain-specific methodology for the generation of benchmarks in Section 5.3.3. Then we use both to formulate the practical application in our execution plan for the generation of *CamBench*.

Literature review on benchmark generation approaches

The creation of a benchmark for a specific domain requires first an understanding of the characteristics of well-established more general benchmarks in the community. Our literature review to establish a benchmark for cryptographic API misuse detection tools has yielded a few relevant approaches from more general benchmarks. For our approach, two findings stand out, namely the *DaCapo benchmarks* [Bla+06] and the Automated Benchmark Management System (*ABM*) [DEB16]. The *DaCapo benchmarks* are a collection of widely used open-source benchmarks for the Java programming language introduced in 2006 and with its latest release in 2018. In their report on the development and establishment of the *DaCapo benchmarks*, Blackburn

et al. stressed the importance of an open and transparent process with community feedback. Their success is a compelling argument to follow transparency and community feedback as guiding principles when developing *CamBench*. Similarly, the *Qualitas Corpus* [Tem+10] also provides a large collection of real-world open-source Java code. The way the real-world applications are organized and enriched with metadata, also for versioning, is valuable information for the design of benchmarks regarding extensibility. However, the *Qualitas Corpus* was introduced in 2010 and curated, but with its latest update in 2013, it appears to be unmaintained.

The *DaCapo benchmarks* and *Qualitas Corpus* are both benchmarks for Java in general, targeting the evaluation of the performance of Java code and Java virtual machines. However, evaluating tools that detect misuses of cryptographic APIs is a different task. The main focus here rather lies on the correctness of the results, i.e., precision and recall of the detected issues and the soundness of the analysis itself. Moreover, how usable such a cryptographic API misuse detection tool is also relies on its precision and recall [CB16] – too many false positives are one of the main reasons why developers omit using static analysis tools. Such domain-specific requirements can be better addressed by the approach of Nguyen Quang Do et al. called *ABM* [DEB16] since domain-specific datasets are needed to evaluate specific research questions. They describe how to semi-automatically generate and maintain domain-specific benchmark suites. The benchmark suites are collected with a well-described process of crawling buildable open-source projects of the target domain. Therefore, they provide a representative set of real-world applications relevant in the domain. We derive our methodology from the experience that can be taken from the *DaCapo benchmarks* and the proposed procedure by *ABM* and provide a more in-depth explanation of the execution in Section 5.3.4. Moreover, *ABM* already describes that filtering the real-world applications for relevance is integral to being representative. Hence, understanding the requirements of a domain is key to generating a representative benchmark suite for it.

Analysis of the domain for benchmarking cryptographic API misuse detection tools

Section 5.3.2 presents preliminary findings of benchmarks targeted for cryptographic API misuse detectors. The goal that cryptographic API misuse detection tools aim to achieve is detecting and reporting cryptographic API misuses in applications that use cryptographic APIs. Employing cryptography in general sets the focus on security. Therefore, when comparing misuse detection tools, it is important to evaluate their performance in terms of cryptographic API coverage, precision and recall, and

analysis capabilities. The coverage is important for developers to know whether their code using a cryptographic API is completely or only partially covered, e.g., common known vulnerabilities. Research on reasons why misuse detection tools are rarely used has shown that precision and recall of a tool play a key role [CB16] – too many false positives are a major reason for developers to omit using a tool [Joh+13]. Moreover, the evaluation of analysis capabilities is important as well. Whether an analysis is intraprocedural, inter-procedural, flow-sensitive, context-sensitive, field-sensitive, path-sensitive, and object-sensitive or not has direct implications on what the analysis is actually able to detect. Unfortunately, the benchmarks identified in Section 5.3.2 either ignore such properties or only include a subset, such as *CryptoAPI-Bench* [ARY19].

Domain-specific Benchmark Requirements

Concluding the theoretical foundations of our methodology, we specify the requirements specific to the domain of benchmarking cryptographic API misuse detection tools and propose our benchmark design based on the results of the previous sections. Following the takeaways from the *DaCapo benchmarks* [Bla+06] (transparent and open process, a variety of real-world applications maximizing the domain coverage, easy to use, and providing metrics), we design *CamBench* as a benchmark suite consisting of several benchmarks. A good example of easy-to-use is the well-usable deployment system of *MuBench* [Ama+16], which delivers its execution pipeline as a Docker container and provides extensive documentation on how to use and adapt the benchmark as well as on how to contribute. Moreover, we enrich the collection process of real-world applications with the semi-automated approach of *ABM* [DEB16].

Based on these presented approaches, we derive the following requirements that *CamBench* must meet:

1. The benchmark generation process needs to be transparent and consider community feedback.
2. The benchmark contains a representative and diverse collection of real-world applications using cryptographic APIs. Projects fulfill these properties: *open-source*, *compilable (source code and binaries are available)*, and *representative of the domain's real-world applications, e.g., diverse size, context, ...*.
3. The benchmark is open-sourced.
4. The benchmark is provided with a deployment system.

CamBench is designed as a suite of benchmarks covering the following three main aspects: *(a)* $CamBench_{REAL}$ with *real-world applications with cryptographic API usage* to evaluate the performance of misuse detection tools on relevant applications [DEB16], *(b)* $CamBench_{CAP}$ with *analysis capabilities test cases*, and *(c)* $CamBench_{COV}$, a heuristic for *cryptographic API coverage* to evaluate which fraction of the API a misuse detection tool supports. We added $CamBench_{CAP}$ and $CamBench_{COV}$ to accommodate the requirements of the domain of benchmarking cryptographic API misuse detection tools. Moreover, in the first version of *CamBench*, we will support specifically the Java Cryptography Architecture (JCA) as it is the most used API.

To facilitate the evaluation of analysis capabilities, we develop synthetic test cases for all relevant properties in $CamBench_{CAP}$. When building an analysis tool, it is necessary to make assumptions that can lead to unsoundness [Rep00; Liv+15]. Therefore, evaluating the analysis capabilities which are part of such assumptions is important to draw implications concerning a tool's unsoundness and precision. The properties we include in $CamBench_{CAP}$ are *flow-sensitivity*, *context-sensitivity*, *field-sensitivity*, *object-sensitivity*, and *path-sensitivity*.

CamBench provides $CamBench_{COV}$, a heuristic for testing the coverage of Java cryptographic APIs. Similar to *ABM*'s [DEB16] focus on automation, our goal is to develop a process with a high degree of automation for the curation of the benchmarks for the coverage of the JCA. This will help in maintaining and updating the benchmark in the future.

Further, test cases are enriched with metadata similarly to *MuBench* [Ama+16], where misuse code examples are specified with YAML files enriched by additional information like misuse type and description. Like them, we also provide examples for correct usages. Moreover, the metadata (cf. Listing 5.3) comprises information on whether a test case is a (in-)secure cryptographic usage (whether a cryptographic API usage is (in-)secure might also be context-dependent, e.g., using a hash algorithm like MD5 is considered insecure [Kas+13], yet using MD5 in the context of file validation [AWS] might still be acceptable), usage category [Ama+18], and the severity. Moreover, we will consider metadata on versioning of real-world projects as proposed in the *Qualitas Corpus* [Tem+10].

To summarize, *CamBench* will provide $CamBench_{REAL}$, a collection of *real-world applications* with cryptographic API usage following *ABM* [DEB16], $CamBench_{CAP}$, a collection of synthetic tests to evaluate the cryptographic API misuse detectors' *analysis capabilities*, and $CamBench_{COV}$, a heuristic for checking the *cryptographic API coverage* of the JCA.

5.3.4 Execution Plan

In this section, we first describe the generation of *CamBench* and its benchmarks in Section 5.3.4, namely $CamBench_{REAL}$ for *real-world applications*, $CamBench_{CAP}$ for *analysis capabilities*, and $CamBench_{COV}$, the heuristic for *cryptographic API coverage*. Then we will discuss the publishing and deployment process of *CamBench* in Section 5.3.5.

Curation Plan

For the curation of *CamBench*'s two benchmarks, we described our methodology in Section 5.3.3. For the concrete implementation of our methodology for the first version of *CamBench*, we set the scope to the standard cryptography library in Java, the JCA, and static analysis tools detecting misuses of it.

Benchmark for real-world applications – $CamBench_{REAL}$ For the collection and creation of the real-world application benchmark, we adapt the process specified by *ABM* [DEB16]. This process consists of six steps of collecting, filtering, and building open-source projects. The first three steps are *(i) mining open-source projects*, *(ii)* filtering for *active projects*, and *(iii)* filtering for *known build systems*. These steps are automated. The subsequent steps require manual effort. In their instantiation of *ABM* for the domain of Java business web applications, Nguyen Quang Do et al. chose GitHub [Gitc] as a platform for mining open-source projects [DEB16]. Since this has worked successfully and GitHub is a popular and well-used platform for hosting open-source projects, we use it, too. The next step is *(iv)* downloading and *building the projects*. The real-world application benchmark only includes projects that build with standard build systems. Another filtering step follows where, in our case, we *(v)* check whether the *JCA is used* in the projects. The last step is *(vi) compiling the binaries* and adding projects with binaries to the benchmark dataset (cf. requirements in Section 5.3.3).

Furthermore, we add a step to the approach of *ABM* [DEB16] by adding metadata files including information on whether the usage is (in-)secure for selected usages of the JCA. To automatically identify usages of the JCA in code, we use a simple static analysis implemented using *OPAL* [Hel+20]. For the ground truth, whether a JCA usage is secure or insecure, we use the guidelines of NIST [NISa; NISb], SOG-IS [SOG], or BSI [BSI] for manual labeling. Since the guidelines of the BSI are updated regularly and are the most up-to-date, we primarily use them. Furthermore,

the employed guidelines can be documented in the metadata files. The design of the metadata files is derived from *MuBench* [Ama+16] and takes the experiences concerning versioning from *Qualitas Corpus* [Tem+10] into account. The dataset of the real-world applications will contain too many usages of the JCA for the manual creation to be feasible. Therefore, we will select a representative subset of the JCA usages to write the metadata files.

Benchmark for analysis capabilities – *CamBench$_{CAP}$* The benchmark for evaluating analysis capabilities consists of synthetic test cases specifically designed to test relevant analysis properties for common cryptographic API misuses. The test cases for the analysis capabilities require manual effort from domain experts. They will include test cases for *flow-sensitivity*, *context-sensitivity*, *field-sensitivity*, *object-sensitivity*, and *path-sensitivity*, as well as *intraprocedural* and *inter-procedural* test cases. The selected sensitivities are accepted as relevant capabilities for static analyses [Li+17; QWR18]. All test cases for this benchmark provide the metadata files for cryptographic API (mis-)uses.

Heuristic for cryptographic API coverage – *CamBench$_{COV}$* During our literature review, we identified that understanding the covered API classes is of importance for crypto-API misuse detectors. A detector that misses an API class may miss a vulnerability in an application while having a high precision and recall. Thus, for the assessment of the detectors, being aware of the covered classes is of importance. While test generators seem to be a good choice for this task and aid our aim of maintainability and extensibility of *CamBench*, they are inadequate. Previous approaches summarized in a survey [MWI17] have been shown to be insufficient for generating and establishing a ground truth needed for *CamBench*. Instead of using test generators, we propose employing a heuristic that harnesses the two other benchmarks of *CamBench* to create an API coverage feature.

This feature is inspired by *Hermes* [Rei+17], which is a framework to assess collections like the *Qualitas Corpus* and uses queries to derive subsets suited for analyzing, e.g., a specific API. With this heuristic, we can extract all classes and methods of the JCA and match them with our benchmarks for *real-world applications* and *analysis capabilities*. We use the heuristic to guide the effective generation of *CamBench$_{REAL}$*, as the heuristic enables us to select a diverse set of real-world JCA usages to enrich them with the metadata. Overall, we assume that this heuristic approach should yield an almost complete coverage of JCA class and method usages for a highly used API like the JCA.

5.3.5 Publishing and Deployment Plan

Open access and ease of use are key factors for establishing a benchmark as described in Section 5.3.3. Hence, it is important to publish *CamBench* on GitHub[14] with proper instructions and documentation. To encourage, include, and harness community feedback, we make the GitHub repository accessible during the creation of *CamBench*, e.g., via pull requests. We will also contact the authors of existing cryptographic API misuse detection tools and benchmarks (cf. Table 5.3).

5.3.6 Threats to Validity

For the first version of *CamBench*, we introduced some limitations (cf. Section 5.3.1) to have a clear target domain and develop a tangible benchmark. These limitations are choosing a specific programming language in Java, a specific API in the JCA, and finally focusing on static analysis tools. However, our methodology (cf. Section 5.3.3) is designed for semi-automated benchmark generation and extensibility. Hence, extending *CamBench* to support other APIs or dynamic analyses has been taken into consideration during the initial development as a reasonable future step and is, as a result, not inhibited whatsoever by the design. Furthermore, instantiating benchmarks for other programming languages by applying our methodology is also possible. However, these extensions would each require a lot of additional effort and leave a benchmark that is significantly changed in shape and size, as the new real-world applications need to be mined with new criteria, and thus, the benchmark for analysis capabilities and API coverage heuristic need adaptation, too.

Besides this, the generation of the benchmark for *analysis capabilities* and *API coverage heuristic* is subject to manual work by the authors. Since the authors are experts regarding static analysis and cryptographic API misuse detection, the chosen domain limitations make them an appropriate choice to develop and label the test cases. Furthermore, community feedback was part of the generation process and therefore helps to mitigate this threat to validity.

5.3.7 Implications

Benchmarks generally drive development and research in their fields as they provide measurable optimization goals [Bla+06]. Therefore, establishing a reference bench-

[14] `https://github.com/CROSSINGTUD/CamBench/releases/tag/CamBench` (last accessed: 2026-06-06)

mark for the domain of benchmarking cryptographic API misuse detection tools in the form of *CamBench* with its open and transparent generation process will help the community in many ways: *CamBench* will allow us to evaluate and compare current cryptographic API misuse detection tools on the same benchmark, which has not been done yet. Having this comparison, future development can reference the comparison as well as *CamBench* in new contributions.

Furthermore, by deriving, extending, and defining the process for the creation of *CamBench*, we hope to provide a template for creating and maintaining benchmarks for specific domains in need of a reference benchmark and thereby steering current developments towards more open and reproducible research.

5.3.8 Implementation of $CamBench_{CAP}$

In this section, we describe the implementation of *CamBench* after the registered report for its creation was published. The implementation of $CamBench_{REAL}$ and $CamBench_{COV}$ is published in the PhD thesis of Anna-Katharina Wickert [Wic25]. In this section, we focus on the curation of $CamBench_{CAP}$ as a main contribution (*Contribution 3*) of this thesis.

Following the execution plan described in Section 5.3.4, we developed the *benchmark for analysis capabilities*, called $CamBench_{CAP}$. Similarly to *MuBench* [Ama+16], for *CamBench* we chose to provide the ground truth with metadata in the form of a YAML file per test case. For classification of the type of misuses, we used *FUM* [Sch+22a] (cf. Chapter 4). For the capabilities and sensitivities of static analysis tools [Li+17; QWR18], we followed a systematic approach and first reviewed the literature and identified the following as relevant: *flow-sensitivity*, *context-sensitivity*, *field-sensitivity*, *object-sensitivity*, and *path-sensitivity*, as well as *intraprocedural* and *interprocedural* cases. Further, we identified the most common cryptographic API misuse patterns from previous studies [Ege+13; Shu+14; Wic+19; HGN20; Wic+22; Zha+22]: *BrokenCrypto* (insecure encryption), *BrokenHash* (insecure hashing), *ECBMode* (insecure default mode in encryption), *InsecureRandom* (use of insecure randomness), *PBEParameters* (too small iteration count), *SmallKeySize* (too small key sizes, e.g., < 2048 for RSA), and *StaticIV* (use of static values for initialization vector).

Metadata File We derived our metadata format based on the work of Amann et al. in *MuBench* [Ama+16]. For the metadata files, we use machine-readable and

structured YAML files to support automation in the use of *CamBench*. For each case in $CamBench_{CAP}$, one metadata file was created. We defined the format of the metadata as JSON-schema[15] and provide an example in Listing 5.3. The metadata file is designed to be informative and flexible, but still needs to hold key information. Therefore, fields such as `api` in Line 67 specifying the misused cryptographic class are mandatory, as they provide pivotal information. Other fields, such as `name` in Line 63 specifying the name of the case, hold valuable information, but with respect to flexibility, e.g., for adaptability in future versions, are optional.

Important information about the cryptographic API usage is specified in the mandatory `crypto-usage` in Line 75; whether the case is a secure or insecure case is specified in `violation` in Line 78 with a boolean value. The type of API usage constraint [Sch+22a] (cf. Section 4.4) (and misuse if the `violation` field is `true`) is specified using *FUM* [Sch+22a] (cf. Section 4.5) with the `FUM` field in Line 81. Moreover, we include optional fields, e.g., `root` in Line 99, to specify concepts, such as error chains [Wic+24] (cf. Section 5.2), which may not be widely adopted at the moment. However, including such information may drive the development of such features in static analysis tools in the future. These optional fields containing further information include `fix` information (cf. Line 116) referencing a secure version of the case, `codemetrics` in Line 128 such as McCabe's cyclomatic complexity [McC76] in Line 134, and analysis `capabilities` or sensitivities in Line 138 assessed in the case. Providing additional information such as the `fix` information (cf. Line 116) can potentially help in building better features for *Fix Support* (cf. Section 3.3.2).

Listing 5.3: Structure of the metadata file used for CamBench (cf. `https://github.com/CROSSINGTUD/CamBench/blob/main/CamBench_Cap/src/metadata/metadata.yaml`).

```
61  #[optional]
62  #The full name of the test case.
63  name: Broken Hash Algorithm Case 1
64
65  #[required]
66  #The api that is (mis)used in the test case.
67  api: java.security.MessageDigest
68
69  #[optional]
70  #A description of the test case.
71  description: A broken hash algorithm (MD2) is initialized to compute the hash value of some
          data.
72
73  #[required]
74  #Collection of mappings that define the (mis-)usage of cryptography in the test case.
75  crypto-usage:
76    #[required]
```

[15] `https://github.com/CROSSINGTUD/CamBench/blob/main/CamBench_Cap/src/metadata/cambench_metadata_schema.json` (last accessed: 2026-06-06)

```yaml
#Specifies, if the test case contains a cryptographic misuse.
violation: true
#[required]
#Sequence of FUM based API misuse classes to categorize the API (mis-)use.
FUM:
  - argument_state/string_format
#[required]
#Location of the (mis-)use of the cryptographic API; specified by a path to a file,
#the containing method signature and the line of code.
location:
  #[required]
  #Path to the codefile of the test case.
  file: main/java/org/cambench/cap/basic/brokenhash/misuse.java
  #[required]
  #The method signature of the method that contains the cryptographic API (mis-)use.
  method: main(String[] args)
  #[required]
  #The linenumber of the cryptographic API (mis-)use in the codefile.
  line: 20

  #[optional]
  #Contains the location of the root of a cryptographic misuse.
  root:
    #[required]
    #Location of the root cause of the cryptographic API misuse; specified by a path to a file,
    #the containing method signature and the line of code.
    location:
    #[required]
    #Path to the codefile that contains the root of a cryptographic API misuse.
    file: main/java/org/cambench/cap/basic/brokenhash/root.java
    #[required]
    #The method signature of the method that contains the root of a cryptographic API misuse.
    method: initVariable()
    #[required]
    #The linenumber of the root of a cryptographic API (mis-)use in the codefile.
    line: 10

#[optional]
#Contains a description on how to fix the misuse as well as an optional file and commit id to an implemented fix.
fix:
  #[required]
  description: A description how to fix a misuse.
  #[optional]
  #Path to the codefile that implements a fixed version of the test case.
  file: main/java/org/cambench/cap/basic/brokenhash/fixed.java
  #[optional]
  #The commit that fixes a cryptographic misuse.
  commit: github.com/path/to/commit

#[optional]
#Codemetrics of the test case.
codemetrics:
  #[optional]
  #Lines of Code
  loc: 25
```

```
132     #[optional]
133     #McCabes Cyclomatic Complexity
134     cc: 1
135
136   #[optional]
137   #Lists all the analysis capabilities that are covered by the test case.
138   capabilities:
139     - interprocedural
140
141   #[optional]
142   security:
143     #[required]
144     #States, whether the testcase contains a security relevant misuse or not.
145     issue: true
146     #[optional]
147     #A field to add a description/additional information about the issue(s) contained in the
          test case.
148     issue-description: A description of the issue(s) contained in the test case.
```

Basic test case patterns The test cases of *CamBench*$_{CAP}$ have been designed and structured systematically. First, we provide basic pattern cases, one for each analysis capability or sensitivity: *flow-sensitivity*, *context-sensitivity*, *field-sensitivity*, *object-sensitivity*, and *path-sensitivity*, as well as *inter-procedural* with depth 2 and 3. For each capability and sensitivity, we provide a simple example case without cryptographic API usage for which a static analysis needs the respective capability to reason about. These example cases are supposed to show the patterns used for the capabilities and sensitivities in the more complex cryptographic test cases. For instance, Listing 5.4 shows an example for *field-sensitivity*; `Class1` in Line 159 has two fields, `value1` in Line 160 and `value2` in Line 161. A secret is stored in `value1` in Line 153 and printed in Line 156. A field-sensitive analysis is able to distinguish between fields and can decide whether a field contains sensitive data.

Next, we include basic mixed patterns combining two analysis capabilities or sensitivities, e.g., *field-sensitivity* and *flow-sensitivity* as shown in Listing 5.5. Compared to Listing 5.4, this case has one additional assignment in Line 172 where `value1` is assigned with a non-secret value. This assignment happens after the secret information in `value1` was already printed. To properly reason on whether a secret value was leaked, an analysis additionally needs to be *flow-sensitive* (and *field-sensitive*) as it then can reason on which String is assigned to `value1` in Line 170.

Lastly, we include basic cryptographic API misuse patterns for the most common cryptographic API misuse patterns from previous studies [Ege+13; Shu+14; Wic+19; HGN20; Wic+22; Zha+22]: *BrokenCrypto*, *BrokenHash*, *ECBMode*, *InsecureRandom*, *PBEParameters*, *SmallKeySize*, and *StaticIV*. An example of *BrokenCrypto* is shown

Listing 5.4: Basic pattern case for *field-sensitivity* (cf. `https://github.com/CROSSINGTUD/CamBench/blob/main/CamBench_Cap/src/main/java/org/cambench/cap/patterns/pure/FieldSensitivity.java`).

```
149 public class FieldSensitivity {
150
151   public static void main(String[] args) {
152     Class1 classObject = new Class1();
153     classObject.value1 = "secret value";
154     classObject.value2 = "non secret value";
155
156     System.out.println(classObject.value1);
157   }
158
159   public static class Class1 {
160     String value1;
161     String value2;
162   }
163 }
```

Listing 5.5: Basic mixed pattern case for *field-sensitivity* and *flow-sensitivity* (cf. `https://github.com/CROSSINGTUD/CamBench/blob/main/CamBench_Cap/src/main/java/org/cambench/cap/patterns/mixed/FieldFlowSensitivity.java`).

```
164 public class FieldFlowSensitivity {
165   public static void main(String[] args) {
166     Class1 classObject = new Class1();
167     classObject.value1 = "secret value";
168     classObject.value2 = "non secret value";
169
170     System.out.println(classObject.value1);
171
172     classObject.value1 = "non secret value";
173   }
174
175   public static class Class1 {
176     String value1;
177     String value2;
178   }
179 }
```

in Listing 5.6, where the insecure algorithm `DES` is used for encryption in Lines 182 and 183. Since an analysis tool correctly could report the insecure use of *DES* in two different locations (cf. Lines 182 and 183), this case, *BrokenCrypto1*, received two ground truth metadata files – one per location.

For encryption, the JCA provides many algorithms that can be used syntactically correctly, but are insecure, such as *DES*. Therefore, we included structurally the same cases with other algorithm configurations: `Blowfish`, `RC2`, `RC4`, and `RC5`. All cases are fixed in one `CorrectedCrypto`-file by using the secure algorithm `AES`. Overall, for *BrokenCrypto*, there are 5 basic pattern cases that are insecure (*true positive* if an analysis tool reports them) and one secure case (*false positive*). Similarly, the other patterns may also have multiple cases as shown in Table 5.4.

Listing 5.6: Basic cryptographic API misuse pattern case for *BrokenCrypto* (cf. `https://github.com/CROSSINGTUD/CamBench/blob/main/CamBench_Cap/src/main/java/org/cambench/cap/basic/brokencrypto/BrokenCrypto1.java`).

```
180 public class BrokenCrypto1 {
181   public static void main(String[] args) throws NoSuchPaddingException,
        NoSuchAlgorithmException, InvalidKeyException {
182     Cipher cipher = Cipher.getInstance("DES");
183     KeyGenerator keyGen = KeyGenerator.getInstance("DES");
184     cipher.init(Cipher.ENCRYPT_MODE,keyGen.generateKey());
185   }
186 }
```

Benchmark test case structure *CamBench*$_{CAP}$ systematically provides test cases by combining basic patterns for common cryptographic API misuses (cf. Listing 5.6) with patterns for analysis capabilities and sensitivities (cf. Listings 5.4 and 5.5). The structure and number of corresponding test case files are shown in Table 5.4; the top part represents basic patterns, the middle part shows the systematic combinations of cryptographic patterns with individual analysis capabilities and sensitivities, and the bottom part provides systematic combinations of cryptographic patterns with two *mixed* (combined) analysis capabilities and sensitivities. For the combination of cryptographic patterns with individual analysis capabilities and sensitivities, we provide different levels of complexity, e.g., for *field-sensitivity* (cf. Listing 5.4) the basic cases have a field-depth of one; for higher complexity, which might be relevant in practice, we also provide cases for a field-depth of two and three.

Listing 5.7: Mixed *field-sensitivity* and *flow-sensitivity* case for the cryptographic API misuse pattern *BrokenCrypto* (cf. `https://github.com/CROSSINGTUD/CamBench/blob/main/CamBench_Cap/src/main/java/org/cambench/cap/mixedsensitivities/fieldflow/truepositive/brokencrypto/BrokenCrypto1.java`).

```
187   public class BrokenCrypto1 {
188     public static void main(String[] args) throws NoSuchPaddingException,
        NoSuchAlgorithmException, InvalidKeyException {
189       CryptoClass1 cryptoClass = new CryptoClass1();
190       cryptoClass.cipher1 = "DES";
191       cryptoClass.cipher2 = "Blowfish";
192       cryptoClass.cipher2 = "AES/GCM/NoPadding";
193
194       Cipher cipher = Cipher.getInstance(cryptoClass.cipher1);
195       KeyGenerator keyGen = KeyGenerator.getInstance(cryptoClass.cipher1);
196       cipher.init(Cipher.ENCRYPT_MODE,keyGen.generateKey());
197     }
198
199     public static class CryptoClass1{
200       public String cipher1;
201       public String cipher2;
202     }
203   }
```

The bottom part of Table 5.4 shows the most complex part of *CamBench$_{CAP}$*, where two analysis capabilities and sensitivities are mixed, e.g., as seen for *field-sensitivity* and *flow-sensitivity* in Listing 5.5, and combined with the common cryptographic API misuse patterns such as *BrokenCrypto* in Listing 5.6. The combination of the previous examples is shown in Listing 5.7; *field-sensitivity* is needed to differentiate between the two fields (cf. Lines 200 and 201) of `CryptoClass1` (cf. Line 199), and *flow-sensitivity* is required to decide which value is assigned to the field `cryptoClass.cipher2` (cf. assignments in Lines 191 and 192) when the respective `getInstance(_)`-method is called on `Cipher` in Line 194 and `KeyGenerator` in Line 195, and the analysis needs to be able to detect the cryptographic API misuse in general.

While a single case of these *mixed-sensitivity* cases may be detected correctly by a static analysis tool for cryptographic API misuse detection that lacks the respective sensitivity, detecting all correctly is very unlikely. Further, *CamBench$_{CAP}$* contains a secure (*false positive*) case for each insecure (*true positive*) case as well as a fixed case (`Corrected...`). Therefore, *CamBench$_{CAP}$* overall provides more *false positive* cases (404) than *true positive* cases (219).

5.3.9 Conclusion

In this chapter, we presented two publications studying cryptographic API misuses (cf. Sections 5.1 and 5.2) that we contributed to as co-authors, which are fundamental for the further main contributions of this thesis, including the third main contribution of our thesis: *CamBench* (*Contribution 3*, cf. Section 5.3) and its implementation *CamBench$_{CAP}$* (cf. Section 5.3.8). The development of *CamBench* was driven by the experience and knowledge gained from the investigation of cryptographic API misuses in business applications (cf. Section 5.1) and error chains in real-world projects (cf. Section 5.2). Further, we used *FUM* (*Contribution 2*, cf. Chapter 4) to classify API usage constraints and misuses in metadata files of *CamBench*. Since benchmarks can drive future development, e.g., of features of static analysis tools, we included information such as *fix information* in the metadata files to, e.g., increase *Fix Support* (cf. Section 3.3.2), which is the feature least supported by static analysis tools, as shown in Chapter 3 (*Contribution 1*).

CamBench$_{CAP}$, hereafter referred to as *CamBench*, provides a systematic benchmark to assess the capabilities and supported sensitivities of static analysis tools for cryptographic API misuse detection. It consists of 219 insecure and 404 secure cases of Java cryptographic API usage in different levels of complexity; each case

Table 5.4: Structure and number of cases of *CamBench*$_{CAP}$ (cf. `https://github.com/CROSSINGTUD/CamBench/tree/main/CamBench_Cap/src/main/java/org/cambench/cap`).

Cryptographic API misuse type		**# true positives**	**# false positives**	**Sum**
BrokenCrypto		5	1	6
BrokenHash		4	1	5
ECBMode		1	1	2
InsecureRandom		2	1	3
PBEParameters		2	1	3
SmallKeySize		2	1	3
StaticIV		2	1	3
Capability & Sensitivity Patterns		-	22	22
	Subtotal	**18**	**29**	**47**
Capability & Sensitivity	**Complexity**	**# true positives**	**# false positives**	**Sum**
Context-sensitivity	basic	8	15	23
	callsites5	5	10	15
Field-sensitivity	basic	8	15	23
	field-depth2	5	10	15
	field-depth3	5	10	15
Flow-sensitivity	basic	8	15	23
	valueswap	5	10	15
Interprocedural	basic-depth2	8	15	23
	basic-depth3	8	15	23
Object-sensitivity	basic	8	15	23
	multipleobjects	5	10	15
Path-sensitivity	basic	8	15	23
	path-depth2	5	10	15
	path-depth3	5	10	15
	Subtotal	**91**	**175**	**266**
Mixed Capability/Sensitivity		**# true positives**	**# false positives**	**Sum**
Context-field		6	12	18
Context-flow		6	12	18
Context-object		6	12	18
Context-path		10	16	26
Field-flow		10	16	26
Field-object		6	12	18
Field-path		10	16	26
Flow-object		6	12	18
Flow-path		6	12	18
Object-path		10	16	26
Interprocedural2-context		6	12	18
Interprocedural2-field		6	12	18
Interprocedural2-flow		10	16	26
Interprocedural2-object		6	12	18
Interprocedural2-path		6	12	18
	Subtotal	**110**	**200**	**310**
	Total	**219**	**404**	**623**

has a metadata file with the specified ground truth. In the context of this thesis, *CamBench* is used to train $FP_{Predictor}$ (cf. Section 7.4.8) to predict false positives in the

warning reports of *CogniCrypt*; $FP_{Predictor}$ is then used as a feature of our final main contribution, *SecAI* (*Contribution 5*, cf. Chapter 7). *SecAI* also makes use of the threat model introduced in Section 5.1 for its *Severity Score* feature (cf. Section 7.4.7) and employs the improved analysis of *CogniCrypt* (cf. Section 5.2), which also allows us to provide a visualization of error trees (cf. Section 7.4.3). Moreover, *CamBench* is used to evaluate to what extent our fourth main contribution, $LLM\text{-}SAST_{Repair}$ (*Contribution 4*, cf. Chapter 6), can be used to automatically repair cryptographic API misuses. This will be discussed in the following chapter.

6 *LLM-SAST$_{Repair}$* — Can SAST Tools Support LLMs in Automatically Repairing Cryptographic API Misuses?

In this chapter, we present our fourth main contribution of this thesis, *LLM-SAST$_{Repair}$* (*Contribution 4*), to support developers in reducing Java cryptographic API misuses by automatically repairing them. In our study of usability criteria addressed by static analysis tools (*Contribution 1*, cf. Chapter 3), we identified the usability category of *Fix Support* (cf. Section 3.3.2) as the least supported. We aim to address this by proposing *LLM-SAST$_{Repair}$*, an automated repair system that leverages the strengths of *large language models (LLMs)* and *Static Application Security Testing (SAST) tools*. We systematically evaluate to what extent LLMs on their own, and LLMs assisted by SAST tools, can automatically repair cryptographic API misuses by repairing two relevant benchmarks — *CryptoAPI-Bench* and *CamBench* (*Contribution 3*, cf. Section 5.3).

In our experiments, we use a selection of representative LLMs and the two SAST tools *CogniCrypt* and *CryptoGuard*. The results indicate that SAST tools do help LLMs repair vulnerabilities. The best combinations are *CogniCrypt* paired with *gpt-4o* or *llama3.3*, increasing the security rate from 18.4% to 77.2% for *CryptoAPI-Bench* and from 64.8% to 88.9% for *CamBench*. Overall, combining LLMs with SAST tools in *LLM-SAST$_{Repair}$* provides promising repair results. In the future, extending and integrating the approach into workflows may help developers in repairing cryptographic API misuses. Therefore, we integrate *LLM-SAST$_{Repair}$* as a feature into the final main contribution of this thesis, *SecAI* (*Contribution 5*, cf. Chapter 7).

6.1 Introduction

To aid in secure software development, cryptography needs to be used correctly and securely; however, developers struggle [Nad+16]. Employing *Automated Program Repair (APR)* [LPR19] allows for automatically identifying and repairing security

vulnerabilities with *patches*, thus increasing the security of a codebase. Recent work by Bui et al. [Bui+23] empirically evaluated nine state-of-the-art APR tools and one vulnerability-specific repair system across 79 real-world Java vulnerabilities in the Vul4J dataset. Their results reveal that existing APR systems can produce testable patches for only about 20% of vulnerabilities; among these, 73% successfully eliminate the vulnerability, yet just 44% preserve program functionality. In contrast, developments in deep learning lead to increases in patch precision and generalization across multiple benchmarks [Zha+24; Fu+22; JAM25].

New approaches such as RepairLLaMA [SFM25] make use of *Large Language Models (LLMs)* and their potential for APR. However, LLMs are black-box models that lack transparency, making it difficult to understand or predict their decision-making processes [Fan+24]. Additionally, LLMs might introduce security vulnerabilities because their training data often includes insecure or outdated coding practices [Liu+24], which can be inadvertently reproduced in the generated code. Further, LLMs have the potential to introduce new vulnerabilities during code generation through hallucinations [Ji+23], which limit their reliability for critical and secure applications.

One way to address the drawbacks of LLMs is to use *Static Application Security Testing (SAST) tools* as the expert that iteratively supervises the LLM, such that the end user receives a fairly reliable and secure implementation of the code [Nun+24; LDN25]. SAST tools are an established approach for identifying security vulnerabilities in code. They operate by examining the source code or compiled bytecode of a software application without executing it [CW07]. Such tools analyze the code structure, logic, and data flow to identify potential security vulnerabilities, which enables developers to address security flaws at the source code level. SAST tools such as *CogniCrypt* [Krü+19], *CryptoGuard* [Rah+19], *FindSecBugs* [Art20], *SonarQube* [Sonb], etc., have already proven extremely useful. Each has its own ruleset, which security experts can easily update. Due to the inherent undecidability of static code analysis, they can only provide partial solutions in practice [Lan92]. For instance, de Mendonça et al. [Men+13] revealed that SAST tools such as *FindSecBugs* [Art20], *PMD* [PMD], and *CheckStyle* [Che] produce a high rate of false positives. They typically require human intervention to fix detected issues, which raises process costs as they waste time that could be better spent on actual problems [Lan92]. Firouzi et al. [FG25] showed that LLMs can be used for misuse detection, too. They used *gpt-4o* [Ope] to detect vulnerabilities in *CryptoAPI-Bench* [ARY19] and *CamBench* [Sch+22c]. Moreover, Firouzi et al. prompted *gpt-4o* for fixes; however, the evaluation for fixing only checks whether *gpt-4o* made a fix suggestion matching the detected misuse, lacking a validation.

Overall, LLMs seem to be promising for APR with their capabilities to perform code changes, but they typically lack cryptographic expertise. The study of Firouzi et al. [FG25], while limited, already suggests that LLMs can be employed in APR for vulnerability repair, but it lacks a systematic evaluation of the repair and SAST tool support. On the contrary, specialized SAST tools contain cryptographic expert knowledge in their rulesets and can detect vulnerabilities, but typically, they are unable to patch detected issues automatically. If we combine LLMs and SAST tools, we could harness their strengths and mitigate their weaknesses, respectively. Studies [Nun+24; LDN25] show that this combination is promising for the similar domain of code generation for which Kavian et al. [Kav+24] presented a preliminary idea to combine LLMs and SAST tools, yet lacking an empirical evaluation. Thus, we hypothesize that:

Hypothesis: *A combination of LLMs and SAST tools will improve the automatic repair of cryptographic API misuses.* We believe that using a SAST tool to support the LLM in the repair process will yield better results. We thus propose *LLM-SAST*$_{Repair}$, a novel approach to combine LLMs and SAST tools for APR [LPR19] to perform automated vulnerability repair caused by the misuse of cryptographic APIs. To evaluate this hypothesis, we pose two research questions and perform two experiments.

RQ1: *How well do different LLMs on their own perform in automatically repairing cryptographic API misuses in the chosen benchmarks?* Prior work [SFM25; JAM25] has shown that LLMs can be employed for APR. We use *LLM-SAST*$_{Repair}$ to instrument the first evaluation experiment with a representative selection of LLMs performing fault localization and code patching for chosen benchmarks. This way, we can assess how effective LLMs can be in fixing security issues on their own, which LLM performs best, and whether there are differences among benchmarks.

RQ2: *To what extent can a SAST tool support LLMs in automatically repairing cryptographic API misuses?* In *LLM-SAST*$_{Repair}$, SAST tools can be employed for several APR tasks, where they serve as fault locators, warning and fix input providers, and validators. Running the second experiment with multiple combinations of SAST tools and LLMs allows for a deeper assessment.

In this chapter, we aim to automatically repair Java cryptographic API misuses. The chapter presents the following original **contributions**:

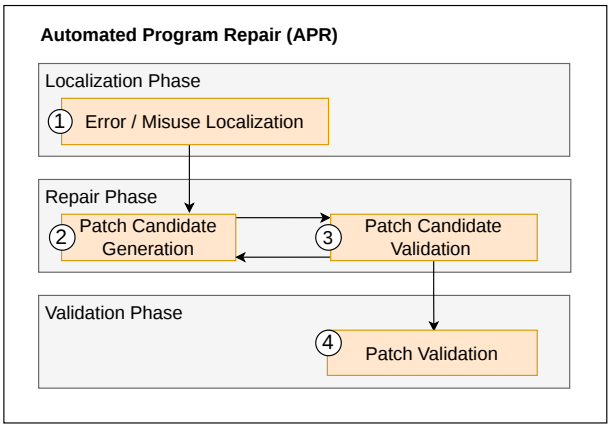


Figure 6.1: High-level visualization of APR tasks.

- We present a systematic empirical evaluation of how well five different LLMs repair cryptographic API misuses on their own and to what extent two SAST tools can support LLMs in automatically repairing cryptographic API misuses. The experiments are performed with the benchmarks *CryptoAPI-Bench* [ARY19] and *CamBench* [Sch+22c]. We show that LLMs struggle with a more secure codebase and that SAST support makes a big difference. The combination of LLMs with a SAST tool increases a benchmark's security rate by about 70%, reaching a total security of nearly 90%.

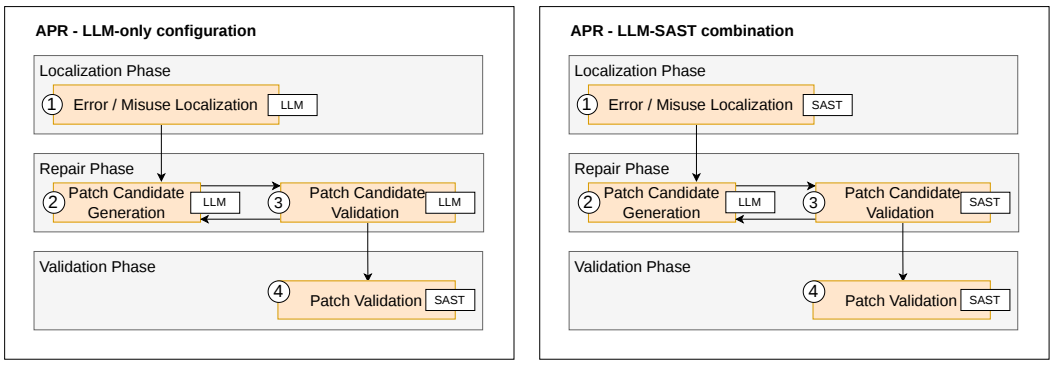
- We propose the APR approach *LLM-SAST*$_{Repair}$, which facilitates the usage and combination of LLMs and SAST. Our implementation of *LLM-SAST*$_{Repair}$ strives for extensibility and generalizability, which is demonstrated by the agnostic integration of LLMs via API keys and the integration of the SAST tools *CogniCrypt* [Krü+19] and *CryptoGuard* [Rah+19].

- We provide an extensive artifact [Sch+26a] that includes the *LLM-SAST*$_{Repair}$ implementation, evaluation scripts, final results, and all intermediary experimental results.

6.2 Approach

We propose *LLM-SAST*$_{Repair}$ as an approach to perform APR [LPR19] by harnessing the respective strengths of LLMs and SAST tools. The goal is to automatically repair cryptographic API misuses by employing the generative power of LLMs and using the automated security support from SAST tools.

APR [LPR19] is the process of using tooling to repair issues in source code without human intervention automatically. For evaluations, typically successful patches and data regarding patch attempts are reported. This automation entails several phases, namely the *Localization Phase*, the *Repair Phase*, and the *Verification Phase* [Zha+23], visualized in Figure 6.1. For *LLM-SAST*$_{Repair}$ to detect and repair security vulnerabilities caused by misuse of cryptographic APIs, we further divide these three phases into four main required tasks. These are ① *Fault localization*, ② *Generation of a repair candidate*, ③ *Validation of a repair candidate*, and ④ *Patch validation* as visualized in Figure 6.1. Most of these tasks can be performed by either LLMs or SAST tools; however, only LLMs have the textual generative power to perform task ② *Generation of a repair candidate* in our approach, and we chose to only use SAST tools for task ④ *Patch validation*, because of their deterministic results. Note that if *LLM-SAST*$_{Repair}$ is configured to employ the same SAST tool for ③ *Validation of a repair candidate* and ④ *Patch validation*, its last validation may be skipped.



(a) *LLM-only* configuration to answer **RQ1**. **(b)** *LLM-SAST* combination to answer **RQ2**.

Figure 6.2: High-level visualization of APR in different configurations.

To test our ***Hypothesis*** and answer our research questions, we use two configurations of *LLM-SAST*$_{Repair}$ presented in Figure 6.2. Both configurations make use of LLMs and SAST tools for at least one of the four tasks for the automatic repair and patch validation, hence the naming of *LLM-SAST*$_{Repair}$. To answer **RQ1**, the first configuration, *LLM-only*, is shown in Figure 6.2a, employing LLMs for the first three tasks and using the SAST only for task ④ *Patch validation*. To answer **RQ2**, the second configuration, *LLM-SAST*, presented in Figure 6.2b, uses SAST tools for all tasks that can be performed by a SAST tool, namely, ① *Fault localization*, ③ *Validation of a repair candidate*, and ④ *Patch validation*. The remaining task ② *Generation of a repair candidate* is performed using LLMs.

LLM-SAST$_{Repair}$ is designed to facilitate the repair of cryptographic API misuses for both described configurations. The logic of *LLM-SAST*$_{Repair}$ to control necessary

components to employ LLMs and SAST tools, their tasks, and the interaction between components, is implemented in the *Logic Component* with a designated *Prompt Manager* for generating the task-dependent prompts for LLM usage. SAST tools are integrated via a *SAST Component*, and LLMs are integrated via an *LLM Component*.

6.2.1 Localization Phase

The *Localization Phase* of *LLM-SAST*$_{Repair}$ starts the APR process with task ① *Fault localization*. The goal is the detection of cryptographic API misuses in code. *LLM-SAST*$_{Repair}$ receives a code file as input for APR.

In configuration *LLM-only* (cf. Figure 6.2a), the *Logic Component* uses the *Prompt Manager* to prompt the *LLM Component*. The *Prompt Manager* uses optimized prompt templates, e.g., shown in Listing 6.3, to task the *LLM Component* with localizing cryptographic API misuses in the code file. If a cryptographic API misuse is detected by the *LLM Component*, the *Logic Component* then moves on to the *Repair Phase*; otherwise, *LLM-SAST*$_{Repair}$ stops for this file since there is nothing to be repaired.

For the second configuration, *LLM-SAST* (cf. Figure 6.2b), the *Logic Component* tasks the *SAST Component* to analyze the code file. If the *SAST Component* reports a cryptographic API misuse, the *Logic Component* initiates the *Repair Phase*; otherwise, *LLM-SAST*$_{Repair}$ stops for this file, too.

6.2.2 Repair Phase

LLM-SAST$_{Repair}$ attempts to repair cryptographic API misuses identified in the *Localization Phase*. The *Repair Phase* is designed to perform tasks ② *Generation of a repair candidate* and ③ *Validation of a repair candidate* in a *patch loop*.

Several *patch loop* iterations may be necessary, e.g., because of so-called *multi-object protocols* [Sch+22a] which are prevalent in cryptography or because of multiple constraints on how to configure cryptographic API objects securely. *Multi-object protocols* require a call sequence over multiple objects to be performed in the correct order and with secure parameters to achieve a goal, e.g., the secure *encryption* of a file. A simplified example in Listing 6.1 shows this effect. The `KeyGenerator` object is insecure because of the insecure `Cipher` object, which needs a secure encryption algorithm and mode to be initialized securely. The automated repair with *LLM-SAST*$_{Repair}$ needed two iterations. First, the insecure algorithm `Blowfish` is detected in Line 208; in the first iteration, the LLM changes this into `AES`, shown

in Line 210, and in iteration two, the insecure default mode is changed to the secure mode `AES/GCM/NoPadding`. In this case, the LLM struggled to choose a secure parameter in the first iteration, but with the support of the SAST tool was able to do so in the second iteration. Wickert et al. [Wic+24] showed that such protocols where multiple objects are involved can cause *dependent errors*, even resulting in *error trees* where multiple *causation locations* (such as Line 208, cf. Listing 6.1) can cause a cryptographic API misuse that manifests at a later position in code (such as Lines 216 and 217, cf. Listing 6.1). When building *LLM-SAST*$_{Repair}$, we saw that such more complicated cryptographic API misuses need multiple iterations of generating a *repair candidate*, even though many issues may be solved with one iteration.

For the first configuration, *LLM-only* (cf. Figure 6.2a), the *Logic Component* employs the *Prompt Manager* to prompt (cf. Listing 6.3) the *LLM Component* to perform task ② *Generation of a repair candidate* and task ③ *Validation of a repair candidate* to attempt to repair the cryptographic API misuse and validate whether the patch candidate is secure. If so, the *patch loop* stops and the *Logic Component* moves to the *Validation Phase*. Otherwise, the *patch loop* runs another iteration or stops if another *exit condition*, such as maximum iteration count or timeout, is met.

Listing 6.1: Simplified example of an insecure case from *CamBench* [Sch+22c] (*context* and *field-sensitive*): BrokenCrypto2. Lines 208 to 212 show the repair of the code in two iterations. The combination of *CogniCrypt* [Krü+19] and *gpt-4o* [Ope] needed two iterations to repair the insecure Line 208. After the first iteration, this was changed to Line 210, and after the second iteration to Line 212.

```
204  public class BrokenCrypto2 {
205    public static void main(String[] args) throws NoSuchPaddingException,
        NoSuchAlgorithmException, InvalidKeyException {
206      CryptoClass cryptoClass = new CryptoClass();
207
208      cryptoClass.cipher1 = Identity("Blowfish");
209      // Iteration 1: Changed from "Blowfish" to "AES"
210      cryptoClass.cipher1 = Identity("AES");
211      // Iteration 2: Changed from "AES" to "AES/GCM/NoPadding"
212      cryptoClass.cipher1 = Identity("AES/GCM/NoPadding");
213
214
215      cryptoClass.cipher2 = Identity("AES/GCM/NoPadding");
216      Cipher cipher = Cipher.getInstance(cryptoClass.cipher1);
217      KeyGenerator keyGen = KeyGenerator.getInstance("AES");
218      cipher.init(Cipher.ENCRYPT_MODE,keyGen.generateKey());
219    }
220    public static String Identity(String s){
221      return s;
222    }
223    public static class CryptoClass{
224      public String cipher1;
225      public String cipher2;
226    }
227  }
```

In the second configuration, *LLM-SAST* (cf. Figure 6.2b), the *Logic Component* also tasks the *Prompt Manager* to prompt the *LLM Component* to perform task ② *Generation of a repair candidate* to attempt to repair the cryptographic API misuse. The *Logic Component* then extracts the code (*patch candidate*) from the output of the *LLM Component* via regular expression (available in the artifact [Sch+26a]). The *Logic Component* next employs the *SAST Component* to perform task ③ *Validation of a repair candidate* and analyze the patch candidate file. If the *SAST Component* still reports a cryptographic API misuse, the *Logic Component* triggers another iteration of the *patch loop*. Else, if the *SAST Component* did not report a cryptographic API misuse, or an *exit condition* is met, it moves to the *Validation Phase* (cf. Section 6.2.3).

6.2.3 Validation Phase

The *Logic Component* starts the *Validation Phase* after the *Repair Phase* ended and the code output is considered the patch generated in the *patch loop*.

For both configurations, *LLM-only* (cf. Figure 6.2a) and *LLM-SAST* (cf. Figure 6.2b), the *Logic Component* employs the *SAST Component* to perform the final task ④ *Patch validation* and analyze the patch. If the *SAST Component* returns no cryptographic API misuse in its report, the patch is considered successful and secure; otherwise, the repair attempt failed. When using one SAST tool in the *LLM-SAST* (cf. Figure 6.2b) configuration, ④ *Patch validation* may be skipped, as the last ③ *Validation of a repair candidate* already performed the analysis. This concludes the APR process of *LLM-SAST*$_{Repair}$.

6.3 Implementation

In this section, we present the implementation details of *LLM-SAST*$_{Repair}$.

The *Logic Component* consists of the *Logic Manager*, the *Prompt Manager*, and a database to store (intermediate) results. The *Logic Manager* orchestrates the logic of the *Logic Component* and the interaction with the *LLM Component* as well as the *SAST Component*.

6.3.1 Logic Component

The *Logic Component* manages the three phases and four tasks of the APR process of *LLM-SAST*$_{Repair}$. It facilitates the interaction between the *SAST Component* and the *LLM Component* via the *Prompt Manager*.

Logic Manager

The *Logic Manager* also takes care of extracting relevant information and storing it in a database. Analysis results provided by the *SAST Component* are read from the corresponding JSON fields in the *SARIF* report, which includes issue descriptions with line references and contextual security guidelines from the respective SAST tool. Since LLM outputs may contain both source code and explanatory text, a parsing step using a regular expression ensures that only valid Java class definitions are extracted, thereby preserving consistency for analysis.

Prompt Manager

The *Prompt Manager* is a crucial component that manages the prompts sent to the *LLM Component*. It ensures that the prompts are tailored to the specific requirements of the respective task and information provided by the *Logic Component*. Performing prompt engineering [Whi+23], we crafted several optimized prompt templates with wildcards to be replaced with the relevant information. The *Prompt Manager* will either generate a prompt for ① *Fault localization*, ② *Generation of a repair candidate*, or ③ *Validation of a repair candidate*. For instance, our engineered repair prompt template in Listing 6.2 sets the role of the *LLM Component* as a security expert in cryptographic algorithms, provides the input code context, and instructs the repair of vulnerable parts based on the provided vulnerability reports.

Listing 6.2: Prompt template for patch candidate generation with SAST tool support.

```
228  {"system": "You are a security expert in cryptographic algorithms. You try to write the
        most secure code everytime. You code in Java programming language. Every code you write
        should be able to compile on its own. You can fix code based on comments provided by a
        static analysis tools. Important: Your task is not to clean the code or add any other
        information based on personal knowledge but to fix the code according to the
        instruction. Do not change the original code except when absolutely necessary to fix the
        provided error. Make sure to provide the complete java class again with the same name.",
        "user": "This is the  previous code:\n {{ code }} \n\n Fix the code based on the
        following warnings provided by a static analysis tool:\n {{ error_report }}. If multiple
        tools report the same error, fix it only once."}
```

Listing 6.3: Prompt template for patch candidate generation without SAST support.

```
229  {"system":"You are a security expert in cryptographic algorithms.You try to write the most
       secure code everytime. You code in Java programming language. A security vulnerability
       is a flaw, weakness, misconfiguration, or bug in software or system configurations that
       can be exploited by attackers to gain unauthorized access, steal data, or disrupt
       services, compromising the system's confidentiality, integrity, or availability. Check
       if there are any security vulnerabilities in the given code. if yes, then fix it. if
       there are no security vulnerabilities, just return the same code. Every code you write
       should be able to compile on its own.", "user": "This is the  previous code:\n {{ code
       }} \n\n If the code doesn't contain any security vulnerability, return the original
       code,but dont make any changes in code and then write a comment at the end saying '/* No
       Vulnerability Found */ \n Do not worry about code quality , just focus on solving
       vulnerabilites"}
```

LLM Component

The *LLM Component* is responsible for executing the prompts provided by the *Prompt Manager* for the configured LLM. For instance, to generate a *repair candidate*, it takes the provided code, analyzes it for security issues, and generates fixes, e.g., as instructed by the prompt in Listing 6.3.

SAST Component

The *SAST Component* facilitates running different SAST tools. In *LLM-SAST*$_{Repair}$, we have integrated *CogniCrypt* [Krü+19] and *CryptoGuard* [Rah+19]. Depending on the configuration, the *SAST Component* runs the analysis of the respective tool on the code received from the *Logic Manager* and returns its result in the form of a SARIF report. The abstraction layer introduced with the *SAST Component* is designed such that multiple SAST tools can be integrated.

By designing the architecture of *LLM-SAST*$_{Repair}$ to be modular, making use of the SARIF format widely adopted by SAST tools, and LLM API calls, we strive for extendability. In our evaluation experiments, we show this by supporting a selection of five LLMs (cf. Table 6.1) and two SAST tools, namely *CryptoGuard* [Rah+19] and *CogniCrypt* [Krü+19].

The *SAST Component* is used as an abstraction layer to enable *LLM-SAST*$_{Repair}$ to support several SAST tools. For flexibility in the integration of SAST tools, we use *SARIF* [OAS20], a JSON-standardized format for SAST tools for reporting their findings, which is adopted by many SAST tools. Using *SARIF*, we have integrated *CogniCrypt* [Krü+19] and *CryptoGuard* [Rah+19]. Depending on the configuration, the *SAST Component* runs the analysis of the respective tool on the code received

from the *Logic Manager* and returns its result in the form of a *SARIF* [OAS20] report. Hence, the *Logic Component* reads the fields containing information it needs.

LLM-SAST$_{Repair}$ was designed with components as an abstraction layer to be extendable and generalizable. To show this, we have integrated two SAST tools, namely *CogniCrypt* [Krü+19] in version 4.0.1 and *CryptoGuard* [Rah+19] in release 04.05.03. We also tested *CodeQL* [Gitb]; however, its recall on either benchmark we used in preliminary evaluations was too low to be considered for, e.g., *patch evaluation*, as it misses too many cryptographic API misuses. In preliminary tests, this also showed the flexibility of the approach — we can run *LLM-SAST*$_{Repair}$ for other programming languages such as C, C++, etc., which is supported by *CodeQL*, highlighting the generalization potential of the approach.

6.4 Evaluation

To assess the capabilities and performance of *LLM-SAST*$_{Repair}$ in terms of repairing security issues in code, we perform an evaluation in both configurations shown in Figure 6.2. Similar to the study of Firouzi et al. [FG25], who investigated how well *gpt-4o* performs in detecting security issues, we chose *CryptoAPI-Bench* [ARY19] and *CamBench* [Sch+22c] for benchmarking. We run two different experiments. To answer **RQ1**, we run *LLM-SAST*$_{Repair}$ using the *LLM-only* configuration (cf. Figure 6.2a) for the selected LLMs (cf. Table 6.1). Unlike Firouzi et al. [FG25], we are also interested in whether different LLMs perform better than others. Therefore, we made a selection of LLMs to cover a range of capabilities, from commercial models to open-weight alternatives that are lightweight and cost-efficient. The selected LLMs are listed in Table 6.1 with their respective selection rationale. For instance, *llama3.1* [Met24] is a lightweight open-weight model and, in pre-testing, was deployed on a MacBook Pro M1, demonstrating the ability to run *LLM-SAST*$_{Repair}$ locally on resource-constrained environments. To answer **RQ2**, we run *LLM-SAST*$_{Repair}$ in configurations making use of two SAST tools (cf. Figure 6.2b) and LLMs where the SAST tools *CogniCrypt* and *CryptoGuard* perform the tasks of ① *Fault localization*, ③ *Validation of a repair candidate*, and ④ *Patch validation* (cf. Figure 6.1).

6.4.1 Setup

The evaluation experiments were performed on a server with Debian OS with one NVIDIA H100 GPU, 32 GB RAM, 8 CPU cores, and 300 GB SSD of storage. The server

Table 6.1: Selected two commercial and three open-weight LLMs for the evaluation of *LLM-SAST$_{Repair}$* with rationale for selection. Model configurations are provided in the artifact [Sch+26a]. Models were downloaded from Hugging Face [Hug].

LLM model	Abbreviation	Reason for selection
GPT-4o [Ope]	*gpt-4o*	large commercial model
GPT-4o-mini [Ope]	*gpt-4o-mini*	lightweight commercial model, cost-efficient
Qwen3-Coder-30B-A3B-Instruct-FP8 [Ali]	*qwen-3.0*	code generation model
Llama-3.3-70B-Instruct [Met24]	*llama3.3*	large open-weight model
Llama-3.1-8B-Instruct [Met24]	*llama3.1*	lightweight, deployable offline on a laptop

was used to run benchmarks for *LLM-SAST$_{Repair}$* for different configurations. The different benchmark datasets required different setups to run the evaluation. Hence, different versions of Java, including Java 8 and Java 11, were installed on the server. The evaluation script was written in Python 3.9.19 and used the subprocess module to run the evaluations in parallel.

We describe the evaluation process. First, the ground truth is loaded from the truth file, which contains the expected vulnerabilities and their line numbers. Every file in the benchmark is passed through the *LLM-SAST$_{Repair}$* framework. We collect the APR results of *LLM-SAST$_{Repair}$* in its database. We compare the initial ① *Fault localization* report and the final ④ *Patch validation* report after the *Repair Phase* (cf. subsection 6.2.2) has finished to calculate the number of errors detected and successfully repaired. For the ④ *Patch validation*, we employ SAST tools as an automated heuristic to analyze whether the patch is secure. Finally, the results are stored in a CSV file for manual evaluation. To account for the non-deterministic output of LLMs, we repeat this process three times for each combination of *LLM-SAST$_{Repair}$* configuration and benchmark, and report the best-of-three result for robustness [FG25]. We employ the SAST tools as automated heuristic validators of the patches because manual validation of the large number of files is infeasible. In total, we run 45 repair runs (five LLMs and two SAST tools result in 15 unique configurations, three runs each) for two benchmarks each, resulting in up to 34,155 files to be validated.

6.4.2 Definition of Metrics

In Table 6.2 and Table 6.3, we list and define the evaluation metrics we use for clarity. Table 6.2 shows standard metrics for testing evaluation.

Table 6.2: Standard metrics and definitions for static analysis — provided with abbreviations, and description of metrics used in the evaluation.

Metric	Abbr.	Definition	Description
True Positive	TP		Misuse detected correctly based on ground truth
False Positive	FP		Misuse detected incorrectly based on ground truth
False Negative	FN		Undetected misuse based on ground truth
True Negative	TN		Correctly no report based on ground truth
Precision	Pr	TP/(TP+FP)	Precision of detection reports
Recall	Re	TP/(TP+FN)	Overall fraction of correctly detected misuses
F1-Score	F1	TP/(TP+0.5*(FP+FN))	Harmonic mean of the precision and recall

In our APR approach, we perform several testing tasks, especially task ① *Fault localization*, for which the metrics of Table 6.2 are used, and ④ *Patch validation*, where the code was changed, hence making the ground truth inapplicable. Therefore, we introduce the definitions in Table 6.3. For clarity, we do not reuse terms such as *true positive (TP)* when discussing patched files, because the *TP* is decided on the basis of a ground truth which we do not have for the patched code. Since we reason about the patched code in the context of it being secure or vulnerable before, according to the ground truth, we define new terms consisting of two parts, respectively. The first part is either *True* or *False* according to the ground truth, and the second part of the term is either *Fixed* or *Not Fixed*, denoting the SAST validation result; e.g., a *True Fixed (TF)* means that a vulnerable file (ground truth *T*) was repaired and is secure according to the respective SAST tool. Similarly, *True Not Fixed (TNF)* means that a vulnerable file was not repaired (fixed). The concerning case is the *False Not Fixed (FNF)*, as this denotes a case where a secure file (ground truth *F*) was falsely changed and is then vulnerable according to the respective SAST tool (*NF*).

Figure 6.6 visualizes this as a Sankey diagram of one patch run of *LLM-SAST*$_{Repair}$ using the *LLM-SAST* configuration (cf. Figure 6.2b) on *CamBench* with *CogniCrypt* and *llama3.3*. From left to right, the Sankey diagram starts with the benchmarks' ground truth; negative cases are shown in green, and positives in red. Next, in task ① *Fault localization* benchmark cases are analyzed by *CogniCrypt* and categorized as *TP*, *FP*, *TN*, and *FN*. Only reported issues (*TP* and *FP*) are repaired in tasks ② *Generation of a repair candidate* by *llama3.3*, and ③ *Validation of a repair candidate* is performed by *CogniCrypt*. In task ④ *Patch validation*, the patches are validated by *CogniCrypt* and categorized as *TF*, *FF*, *TNF*, and *FNF*. The final security of the patched benchmark is displayed on the right side, showing secure cases in green and insecure cases in red. Note that *TN* and *FN* (cases without reported issue in task ① *Fault localization*) are directly carried over to secure and insecure, respectively. The metrics from Table 6.2 are used to evaluate the benchmark and the results of

task ① *Fault localization*, and the metrics from Table 6.3 are used to evaluate the results of ④ *Patch validation* and final security results.

Table 6.3: Definitions of the metrics used after APR. *GT* denotes the ground-truth value for each benchmark case.

Metric	Abbr.	Definition	Description
True Fixed	TF	–	No cryptographic API misuse reported and *GT* was insecure.
False Fixed	FF	–	No cryptographic API misuse reported, but *GT* was secure.
False Not Fixed	FNF	–	Cryptographic API misuse reported, but *GT* was secure.
True Not Fixed	TNF	–	Cryptographic API misuse reported and *GT* was insecure.
Exception	Ex	–	The repair attempt resulted in incompilable code.
Positives	Pos	–	Total number of insecure cases in *GT*.
Negatives	Neg	–	Total number of secure cases in *GT*.
Initial Security Rate	ISR	$\mathrm{Neg}/(\mathrm{Pos}+\mathrm{Neg})$	Initial rate of secure files in the benchmark.
Final Security Rate	FSR	$(\mathrm{TF}+\mathrm{FF}+\mathrm{TN})/(\mathrm{Pos}+\mathrm{Neg})$	Final rate of secure files after repair.
Normalized Security Increase	NSI	$(\mathrm{FSR}-\mathrm{ISR})/(1-\mathrm{ISR})$	Normalized rate of security increase after patching.

To reason about the effectiveness of the APR for the whole benchmark, we introduce three metrics describing the security. Since the benchmarks differ in size and distribution of vulnerable files, we define the *Initial Security Rate (ISR)* to describe the fraction of secure files in the benchmark of its total number of files. The goal of *LLM-SAST$_{Repair}$* is to automatically increase the security of code, e.g., the benchmarks in the evaluation. Therefore, we define metrics indicating this security as follows, because other metrics used in APR often lack security focus: The *Final Security Rate (FSR)* is the rate of secure files in the patched benchmark. This also includes *False Fixed (FF)* cases, where secure code was unnecessarily changed, but is still secure and therefore rather benign. To make the security rate changes comparable between benchmarks, we need to normalize them with the *Normalized Security Increase (NSI)* since, e.g., a benchmark with a small fraction of vulnerable files only has a low potential to increase the security rate.

6.4.3 Benchmarking

LLM-SAST$_{Repair}$ was designed to repair security issues in Java code. To ensure a comprehensive evaluation, we selected two benchmarks focused on security vulnera-

bilities in Java code, matching the domain of the two SAST tools we use – cryptographic API misuse detection: *CryptoAPI-Bench* [ARY19], and *CamBench* [Sch+22c] (cf. Table 6.4). Further, this combination of benchmarks has been used in a similar study by Firouzi et al. [FG25], where they mainly used *gpt-4o* for misuse detection. We report the best-of-3 run for each configuration for robustness of results [FG25], because of the non-deterministic behavior of LLMs.

For task ④ *Patch validation*, we have used both *CogniCrypt* [Krü+19] and *CryptoGuard* [Rah+19] as validators for each run. Due to limited space, we only report the results for *CogniCrypt* as a validator in this chapter; all results including *CryptoGuard* as a validator are contained in the artifact [Sch+26a]. The results of the experiment for **RQ2** show clearly that *CogniCrypt* is the more conservative validator. Although it reports many false positives on *CamBench*, namely 243 *FP* (cf. Table 6.6), it only misses 3 (*FN*) of 219 insecure test cases, resulting in a high recall of 98.6%. Thus, when using *CogniCrypt* as a heuristic for ④ *Patch validation*, its high rate of false positives mainly leads to an underestimation rather than an overestimation of the final security rate *FSR*. For instance, the *FSR* of 0.889 measured for the *LLM-SAST* configuration (cf. Figure 6.2b) on *CamBench* with *CogniCrypt* and *llama3.3* (cf. Table 6.10) should therefore be interpreted as a conservative lower-bound estimate; the actual value is likely closer to ∼0.95. The code of the 243 *FP* was changed in the repair process, yet most were evaluated to be still secure; namely, 203 *FF* cases.

Table 6.4: Selected benchmarks for the evaluation of *LLM-SAST*$_{Repair}$. *ISR* denotes the *initial security rate.*

Benchmark	File Count	Positives	Negatives	ISR
CamBench [Sch+22c]	623	219	404	0.648
CryptoAPI-Bench [ARY19]	136	111	25	0.184

CryptoAPI-Bench

CryptoAPI-Bench [ARY19] is a micro-benchmark for cryptographic APIs (cf. Section 5.3.2). In our experiments, we only use a subset of 136 of the originally 160 test cases that cover various security vulnerabilities in cryptographic APIs, because *LLM-SAST*$_{Repair}$ is limited to single files as input. Further, Torres et al. [Tor+23] have corrected the partially incorrect ground truth of *CryptoAPI-Bench*. We use that corrected ground truth here. The test cases are divided into field-sensitive, path-sensitive, basic, and interprocedural properties to assess static analysis capabilities better. They are further divided into 16 different types of tests corresponding to the 16 vulnerability patterns of *CryptoGuard*, such as *predictable seed*, *predictable pbe*

password, *predictable keystore password*, *credential stored in string, static salt*, etc. This benchmark subset consists of more vulnerable *positive* cases (111) than secure *negative* cases (25), which makes it a good candidate for evaluating the performance of *LLM-SAST$_{Repair}$* in terms of true error reduction. We report the analysis results of the different SAST tools on *CryptoAPI-Bench* from our experiment in Table 6.5. Since we use a subset with corrected ground truth [Tor+23], there is no preexisting comparison. The results in Table 6.5 help in assessing the SAST tools' capabilities for task ④ *Patch validation* in the experiments.

Table 6.5: Initial analysis results of SAST tools for *CryptoAPI-Bench*.

Benchmark	SAST	FC	TP	FP	FN	TN	Precision	Recall	F1-score
CryptoAPI-Bench	*CogniCrypt*	136	92	17	19	8	84.4%	82.9%	0.836
CryptoAPI-Bench	*CryptoGuard*	136	76	14	35	11	84.4%	68.5%	0.756

CamBench

CamBench [Sch+22c] (cf. Sections 5.3 and 5.3.8) is a micro-benchmark for cryptographic APIs, which contains 623 test cases that cover various cryptographic API misuses. *CamBench* consists of 219 *positive* and 404 *negative* tests. The types of tests include various sensitivity types, such as context-, field-, flow-, object-, path-, and mixed-sensitivity. For each type of test, it is further divided into *broken cryptographic algorithm*, *broken hash function*, *ECB mode*, *insecure random value generation*, *PBE parameters*, *small key size*, and *static initialization vector*. We report the analysis results of different SAST tools on *CamBench* from our experiment in Table 6.6. The results in Table 6.6 help in assessing the SAST tools' capabilities for task ④ *Patch validation* in the experiments.

Table 6.6: Initial analysis results of SAST tools for *CamBench*.

Benchmark	SAST	FC	TP	FP	FN	TN	Precision	Recall	F1-score
CamBench	*CogniCrypt*	623	216	243	3	161	47.1%	98.6%	0.637
CamBench	*CryptoGuard*	623	161	95	58	309	62.9%	73.5%	0.678

6.4.4 Research Question 1

In **RQ1**, we assess to what extent LLMs (cf. selection of LLMs in Table 6.1) can detect and repair security issues using *LLM-SAST$_{Repair}$*. We run each LLM performing ① *Fault localization*, ② *Generation of a repair candidate*, and ③ *Validation of a repair*

Table 6.7: Results of *LLM-SAST$_{Repair}$* fixing *CryptoAPI-Bench* running different selected LLMs and using *CogniCrypt* as validator. Best-performing run is highlighted.

	Fault Localization Results							Patch Validation Results						
LLM	TP	FP	FN	TN	Pr	Re	F1	Ex	TF	FF	TNF	FNF	FSR	NSI
gpt-4o-mini	106	24	5	1	81.5%	95.5%	0.880	0	67	21	39	3	0.654	0.577
gpt-4o	89	15	22	10	85.6%	80.2%	0.828	0	44	6	45	9	0.441	0.315
qwen-3.0	90	18	21	7	83.3%	81.1%	0.822	7	40	10	50	8	0.419	0.288
llama3.1	69	16	42	9	81.2%	62.2%	0.704	3	26	9	43	7	0.324	0.171
llama3.3	55	13	56	12	80.9%	49.5%	0.615	0	21	5	34	8	0.279	0.117

Table 6.8: Results of *LLM-SAST$_{Repair}$* fixing *CamBench* running different selected LLMs and using *CogniCrypt* as validator. Best-performing run is highlighted.

	Fault Localization Results							Patch Validation Results						
LLM	TP	FP	FN	TN	Pr	Re	F1	Ex	TF	FF	TNF	FNF	FSR	NSI
gpt-4o	187	72	32	332	72.2%	85.4%	0.782	0	55	17	132	55	0.648	0.000
gpt-4o-mini	214	359	5	45	37.3%	97.7%	0.540	0	101	177	113	182	0.518	-0.370
qwen-3.0	201	287	18	117	41.2%	91.8%	0.569	67	73	129	128	158	0.512	-0.388
llama3.3	153	221	66	183	40.9%	69.9%	0.516	0	48	70	105	151	0.483	-0.470
llama3.1	174	289	45	115	37.6%	79.5%	0.510	30	48	131	126	158	0.472	-0.502

candidate three times for each benchmark and report the respective best-of-three run validated with *CogniCrypt* for robustness of results. Table 6.7 shows the results for *CryptoAPI-Bench* and Table 6.8 for *CamBench*. Figure 6.3 and Figure 6.4 present the best run as a Sankey diagram, respectively.

All LLMs are able to increase the *FSR* for *CryptoAPI-Bench*, which has a very low *ISR* (cf. Table 6.4) of only 0.184. However, for *CamBench*, with an *ISR* of 0.648, no LLM was able to increase the *FSR*, which is visible in the negative or zero *NSI* (cf. Tables 6.7 and 6.8). The best-performing run of *gpt-4o* was the only run overall that did not make the benchmark less secure. It had the same amount (55) of insecure cases repaired (*TF*) as it made secure cases insecure (*FNF*) (cf. Table 6.8). Moreover, for *CamBench*, LLMs on their own generally produce a high number of the costly metric *FNF*, representing originally secure code made insecure. In most cases, the number of *FNF* is larger than the number of *FF*, secure cases that were changed but remain secure. That means that LLMs often turn a large fraction of *False Positives (FP)*, secure code (falsely reported as insecure), into insecure code. Run1 of *gpt-4o* was the only run producing fewer than 100 *FNF* for *CamBench*, because of its comparably low number of *False Positives (FP)* (*FNF* are *FP* that were made insecure). Further, some LLMs, *qwen-3.0* and *llama3.1*, struggled to generate compilable code resulting in *exceptions*.

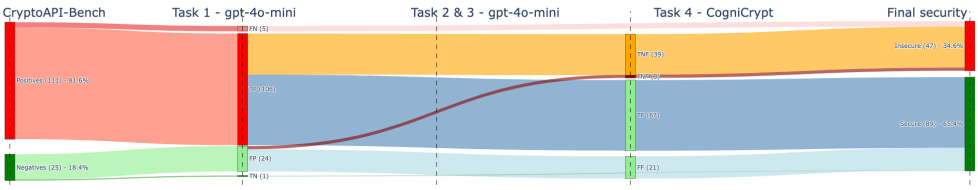


Figure 6.3: Best *LLM-only* configuration *LLM-SAST*$_{Repair}$ patch run for *CryptoAPI-Bench*: gpt-4o-mini-run1.

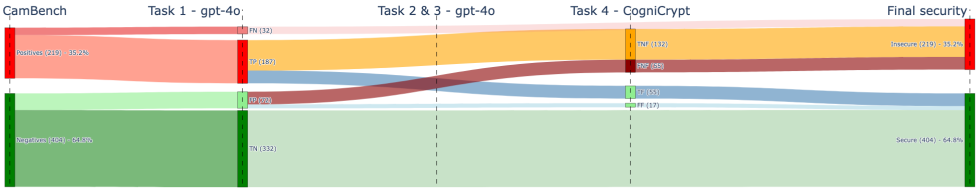


Figure 6.4: Best *LLM-only* configuration *LLM-SAST*$_{Repair}$ patch run for *CamBench*: gpt-4o-run1.

The good performance of *gpt-4o-mini* in repairing *CryptoAPI-Bench* can be explained by its low number of 6 negative reports (5 *FN* and 1 *TN*) — it basically attempted to fix nearly everything. This approach works well for a codebase or benchmark with a low rate of security, such as *CryptoAPI-Bench* with an *ISR* of 0.184. The low *ISR* means that there are many insecure cases that can be repaired while only a few secure cases can be broken.

The opposite is the case for *CamBench* with an *ISR* of 0.648. The resulting high number of *FP* (359) leads to *gpt-4o-mini* creating a lot of *FNF* and decreasing the security of *CamBench* with an *NSI* of -0.370. In contrast, *gpt-4o* performs really well in detecting secure code with 332 *TN* (precision = 72.2%, recall = 85.4%) for *CamBench*. The retention of so many secure cases explains why *gpt-4o* performed best.

Observation 6.1 – RQ1

LLMs failed to increase the total final security rate for *CamBench*, representing a more secure codebase (*ISR = 0.648*). Only for the less secure *CryptoAPI-Bench* (*ISR = 0.184*) LLMs were able to increase the security consistently. Generally, commercial LLMs performed better.

6.4.5 Research Question 2

Table 6.9: Results of *LLM-SAST*$_{Repair}$ repairing *CryptoAPI-Bench* in different selected combinations for LLMs and SAST tools; validator is *CogniCrypt*. Best-performing combination is highlighted.

LLM	SAST	Ex	TF	FF	TNF	FNF	FSR	NSI
gpt-4o	*CogniCrypt*	0	80	17	12	0	0.772	0.721
qwen-3.0	*CogniCrypt*	0	72	12	20	5	0.676	0.604
llama3.3	*CogniCrypt*	0	69	14	23	3	0.669	0.595
gpt-4o-mini	*CogniCrypt*	0	67	14	25	3	0.654	0.577
llama3.1	*CogniCrypt*	0	57	8	35	9	0.537	0.432
llama3.3	*CryptoGuard*	0	41	7	35	7	0.434	0.306
gpt-4o	*CryptoGuard*	0	35	7	41	7	0.390	0.252
qwen-3.0	*CryptoGuard*	2	26	7	50	7	0.324	0.171
gpt-4o-mini	*CryptoGuard*	0	26	6	50	8	0.316	0.162
llama3.1	*CryptoGuard*	1	27	5	49	9	0.316	0.162

In **RQ2**, we evaluate to what extent SAST tools can aid LLMs in repairing security issues using *LLM-SAST*$_{Repair}$. We use *CogniCrypt* and *CryptoGuard* as SAST tools for tasks ① *Fault localization* and ③ *Validation of a repair candidate*. ④ *Patch validation* is reported for *CogniCrypt*. They are combined with each selected LLM (cf. Table 6.1), performing task ② *Generation of a repair candidate*. Each combination is run three times for each benchmark for robustness of results to report the best-of-three result.

The analysis results for task ① *Fault localization* are shown in Table 6.5 and Table 6.6 and are deterministic throughout our experiment. Table 6.9 shows the best-of-three results for *CryptoAPI-Bench* and Table 6.10 for *CamBench*. Figure 6.5 and Figure 6.6 visualize the best configuration and run as a Sankey diagram, respectively.

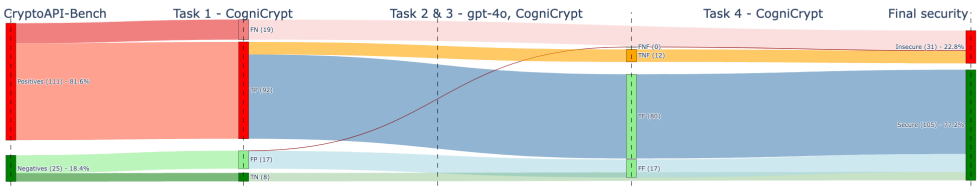


Figure 6.5: Best *LLM-SAST* configuration *LLM-SAST*$_{Repair}$ patch run for *CryptoAPI-Bench*: CogniCrypt-gpt4o-run2.

LLM-SAST$_{Repair}$ was able to increase the *FSR* of both benchmarks in all combinations of SAST tools and LLMs. Further, all combinations with *CogniCrypt* as SAST out-

Table 6.10: Results of *LLM-SAST*$_{Repair}$ repairing *CamBench* in different selected combinations for LLMs and SAST tools; validator is *CogniCrypt*. Best-performing combination is highlighted.

LLM	SAST	Ex	TF	FF	TNF	FNF	FSR	NSI
llama3.3	*CogniCrypt*	0	190	203	26	40	0.889	0.685
gpt-4o	*CogniCrypt*	0	189	199	27	44	0.881	0.662
llama3.1	*CogniCrypt*	6	165	163	51	80	0.785	0.388
gpt-4o-mini	*CogniCrypt*	0	162	148	54	95	0.756	0.306
qwen-3.0	*CogniCrypt*	0	162	135	54	108	0.735	0.247
gpt-4o	*CryptoGuard*	0	89	44	72	51	0.709	0.174
llama3.3	*CryptoGuard*	3	90	43	71	52	0.709	0.174
qwen-3.0	*CryptoGuard*	19	76	44	85	51	0.689	0.114
gpt-4o-mini	*CryptoGuard*	0	77	38	84	57	0.681	0.091
llama3.1	*CryptoGuard*	0	66	38	95	57	0.663	0.041

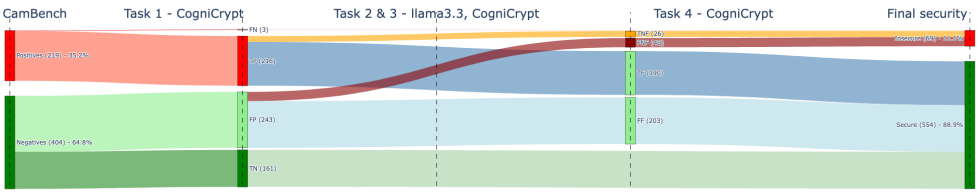


Figure 6.6: Best *LLM-SAST* configuration *LLM-SAST*$_{Repair}$ patch run for *CamBench*: CogniCrypt-llama3.3-run0.

performed all combinations with *CryptoGuard*– the *FSR* was increased to a higher rate even for the weakest *CogniCrypt* combination per benchmark. Moreover, most *CogniCrypt* combinations produce fewer *FNF* than *CryptoGuard* combinations; only weaker performing *CogniCrypt* combinations with *llama3.1*, *gpt-4o-mini*, and *qwen-3.0* result in a higher number of *FNF* for *CamBench*.

This is interesting, considering that *CryptoGuard* produces fewer *False Positives (FP)* than *CogniCrypt* for both benchmarks, meaning that there are fewer secure cases that *LLM-SAST*$_{Repair}$ attempts to repair and thus fewer opportunities to change an originally secure case into an insecure one. Although *CogniCrypt* produces a high number of *FPs*, its strong combinations with *gpt-4o* and *llama3.3* produce the lowest number of *FNF*. Overall, the best combination was *CogniCrypt* with *gpt-4o*.

Observation 6.2 – Hypothesis

LLM-SAST$_{Repair}$ with SAST tools manages to increase the overall security of both benchmarks. All combinations of *CogniCrypt* perform better than any combination of *CryptoGuard*. The best *CogniCrypt* combinations repair a large fraction of insecure cases while showing a low rate of turning secure cases insecure — between 0% (*CryptoAPI-Bench*) and 16.5% (*CamBench*). Overall, the best-performing combinations use *CogniCrypt* and an LLM with a larger number of parameters; here, *gpt-4o* and *llama3.3*.

Discussion

Similar to the results from the *LLM-only* experiment, the *LLM-SAST* combinations with *qwen-3.0* and *llama3.1*, but also *llama3.3*, produced *exceptions*; however, far fewer, e.g., for *CamBench* the number for *qwen-3.0* dropped from 67 to 19 for *CryptoGuard* and 0 for *CogniCrypt*. Generally, *CogniCrypt* combinations produced fewer *exceptions* than *CryptoGuard* combinations. Compared to combinations using only LLMs, especially *CogniCrypt* combinations performed better.

The respective best-performer for *CryptoAPI-Bench* is *CogniCrypt* with *gpt-4o* and achieved an *FSR* of 0.772 and no *FNF*, while *gpt-4o-mini* as LLM-only achieved a lower *FSR* of 0.654 and produced more *FNF*, 3. The respective best-performer for *CamBench* is *CogniCrypt* with *llama3.3*, which achieved an *FSR* of 0.889 and only 40 *FNF*, while *gpt-4o* achieved a lower *FSR* of 0.648 and produced more *FNF*, 55. Overall, *LLM-SAST*$_{Repair}$ with SAST tool support for the LLM resulted in fewer *FNF* than in the LLM-only configurations. This means that SAST tools support LLMs in reducing cases where secure code is falsely made insecure.

When using a SAST tool to support the LLM in *LLM-SAST*$_{Repair}$, the *ISR* of the respective benchmark seems to matter less – the best result for *CamBench* (*ISR* of 0.648) with an *NSI* of 0.685 comes close to the best result for *CryptoAPI-Bench* with an *NSI* of 0.721. In contrast, no LLM on its own could increase the final security of *CamBench* with its high *ISR* of 0.648.

In terms of required iterations to patch a benchmark, *CogniCrypt* combinations performed a median of one iteration in the *patch loop* and averaged between 1.081 and 1.706 for *CryptoAPI-Bench*, and 0.822 to 1.404 for *CamBench*. *CryptoGuard* combinations performed a median of 0 iterations in the *patch loop* and averaged between 0.809 and 0.963 for *CryptoAPI-Bench*, and 0.443 to 0.552 for *CamBench*. LLMs on their own showed a tendency to perform more iterations in the *patch loop*;

they performed a median of between 1 and 2 iterations and averaged between 1.074 and 1.853 for *CryptoAPI-Bench*, and 1.040 to 1.628 for *CamBench*.

Examining both experiments, several factors can be identified for *LLM-SAST*$_{Repair}$ to perform well in automatically repairing cryptographic API misuses.

- The performance of the LLM or SAST tool chosen for ① *fault localization* is important. A high number of *TN* and a low number of *FN* help in retaining secure code while being able to attempt to repair many insecure cases. For instance, *gpt-4o* for *CamBench* retains a large number of *TN* (82.2% of secure code stays unchanged); this is visualized in Figure 6.4.

- The quality of the SAST tool is more important than the LLM it is paired with; *CogniCrypt* combinations performed best. For our APR approach (cf. Figure 6.2), it seems that a high *recall* is more important than high *precision*, because most *FP* stay secure and are turned into *FF*. One reason for this could be detailed information on the issue in the SAST tool's warning reports.

- In SAST-tool supported combinations, the LLM plays an important role in repairing reports and keeping secure code (*FP*) secure. For instance, for *CamBench*, *CryptoGuard's* rate of *FP* turned into insecure (*FNF*) ranges between 53.7% and 60.0%; for *CogniCrypt*, the rate ranges between 16.5% and 44.4%.

Observation 6.3 – RQ2

LLM-SAST$_{Repair}$ performs better with SAST tools supporting the LLM in the repair. The achieved final security rate *FSR* is higher compared to *LLM-only* configurations while turning fewer initially secure cases insecure (*FNF*), and having fewer issues with *exceptions* and performing fewer iterations in the *patch loop*. Overall, the best-performing combinations use *CogniCrypt* with *gpt-4o* or *llama3.3*.

Overall, the empirical evaluation indicates that our hypothesis of SAST tools being able to support LLMs in automatically repairing cryptographic API misuses holds true. The results for **RQ1** show that LLMs on their own are generally capable of repairing cryptographic API misuses; however, they struggle when attempting to repair a rather secure codebase. The results for **RQ2** demonstrate that the support of SAST tools yields a more reliable increase in security and can compensate for shortcomings of the LLMs.

6.4.6 Limitations

By designing *LLM-SAST*$_{Repair}$ to make use of existing LLMs and SAST tools, their respective limitations, e.g., hallucinations for LLMs and false positive reports for SAST tools, carry over. Further, *LLM-SAST*$_{Repair}$ is currently limited to a single file as input. For our evaluation experiments, we used two relevant micro-benchmarks which are well suited for the input and output token size of LLMs. However, this limits the scalability of achieved APR results.

6.4.7 Threats to Validity

In this section, we report threats to validity of our study as well as chosen mitigation strategies [Woh+12].

Construct validity

Using a SAST tool as an automated heuristic for ④ *patch validation* may introduce biases towards the tool's capabilities and limitations, e.g., imprecision. Therefore, the exact security rates are also approximations; nevertheless, the indicated effect can clearly be seen. To mitigate this, we ran two SAST tools and include all results (for *CryptoGuard* [Rah+19] as validator) in the artifact [Sch+26a]. For instance, as discussed in Section 6.4.3, the high amount of *FP* of *CogniCrypt* (in combination with its high *recall*) when used as validator rather means that it underestimates the amount of secure cases. Further, we present the SAST tools' performance metrics for each benchmark in Table 6.5 and Table 6.6, allowing for comparison of the quality of these approximations.

Internal validity

The non-deterministic behavior of LLMs is a threat to the consistency of our evaluation results. For mitigation, we ran each configuration of LLM-only and LLM-SAST combination three times and reported the best-of-three result.

Even though the LLMs receive clear and restricted prompts asking not to remove cryptographic code, this is not completely guaranteed. In pre-experiments as well as the final experiment, we manually checked random cases and never found this behavior.

Reliability validity

The non-deterministic behavior of LLMs poses a threat to the reliability of the presented results. To mitigate this, we further provide all data of the experiments, including each changed code in repair attempts (intermediary results) and final patched code for each run, in our artifact [Sch+26a].

External validity

The evaluation experiments were limited to the scope of the two supported SAST tools, *CogniCrypt* [Krü+19] and *CryptoGuard* [Rah+19], and the two benchmarks, *CryptoAPI-Bench* [ARY19] and *CamBench* [Sch+22c]. While these benchmarks are tailored to Java cryptographic usages, specifically of the JCA, expanding the experiments to more benchmarks and real-world code would improve the robustness of the results.

Limitations of the used SAST tools and LLMs pose a threat to validity. To mitigate this, we implemented support for two SAST tools and made a representative selection of LLMs.

For further generalizability, the implementation of *LLM-SAST*$_{Repair}$ supports the standard Static Analysis Results Interchange Format (SARIF) [OAS20], allowing for future integration of other SAST tools. Based on the SAST tool's language capabilities and adaptations of the prompts for LLM usage, *LLM-SAST*$_{Repair}$ can be adapted to support different languages.

6.5 Related Work

Vulnerability Detection SAST tools such as CodeQL [Gitb], *CogniCrypt* [Krü+19], and *CryptoGuard* [Rah+19] provide rule-based analyses for security weaknesses, including cryptographic API misuse. Despite their maturity, these tools often suffer from false positives and limited contextual reasoning, requiring manual verification. Recent studies continue to enhance SAST precision through rule refinement and pattern learning, but scalability and maintainability remain open problems limiting their adoption in real-world security pipelines.

Recently, research has investigated whether LLMs can assume a role in vulnerability detection or secure code generation. While Tamberg and Bahsi [TB25] focused on benchmarking detection rates, this study implements an APR pipeline.

Secure Code Generation LLMs have also been explored as tools for producing secure code, yet studies consistently indicate that unguided code generation frequently leads to insecure outputs. While Gong et al. [Gon+25] looked at creating secure code from text, this study deals with the constraints of existing code, requiring the LLM to fix a specific vulnerability without breaking its logic.

Collectively, many studies [Li+26; Mou+25; Sau+25] suggest that LLMs and related generative systems show strong capability for producing syntactically valid code but remain prone to security lapses, reinforcing the necessity of analytical guidance and post-generation validation through static analysis.

Recently, Dolcetti et al. [Dol+26] presented a framework for secure code generation and also repair in C, leveraging testing and static analysis as feedback for self-improvement of the LLMs in the code generation process. Their results show that using an iterative repair process with SAST support increases the correctness and security of generated code, which aligns with our findings.

Automated Vulnerability Repair APR approaches extend detection efforts by attempting to generate fixes for known vulnerabilities. Classical APR techniques employ search-based or constraint-based strategies that iteratively modify program syntax or semantics, typically through mutation, template instantiation, or constraint solving, to produce patches that satisfy all available correctness or test-suite specifications [LPR19]. Recent work by Bui et al. [Bui+23] empirically evaluated nine state-of-the-art APR tools and one vulnerability-specific repair system across 79 real-world Java vulnerabilities in the Vul4J dataset. Their results reveal that existing APR systems can produce testable patches for only about 20% of vulnerabilities; among these, 73% successfully eliminate the vulnerability, yet just 44% preserve program functionality. While Bui et al. [Bui+23] focused on general-purpose Java vulnerabilities using the Vul4J dataset, our study targets the specialized domain of cryptographic API misuse using *CryptoAPI-Bench* [ARY19] and *CamBench* [Sch+22c].

In contrast, learning-based APR techniques leverage historical bug-fix pairs to learn repair patterns, enabling models to generalize from prior examples and predict

plausible patches without exhaustive search. Advancements in deep learning, particularly transformer-based architectures, have significantly improved patch precision and generalization across multiple benchmarks [Zha+24; Fu+22; JAM25]. Recent advances such as RepairLLaMA [SFM25] and AlphaRepair [XZ22] demonstrate the effectiveness of large language models for program repair, improving patch accuracy and generalization beyond traditional template-based methods.

Our approach advances the state of the art by integrating state-of-the-art static analyzers directly into the repair loop. Unlike general APR tools that optimize for passing test suites, our system optimizes for satisfying rigorous cryptographic rule sets defined by these expert static analysis tools.

Hybrid Integration of LLMs and Static Analysis Recent work has shown that combining large language models with static program analysis yields more reliable and semantically consistent repairs than using either approach alone. LLMs generate diverse patch candidates but often miss semantic correctness, while static analysis can ensure determinism, it suffers from scalability and adaptability issues [Gaj+25]. While SecureFixAgent focuses on Python, this study targets Java cryptographic misuses.

6.6 Conclusion and Outlook

In this chapter, we make the fourth main contribution of our thesis, *LLM-SAST$_{Repair}$* (*Contribution 4*), to repair cryptographic API misuses automatically. We use our implementation of *LLM-SAST$_{Repair}$* to perform and contribute a systematic evaluation to assess the capabilities of a representative selection of LLMs (*gpt-4o*, *gpt-4o-mini*, *llama3.3*, *llama3.1*, and *qwen-3.0*) in repairing cryptographic API misuses on their own and with the support of a SAST tool, either *CogniCrypt* [Krü+19] or *CryptoGuard* [Rah+19]. In our evaluation experiments, the two benchmarks *CryptoAPI-Bench* [ARY19] and *CamBench* [Sch+22c] (*Contribution 3*, cf. Section 5.3) were attempted to be automatically repaired. Overall, the combination of the SAST tool *CogniCrypt* and an LLM in *LLM-SAST$_{Repair}$* performed best for both benchmarks, achieving a final security rate *FSR* of 0.772 and a normalized security increase *NSI* of 0.721 for *CryptoAPI-Bench* (*CogniCrypt* with *gpt-4o*) and an *FSR* of 0.889 and an *NSI* of 0.685 for *CamBench* (*CogniCrypt* with *llama3.3*). These results support our initial hypothesis that SAST tools can aid LLMs in automatically repairing security vulnerabilities.

The implementation of *LLM-SAST$_{Repair}$* is adaptable and extensible; the use of any LLM is supported via API keys. Further, the integration of SAST tools uses *SARIF* [OAS20], a standardized format used by many SAST tools, which enables the integration of other SAST tools. Using SAST tools that support other programming languages would also provide support for other languages in *LLM-SAST$_{Repair}$* by adapting the prompts with regard to the languages.

To increase the practical relevance of *LLM-SAST$_{Repair}$*, support for repairing multiple files and usable integrations into the developer's workflow (cf. Section 3.3.5) or tools are needed. One option to achieve repairing a cryptographic API misuse spanning multiple files is making more use of a SAST tool. That SAST tool analyzes a complete codebase. For each warning report, only the relevant code elements, e.g., the affected file, are extracted and provided to *LLM-SAST$_{Repair}$* to repair, and the resulting patch is then reapplied. Therefore, we integrate the repair pipeline developed in this chapter, *LLM-SAST$_{Repair}$* (*Contribution 4*), as the *AI-assisted Code Enhancement* feature (cf. Section 7.4.5) into our final main contribution of this thesis, *SecAI* (*Contribution 5*, cf. Chapter 7), a usability-focused *CogniCrypt*-plugin for *SonarQube*. While Chapter 6 with *LLM-SAST$_{Repair}$* focuses on the automated repair of cryptographic API misuses and its evaluation, Chapter 7 with *SecAI* and its features rather focuses on how features such as the repair pipeline of *LLM-SAST$_{Repair}$* can be presented actionably to developers. The *AI-assisted Code Enhancement* feature is based on *LLM-SAST$_{Repair}$* and addresses the usability category *Fix Support* (cf. Section 3.3.2) discussed in Chapter 3 (*Contribution 1*). Moreover, running the *patch loop* of *LLM-SAST$_{Repair}$* to enhance generated code can be employed to support secure code generation, too. This is integrated as the *Secure Code Generation* feature (cf. Section 7.4.6) in *SecAI*.

Finally, our evaluation showed which combinations of LLM and SAST tool perform well and which perform best. Next steps could include optimizations of the LLMs to achieve even higher final security rates, e.g., by fine-tuning the model [SFM25].

7

SecAI — Supporting Developers in Secure Coding: Systematically Improving SAST Tool Usability Through AI-Enhanced Feedback

In this chapter, we introduce the fifth and final main contribution of this thesis, *SecAI* (*Contribution 5*), which builds on top of all other main contributions and presented publications of this thesis. The goal of this thesis is to *help Java developers reduce cryptographic API misuses*. To achieve this, we present *SecAI*, a usability-focused user interface for *CogniCrypt* [Krü+19], which is implemented using the plugin mechanism of *SonarQube* [Sona], a well-established commercial static analysis tool which is used by over 400,000 organizations and more than 7 million developers [Sone]. The idea is to provide a developer with all usability features they need to remediate reported cryptographic API misuses.

To identify usability features that developers need in static analysis tools, we performed a large-scale study (*Contribution 1*, cf. Chapter 3). The results of this study are used as a systematic approach to build *SecAI*. The collaboration with Amaral et al. [Ama+26] to study SAST tool adoption in industry further aided in the requirements engineering. The study of Java API misuses (*Contribution 2*, cf. Chapter 4) and cryptographic API misuses (cf. Chapter 5) aids in increasing the explainability of *Warning Messages* (cf. Sections 3.3.1 and 7.4.2) and in features such as the *severity score* (cf. Section 7.4.7) based on the risk model introduced in Section 5.1, or the visualization of error trees (cf. Section 7.4.3), making use of the improved analysis of *CogniCrypt*. Further, the benchmark *CamBench* (*Contribution 3*, cf. Section 5.3) was used to train $FP_{Predictor}$ (cf. Section 7.4.8) for false positive prediction and to evaluate $LLM\text{-}SAST_{Repair}$ (*Contribution 4*, cf. Chapter 6) to automatically repair cryptographic API misuses, also integrated as a feature in *SecAI* (cf. Section 7.4.5).

The identified features follow one usability concept behind *SecAI*, which is to guide the developer through the warnings of the static analysis tool, here *CogniCrypt*.

The *Severity Score* (cf. Section 7.4.7) and *False Positive Detection* (cf. Section 7.4.8) can be combined as a *priority score* signaling warning reports that are likely true positives and may have severe consequences; thus, the developer should focus on them. Additionally, *Exact Error Localization* (cf. Section 7.4.1), better explained *Warning Messages* (cf. Sections 3.3.1 and 7.4.2), which, e.g., include information on how to fix the issue, and visualization of interconnected errors (cf. Section 7.4.3), support developers in understanding the issue. *Fix Support* (cf. Section 3.3.2) is provided with *Quick Fixes* (cf. Section 7.4.4) and *AI Fix* (cf. Section 7.4.5).

Overall, *SecAI* functions as a demonstrator of the implemented usability features and will allow for a systematic evaluation in user studies with developers to better understand the respective impact.

7.1 Introduction

Research has uncovered various usability factors and issues related to static analysis tools [CB16; Joh+13; SDM20; NSB22]. Nachtigall et al. [NSB22] (*Contribution 1*, cf. Chapter 3) performed a large empirical study on usability criteria addressed by static analysis tools and thereby also provided a systematic compilation of 36 usability criteria which they group into six categories. Based on these usability criteria and categories, we propose to systematically build a new user interface for an existing static analysis tool in the problem space of cryptographic API misuse detection to show which usability features and improvements can be implemented. We contribute our new plugin ***SecAI*** for cryptographic API misuse detection that emphasizes usability, built on top of an existing static analysis tool for ***Sec***urity while maintaining its warning reports and adding ***AI*** enhancements. We selected *CogniCrypt* [Krü+19], a well-maintained static analysis tool for cryptographic API misuse detection, to build *SecAI*.

To support developers, research has shown that *workflow integration* is key [CB16; NSB22]. Therefore, we integrated *SecAI* into *SonarQube* [Sonb], a widely used static analysis tool in industry, leveraging *SonarQube's* plugin platform [Sona] to integrate *SecAI* into developers' existing workflows. This decision is further supported by the findings of Amaral et al. [Ama+26], who investigated the adoption of SAST tools in industry. In their focus group sessions, developers were interested in the reports of *CogniCrypt* and *CryptoGuard*; they were even surprised that cryptographic API misuses were reported that their commercial tool missed. While developers were interested in the benefits of using *CogniCrypt* or *CryptoGuard*, they made clear

that such a tool needs to be integrated into their existing toolchain to be even considered.

Among other aspects, our implementation *SecAI* aims to improve the clarity and usefulness of warning messages, provide concrete solutions for fixing detected issues, and enhance visualization of vulnerabilities to help developers quickly grasp their location and impact within the code. By combining precise detection of misuses accompanied by actionable solutions, *SecAI* aims to reduce developer effort while fostering the effective remediation of cryptographic misuses. Moreover, by integrating artificial intelligence techniques and leveraging the complementary strengths of *CogniCrypt* and *SonarQube*'s platform [Sonb], *SecAI* aims to provide an effective and developer-friendly solution for detecting cryptographic vulnerabilities.

7.2 Foundations

To understand the design choices behind *SecAI*, it is essential to outline the theoretical and technical foundations on which the tool is built. Building on the background presented in Chapter 2, this section introduces some core concepts of our approach, including static analysis tools and artificial intelligence.

7.2.1 Approach of *SecAI*

Our goal is to enhance the usability of SAST tools. Therefore, we follow a systematic approach to build usability features based on the empirical work of Nachtigall et al. [NSB22], providing 36 usability criteria addressed by static analysis tools grouped in six categories (cf. Chapter 3). We build a new user interface for the existing SAST tool *CogniCrypt* [Krü+19] for cryptographic API misuse detection as a representative case. One pivotal aspect highlighted by, e.g., Nachtigall et al. [NSB22] and Christakis and Bird [CB16] is *workflow integration*. *CogniCrypt*, for instance, is used via a command-line interface (*CLI*), which is often cumbersome to integrate into such workflows. The existing Eclipse plugin has not been maintained and therefore no longer works reliably [Coga], whereas the analysis of *CogniCrypt* has been further developed and refactored. Hence, we decided to build a new plugin. To achieve better workflow integration for developers, after rigorous evaluation, we decided to use *SonarQube's* plugin platform [Sonb] since *SonarQube* is a well-established and widely used static analysis tool that is seamlessly integrated into CI/CD pipelines and DevOps workflows. It is used by over 400,000 organizations and more than 7

million developers [Sone]. Amaral et al. [Ama+26] have also shown in their study on SAST tool adoption in industry that conforming with existing tool pipelines, such as *SonarQube*, is important for developers' consideration of tools. The platform of *SonarQube* also provides a mechanism to create custom plugins [Sona], which proved useful when integrating our approach.

7.2.2 *CogniCrypt*

Within *SecAI*, we rely on *CogniCrypt* [Krü+19] (cf. Section 2.2.2), a specialized Static Application Security Testing (*SAST*) tool for Java applications. In Section 5.2, we adapted *CogniCrypt* to reason about error chains. Since then, we have further developed *CogniCrypt* so that it supports additional analysis frameworks[1] in addition to *Soot* [Val+10]; these are *OPAL* [Hel+20] and *SootUp* [Kar+24]. In *SecAI*, we use *CogniCrypt* with *SootUp*; thus, it performs its analysis on the intermediate representation *Jimple* [Kar+24] generated by *SootUp*. The analysis itself is configured with rules in the domain-specific language *CrySL* [Krü+19] (cf. Section 2.2.2), which is used to specify the secure usage of cryptographic Java APIs. *CrySL* rules comprise several sections, such as `CONSTRAINTS` for specifying constraints on parameters for different methods.

7.2.3 *SonarQube*

SonarQube [Sonb] is a well-established and widely used static analysis tool that automates code quality and security reviews; however, it reports different warnings than *CogniCrypt*. It checks code against a comprehensive set of rules covering attributes such as maintainability, reliability, and security. Automated analysis and reviews can be integrated at every stage of the development process to support continuous quality assurance. Furthermore, *SonarQube* is extensible via the *SonarQube* Plugin API [Sona], which is Java-based and allows the development of custom plugins. This enables specialized tools, such as our *SecAI* tool, to be seamlessly integrated into the *SonarQube* ecosystem [Sonb].

[1] Support for multiple frameworks was released in version 4.2.0: `https://github.com/CROSSINGTUD/CryptoAnalysis/releases/tag/4.2.0` (last accessed: 2026-06-06)

7.3 *SecAI — SonarQube*-plugin Design and Implementation

For our *SecAI*-plugin, we use *CogniCrypt* (CryptoAnalysis[2]) in version 5.0.1 [Krü+19] and *SonarQube* Community version v25.2.0 [Sonb]. *SonarQube* operates by running different *sensors* on a project. These are essentially small analysis units with a specific focus that analyze the source code and report the results to the *SonarQube* instance. The *SonarQube* Community Edition, for instance, includes a general *JavaSensor*. Through *SonarQube's* plugin mechanism, we were able to create our own *sensor* that then runs *CogniCrypt* on the project. Since *CogniCrypt* analyzes Java archive (*JAR*) files, the first step in our *sensor* is to build the project using one of the most common build tools, Maven [Apab] or Gradle [Gra]. *SecAI* automatically detects the build tool to be used but allows users to manually select it and provide, e.g., a necessary execution path via the setup menu integrated in *SonarQube's* GUI.

After generating the JAR, our *sensor* runs the *CogniCrypt* analysis itself. The resulting errors are then reported to *SonarQube*, making use of *CogniCrypt's* Static Analysis Results Interchange Format (*SARIF*) report [OAS20], a standardized JSON format supported by many static analysis tools. The platform requires that all issues are tied to a *SonarQube* rule, which provides a static description of the problem. To match the *CogniCrypt* rules precisely, we created custom *SonarQube* rules [Sond] corresponding to the *CrySL* rules of *CogniCrypt*. Since the platform did not have rules that fit our use case, we formulated a custom set of rules specifically for *CogniCrypt* using *SonarQube's* plugin mechanism (cf. Section 7.4.2).

Limitations in the *SonarQube* Integration

While using *SonarQube* [Sona] for our plugin has advantages as an established platform, including workflow integration for the existing developer user base, the native *SonarQube* [Sonb] issue user interface limits what information can be displayed to established API endpoints. Therefore, not all features (cf. Section 7.4) can be directly integrated into the native interface. However, *SonarQube's* plugin mechanism also allows adding custom web pages, which we use to provide our custom *SecAI* view.

[2] `https://github.com/CROSSINGTUD/CryptoAnalysis/releases/tag/5.0.1` (last accessed: 2026-06-06)

7.4 Features of *SecAI*

Integrating a SAST tool such as *CogniCrypt* [Krü+19] into *SonarQube* [Sonb] already improves its usability by embedding it into the development workflow and making it available cross-platform [CB16; NSB22]. Using the native *SonarQube* issue interface already supports many usability aspects. However, based on the work of Nachtigall et al. [NSB22] (cf. Chapter 3), we identified a need for more features, some of which require an additional view within the *SonarQube* plugin.

Nachtigall et al. provided 36 usability criteria grouped in six categories, namely: *Warning Messages*, *Fix Support*, *False Positives*, *User Feedback*, *Workflow Integration*, and *User Interface*. Table 7.1 summarizes the features we identified to fulfill most usability criteria from Nachtigall et al. [NSB22]; we added *Secure Code Generation* as an additional feature that does not correspond to one of the categories. For *SecAI*, we mapped each feature directly to the corresponding usability criterion and its category (cf. more details in the `repository`, cf. Chapter 9). The category *Workflow Integration* is a special case, as it is already mostly fulfilled by the integration into *SonarQube*.

Table 7.1: Mapping between *SecAI* features and usability categories [NSB22] (all are covered). Direct mapping on criterion level can be checked in the `repository`.

Section	Feature	Usability Categories
7.4.1	Exact Error Localization	*Warning Messages* (3.3.1), *User Interface* (3.3.6)
7.4.2	Error Message and Description	*Warning Messages* (3.3.1), *Fix Support* (3.3.2), *User Interface* (3.3.6)
7.4.3	Visualization of Interconnected Errors	*Warning Messages* (3.3.1), *Fix Support* (3.3.2), *User Interface* (3.3.6)
7.4.4	Quick Fixes	*Fix Support* (3.3.2)
7.4.5	AI-assisted Code Enhancement	*Fix Support* (3.3.2)
7.4.6	Secure Code Generation	-
7.4.7	Severity Score	*Warning Messages* (3.3.1), *User Feedback* (3.3.4), *Workflow Integration* (3.3.5), *User Interface* (3.3.6)
7.4.8	False Positive Detection	*False Positives* (3.3.3)

The features are implemented to be functional enough to showcase their effects in terms of usability and, in the future, evaluate them in user studies with developers. Further evaluation and optimization are encouraged.

7.4.1 Exact Error Localization

Generally, code analysis aims to find issues and provide developers with the information to fix them. Nevertheless, to resolve the discovered errors, developers need to know the exact location [NSB22] (cf. Section 3.3.1). However, *CogniCrypt* [Krü+19] only returns the starting line of the violation. This can lead to significant additional effort for a developer to locate an error. For instance, Listing 7.1 shows several errors occurring within the same statement, since multiple insecure parameters are used for the `init` method call (cf. Lines 240 to 247) and all are reported for Line 240. For this reason, we attempt to determine a more accurate location before reporting the error.

We run *CogniCrypt* with *SootUp* [Kar+24] as the static analysis framework, which uses the intermediate representation *Jimple* [Kar+24] of the original source code. As the analysis report includes the violating *Jimple* statement, we can extract the method call that causes the violation. After retrieving the actual code snippet from the source code, we locate the violating method call and its parameter list. Lastly, the exact location can be highlighted according to the error type and additional information from the error message.

Listing 7.1: Insecure example code snippet with multiple errors spread over several lines, but only reported for one line.

```
230  public ErrorLocationDemo() throws InvalidAlgorithmParameterException,
        NoSuchPaddingException, NoSuchAlgorithmException, InvalidKeyException {
231    demo(5, 10); //insecure parameters (Line 244)
232  }
233
234  public void demo(int prime, int exp) throws NoSuchAlgorithmException,
        NoSuchPaddingException, InvalidKeyException, InvalidAlgorithmParameterException {
235    KeyPairGenerator kpGen = KeyPairGenerator.getInstance("DH");
236    kpGen.initialize(2048); // insecure keysize (Line 242)
237
238    byte[] seed = "static seed".getBytes(); // insecure seed (Line 246)
239    KeyAgreement keyAgreement = KeyAgreement.getInstance("DH");
240    keyAgreement.init(
241    // this method call is split over multiple lines
242    kpGen.generateKeyPair().getPrivate(),
243    // we add comments to show that the location
244    new DHGenParameterSpec(prime, exp),
245    // reported by CogniCrypt is not very accurate
246    new SecureRandom(seed)
247    );
248  }
```

The reported issue can be viewed in *SonarQube's* [Sonb] web interface, where the code snippet is shown with the exact error location highlighted. An example of this can be seen in Figure 7.1, where ① points to the main location of an issue for the code from Listing 7.1. From this view, the error can also be opened in an Integrated

Development Environment (*IDE*) (cf. ② in Figure 7.1) if an IDE is bound to the project via the *Connected Mode* [Sonf] of the plugin *SonarQube for IDE* [Sonc]. The IDE jumps to the issue location and opens the *SonarQube for IDE* tab to show the rule definition.

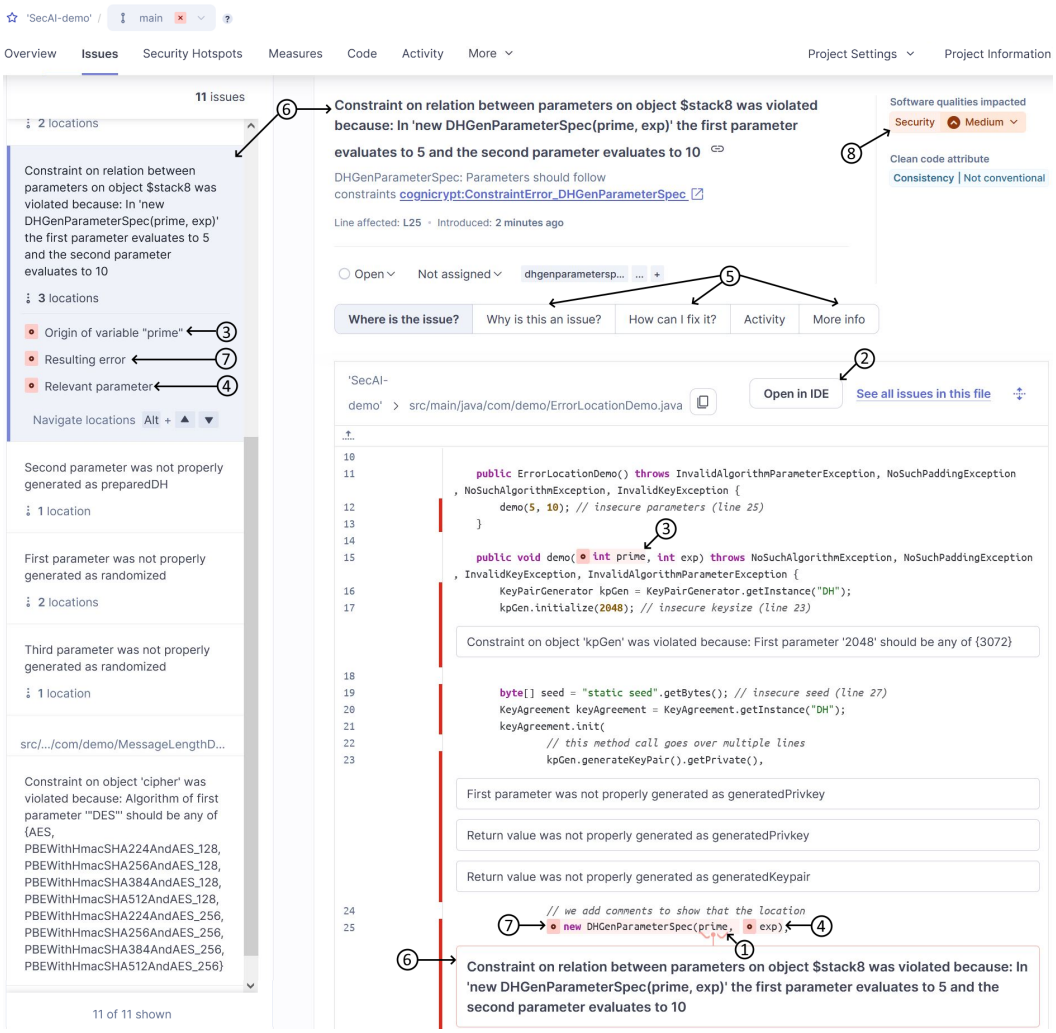


Figure 7.1: *SonarQube* issue interface displaying an example issue. Circled markers, e.g., ③, can occur multiple times.

When reporting an issue to *SonarQube*, it is possible to include supplementary locations to provide more information about the error. In the *SonarQube* issue interface (cf. Figure 7.1), additional locations are listed below the error message in the navigation list on the left and highlighted in the displayed code snippet. *SecAI* utilizes this to report additional information (cf. ③, ④, ⑦ in Figure 7.1).

7.4.2 Error Message and Description

To fix an issue, it is not enough to give the developer the location of the error; they also need to know what went wrong [CB16]. As mentioned in Section 7.2, *SonarQube* [Sonb] requires that all reported issues are tied to a specific rule, which then contains a static description of the problem, possible solutions, and, when applicable, references to additional resources. In the *SonarQube* issue interface, these sections can be accessed through the tabs *Why is this an issue?*, *How can I fix it?*, and *More Info* (cf. ⑤ in Figure 7.1). The same tabs are also shown when opening an issue in an IDE. For consistency, the same tabs were added to our custom *SecAI* view (cf. Figure 7.2), though here *Why is this an issue?* was renamed to *Root Cause* and also contains the affected code snippet.

Since our plugin integrates *CogniCrypt* [Krü+19], we created a set of custom rules based on the *CrySL* rules [Krü+19]. However, one *CrySL* rule can result in many different errors, so we split each of the 51 *CrySL* rules section-wise according to the seven error types it can detect. Some sections, namely *SPEC*, *OBJECTS*, *EVENTS*, and *ENSURES*, were left out as they were not directly related to the violations. An outlier is the error type *ImpreciseValueExtractionError*, which only received one single *SonarQube* rule because the description of this error amounts to the analysis being unable to evaluate some parameter and therefore also incapable of verifying whether there is a problem or not.

In total, 141 custom *SonarQube* rules were created for this *CogniCrypt* integration. Each one includes a small explanation of the error type to explain in the *Why is this an issue?* tab, a natural language explanation of the *CrySL* rule section relevant to this error type as a guide in the *How can I fix it?* tab, and links to the official Java documentation and the Common Weakness Enumeration (*CWE*) references [Mita] associated with the *CrySL* rule in the *More Info* tab.

Outside of the static rule descriptions, *SonarQube* also uses dynamically definable error messages (cf. ⑥ in Figure 7.1, ① in Figure 7.2). *CogniCrypt* also returns error messages. However, its messages do not always fit the format of *SonarQube*. Editing the messages to match the format and extensively shorten them includes these steps:

1. Replacing Jimple references with the original Java references if possible.

2. Removing location information because *SonarQube* already displays this.

3. Shortening fully qualified class names to just class names.

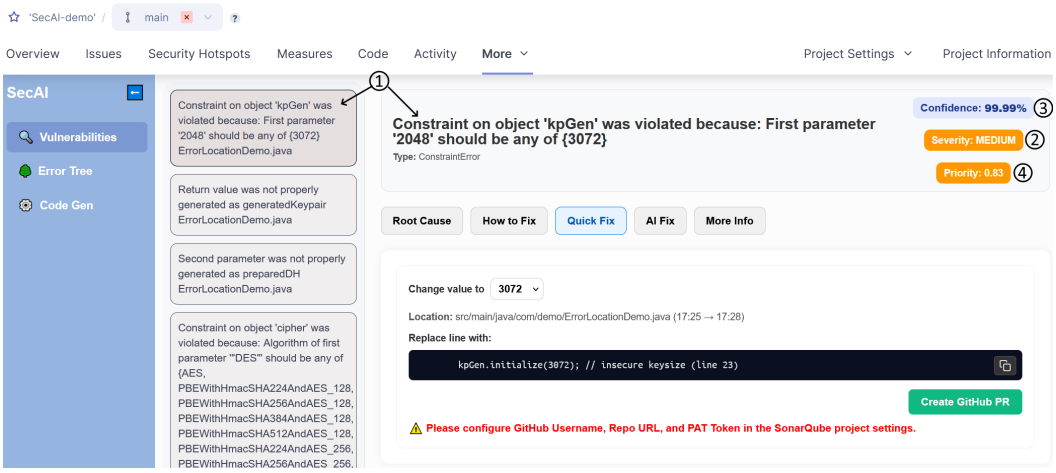


Figure 7.2: Custom *SecAI* error view.

4. Collapsing (sometimes very long) method call lists.

5. Summarizing explanations for *ConstraintErrors*.

7.4.3 Visualization of Interconnected Errors

One major feature provided by *CogniCrypt* [Krü+19] is the ability to detect connections between errors [Wic+24] (cf. Section 5.2). However, currently, the only way to get this information is to request a PNG file of the resulting error tree when passing arguments to the command line. In *SecAI*, we integrated error chains in both the native *SonarQube* [Sonb] issue interface and our custom view.

When reporting an issue to *SonarQube*, it is possible to add multiple additional locations (cf. Section 7.4.1). By including the main locations of all preceding or subsequent errors (cf. ⑦ in Figure 7.1), navigating between different, connected error locations is made easier for the user. The locations are listed in the navigation list on the left and highlighted in the code.

In our custom interface, we build an interactive, visual representation of the error tree (cf. Figure 7.3) where errors are grouped by the error type. When hovering over a node, a tooltip is shown which includes the class the error occurs in, the code snippet, and the error message. This tooltip also contains a shortcut to the issue's detail page within our own custom view and a button that calls the *AI Fix* covered in Section 7.4.5. Clicking the node highlights the edges leading up to the selected node and the ones below it, though it can be customized to highlight only one direction. Errors can be filtered by the originating class.

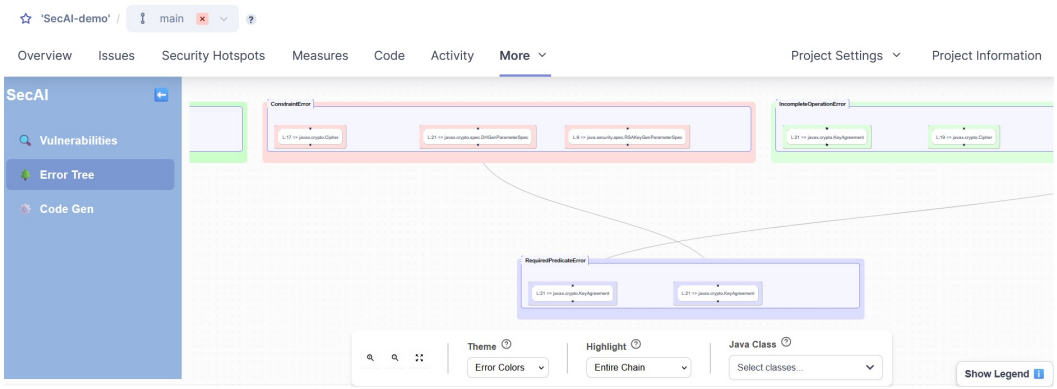


Figure 7.3: Interactive error tree within the custom *SecAI* view.

7.4.4 Quick Fixes

Not all issues require complex resolutions; in terms of usability, it is reasonable to provide the user with quick fixes [CB16]. Using *CogniCrypt* [Krü+19], for a lot of error types the data needed for such quick fixes can be extracted directly from the error message, e.g., for the error in Line 236 in Listing 7.1, the value `2048` needs to be changed to `3072` to be secure.

Unfortunately, while *SonarQube* [Sonb] as a platform does support quick fixes, it does not allow custom plugins to add any. Instead, we have added the information to our custom view, as can be seen in Figure 7.2. Here, the user can select one of the secure values from a dropdown, and the modified line is shown. The user can either copy it to the clipboard or create a GitHub [Gitc] pull request for this *fix* via the GitHub account specified in the project settings. Thus, we recommend providing a designated GitHub account for this. The *Quick Fix* tab only appears when there are actually fixes available.

7.4.5 AI-assisted Code Enhancement

Software vulnerabilities are often complex cryptographic API misuses [Sch+22a; Wic+24] (cf. Chapter 4 and Section 5.2) which cannot be resolved with simple, pre-defined quick fixes (cf. Section 7.4.4). To address this, we implement our version of the approach of Kavian et al. [Kav+24] combining LLMs and SAST tools to incorporate an AI-assisted Code Enhancement (*AI Fix*) providing targeted, verifiable code remediation. This feature is based on *LLM-SAST*$_{Repair}$ [Sch+26b] (*Contribution 4*, cf. Chapter 6) and makes use of its repair pipeline.

AI Fix can be invoked via its tab within our custom error view (cf. Figure 7.2) or from the node tooltip in the error tree. On user demand, *SecAI* invokes the *AI Fix* process with the respective vulnerable code snippet, the violated *CrySL* rule [Krü+19], and the resulting error message. It then generates a candidate fix and iteratively verifies it with *CogniCrypt* [Krü+19]. This *AI Fix* process creates a closed loop between generating a fix candidate and verification: The LLM-generated code is compiled and analyzed. To improve the performance of the LLM, *SecAI* incorporates Retrieval-Augmented Generation (*RAG*), which enriches prompts with external domain-specific knowledge [Ant+25; Zha+26]. If *CogniCrypt* reports violations, the system uses the structured feedback as well as *CrySL* rules as part of our *RAG* to refine the code until all issues are resolved or a pre-defined loop iteration limit is reached. The resulting *fix* is shown in an additional tab with a diff view and indicates `CogniCrypt Verified` if the fix passed the final scan. As for the *Quick Fix* (cf. Section 7.4.4), a GitHub [Gitc] pull request can be created by pressing a button.

AI Fix uses Ollama [Oll] to integrate LLMs, making it model-agnostic. We used OpenAI [Ope] and Gemini [Alp23], using sensible defaults and low-temperature settings to ensure deterministic and reliable fix generation across engines.

7.4.6 Secure Code Generation

In *SecAI*, we introduce the feature *Secure Code Generation* which uses *AI Fix* (cf. Section 7.4.5), but instead of improving existing code, code is first generated with an LLM via a user prompt, and then *AI Fix* is applied. This feature is based on *LLM-SAST*$_{Repair}$ [Sch+26b] (*Contribution 4*, cf. Chapter 6) and makes use of its repair pipeline. The final output contains: (*1.*) A complete, standalone Java class named `Demo`, (*2a.*) a `CogniCrypt Verified` boolean indicating that the code has been verified by *CogniCrypt* [Krü+19] and no violations were found, or (*2b.*) if an issue persists, the final *CogniCrypt* report is appended as a block comment. As developers generally struggle with using cryptographic APIs [Nad+16], this could be a useful feature when adding new elements to their project.

7.4.7 Severity Score

Providing developers with a severity value can support them in deciding which issues they prioritize to fix. We use a risk model (cf. Table 5.1) proposed by Wickert et al. [Wic+22] (cf. Section 5.1) which provides a rough categorization into three severities specific to vulnerabilities that *CogniCrypt* [Krü+19] is able to detect. We

directly map specific sections of *CrySL* rules [Krü+19] to five different levels that conform to *SonarQube's* score of *Info*, *Low*, *Medium*, *High*, and *Blocker*. However, in a few cases, we default to the severity *Medium* as an approximation to showcase the feature.

In *SonarQube*, the impact of issues is graded in three aspects, also referred to as software qualities: *Security*, *Reliability*, and *Maintainability*. For each quality, a separate severity can be set. As *CogniCrypt* [Krü+19] is a security-specific analysis tool, we reported our severity score as the impact on the software quality *Security* (cf. ⑧ in Figure 7.1). The severity score is also included in our custom error view, where it can be seen at ② in Figure 7.2.

7.4.8 False Positive Detection — $FP_{Predictor}$

High false positive rates waste developer effort, reduce trust in the tool, and discourage adoption [CB16]. To address false positives, Bodden et al. [BLH08] suggested using machine learning to reduce false positives from their analysis report, and Tripp et al. [Tri+14] followed this approach and trained a model for a JavaScript commercial static analysis tool to filter warning messages. For *SecAI*, we implement and train our own model, $FP_{Predictor}$ [OSB26b], which we run in our pipeline for false positive prediction (cf. Figure 7.4). In this section, we provide an overview; more details are published in [OSB26b]. We aim to help developers focus on fixing real cryptographic API misuses by automatically differentiating between false and true positives.

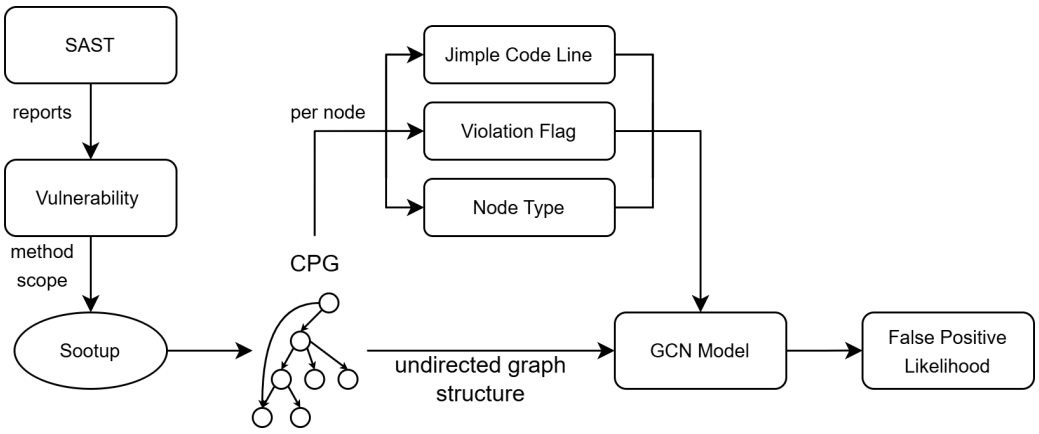


Figure 7.4: Pipeline for predicting whether a warning report is a false positive.

Our approach is based on a Graph Convolutional Network (GCN) model [Wu+21] that utilizes Code Property Graphs (CPGs) [Yam+14], which represent the structural and semantic relationships within the code. We further use a trained Word2Vec

model [Mik+13] on the *Jimple* code, stored as the label attribute of each CPG node, to represent words in a continuous vector space where semantic and syntactic relationships are preserved. Node types are encoded using a one-hot scheme, and as shown in Figure 7.4, each node is thus assigned three attributes: vectorized Jimple code, encoded node types, and violation flag. These attributes form the input to a GCN [Wu+21], and the GCN processes the graph and produces an output via a sigmoid activation function, which we interpret as the likelihood that a reported violation is a false positive in the range $(0, 1)$ [Goo+16].

For model training, we used the benchmark *CamBench* [Sch+22c] (cf. Section 5.3). During evaluation on a single-file subset of *CryptoAPI-Bench* [ARY19], $FP_{Predictor}$ achieved an overall accuracy of 96.6% in classifying warning reports; all experiment data is provided in an artifact [OSB26a].

Integration of $FP_{Predictor}$ in *SecAI*

In *SecAI*, we receive an error report from *CogniCrypt* [Krü+19] from which we generate the CPG, leveraging *SootUp* [Kar+24]. Within the custom view of the *SecAI* plugin, the output of the false positive model is inverted to display the *confidence score* c, i.e., $c = 1 - fp$ (fp denotes the false positive likelihood), representing the likelihood of the report being a true positive. The *confidence score* is displayed in the user interface (cf. ③ in Figure 7.2), positioned at the top-right corner. Beneath it, the *priority score* p (cf. ④ in Figure 7.2) is shown, which combines the *confidence score* c and the encoded *severity score* s of each vulnerability (Section 7.4.7).

Severity levels are encoded as follows: *Info* = 0.1, *Low* = 0.33, *Medium* = 0.66, and *High* = 1. To calculate a *priority score*, we use weights w_1 and w_2, both set to 0.5 by default, such that $p = w_1 * c + w_2 * s$ stays within the boundaries $[0, 1]$.

Finally, with the *confidence score* and *priority score* in addition to the *severity score* available, developers are better enabled to prioritize which vulnerability to address first. However, due to limitations of the *SonarQube* plugin framework [Sona], we can only visualize these scores in the custom *SecAI* view.

7.5 Results

The goal of building *SecAI* as a *CogniCrypt*-plugin [Krü+19] for *SonarQube* [Sonb] is to provide a user interface with features (cf. Table 7.1, Section 7.4) systematically

addressing the usability criteria collected by Nachtigall et al. [NSB22]. In this section, we discuss the achieved feature fulfillment and the performance of running *CogniCrypt* via CLI in comparison to our *SonarQube*-plugin.

7.5.1 Feature Fulfillment and Satisfaction

In Table 7.2, we show our technical self-assessment based on the procedure discussed by Nachtigall et al. [NSB22] (cf. Chapter 3) of features, their satisfaction, and fulfillment reported on the criterion level. For each feature, the number of planned criteria and actually fulfilled criteria is shown; for example, if a criterion was only partially fulfilled, it is not counted in Table 7.2. Overall, we planned to address all of Nachtigall et al.'s [NSB22] usability categories and managed to fulfill 32 of the 36 criteria in the technical self-assessment; the remaining criteria were not fully satisfied due to integration limitations.

Table 7.2: Overview of technical self-assessment of *SecAI*. Satisfaction of planned features. ● denotes all planned or more criteria were fulfilled, and ◐ denotes at least multiple usability criteria are fulfilled. More detailed description and scoring per usability criteria [NSB22] is provided in the `repository`.

Feature	Satisfaction	fulfilled / planned criteria
Workflow Integration	●	fulfilled by *SecAI*-plugin
Exact Error Localization	●	4 / 4
Error Message and Description	◐	13 / 14
Visualization of Interconnected Errors	●	6 / 4
Quick Fixes	◐	3 / 4
AI-assisted Code Enhancement	●	2 / 2
Secure Code Generation	●	*no matching criterion*
Severity Score	◐	6 / 7
False Positive Detection	◐	3 / 7
***SecAI*-plugin**	**Overall:**	**32 / 36**

Limitations

While every feature was successfully developed and integrated, two of them were limited by our use of *SonarQube* [Sona] as the underlying platform. First, although *SonarQube* natively supports *Quick Fixes* (Section 7.4.4), its plugin architecture does not permit the integration of custom quick fix functionality. Second, *False Positive Detection* (Section 7.4.8) faced structural limitations. The *confidence score* and *priority score* are not included in the analysis report of *CogniCrypt* and thus

cannot be shown in the native *SonarQube* interface. Therefore, the *confidence score* and *priority score* are computed via an API service within the frontend. This allows for filtering based on these scores only in the custom *SecAI* view.

7.5.2 Performance Comparison of *CogniCrypt* and *SecAI*

Integrating *CogniCrypt* [Krü+19] into a *SonarQube*-plugin [Sona] was challenging and came with compromises (cf. Section 7.3), e.g., creating 141 custom *SonarQube* rules (cf. Section 7.4.2) for the 51 *CrySL* rules. To compare the performance of *CogniCrypt* run via CLI and our *SecAI*-plugin, we ran both five times on relevant open-source benchmarks, namely *CryptoAPI-Bench* [ARY19] and *CamBench* [Sch+22c], as shown in Table 7.3. We executed all analysis runs on the same machine (AMD Ryzen 5 5600G processor, 32 GB RAM, RTX 3060 (12 GB) graphics card, Windows 10 64-bit OS) running semantically equivalent rulesets.

Table 7.3: Average runtime comparison between running *CogniCrypt* [Krü+19] release 5.0.1 over five runs each from the CLI and with our custom *SecAI*-plugin for two benchmarks.

	***CryptoAPI-Bench* [ARY19]**			***CamBench* [Sch+22c]**		
	Analysis time	Call graph construction time	Typestate analysis time	Analysis time	Call graph construction time	Typestate analysis time
CLI	1.441 s	1.491 s	0.750 s	6.914 s	1.846 s	3.624 s
SecAI	1.724 s	0.517 s	0.923 s	9.407 s	1.276 s	4.836 s

Overall, we can see that when running *SecAI*, the average total *Analysis time* increased by about 20% for *CryptoAPI-Bench* and 36% for *CamBench*. However, on average, we see a reduction in *Call graph construction time* and an increase in *Typestate analysis time*. After inspection, we attribute the variation in timings to different versions of the Java Runtime Library available in the *SonarQube* [Sona] instance compared to the CLI execution, e.g., resulting in a different class hierarchy and thus, a different call graph.

Limitations

Encapsulation in the plugin leads to issues, e.g., in finding even basic Java classes such as `java.lang.Object` to resolve concrete methods. In *SecAI*, the constructed call graph has far fewer edges compared to the analysis via CLI and can therefore be built more quickly (cf. Table 7.3). This also prevented the analysis from resolving

the *CrySL* rule [Krü+19] for `javax.servlet.http.Cookie`. For fair comparison, this rule was excluded from the experiment. However, in practice, this can result in undetected vulnerabilities.

7.6 Conclusion

In this chapter, we make the final main contribution of this thesis, *SecAI* (*Contribution 5*), a tool integrated into the established platform *SonarQube* [Sonb] that implements eight features (cf. Table 7.2) designed to systematically enhance the usability of SAST tools such as *CogniCrypt* [Krü+19], which we used as the analysis tool. In comparison to the 46 tools analyzed in the usability study by Nachtigall et al. [NSB22] (*Contribution 1*, cf. Chapter 3), our approach is, to our knowledge, the first that systematically fulfills all main usability categories and most criteria.

With the *Visualization of Interconnected Errors* (cf. Section 7.4.3), we provide the first interactive user interface for error chains introduced by Wickert et al. [Wic+24] (cf. Section 5.2). This feature allows us to study further the usability implications of presenting error chains to developers.

In describing and implementing the features (cf. Section 7.4), which we identified as essential to provide support and fulfill usability categories and criteria [NSB22] (*Contribution 1*, cf. Chapter 3), we provide guidance and examples on how to build usable interfaces for static analysis tools. Furthermore, by addressing current developments in artificial intelligence in different features, especially *AI-assisted Code Enhancement* based on *LLM-SAST*$_{Repair}$ [Sch+26b] (*Contribution 4*, cf. Chapter 6), *Secure Code Generation*, and *False Positive Detection* [OSB26b] (cf. Section 7.4.8), we also introduce a new feature that cannot be mapped to any of Nachtigall et al.'s usability criteria.

Overall, we have introduced *SecAI* as a demonstrator and discussed the technical implementation and self-assessment of its usability features. However, its practical impact needs to be assessed, e.g., by conducting a dedicated usability study with developers to empirically evaluate the practical impact.

8 Conclusion and Outlook

The goal of this thesis is to *help Java developers reduce misuse of cryptographic APIs*. To achieve this goal, we aimed to improve tooling support through static analysis, which led to several challenges and tasks. As described in Section 2.1, at the beginning of this thesis, fundamental elements needed further investigation to develop tooling with increased support. These elements included understanding: *(i)* the foundations of usability of static analysis tools, *(ii)* the origins of API misuses, e.g., to explain them better, and *(iii)* domain-specific benchmarking to assess a static analysis tool's capabilities and, e.g., adapt its analysis algorithm.

In this thesis, we made contributions that provide the foundations for the final development of *SecAI* (*Contribution 5*, cf. Chapter 7), a new, usability-focused user interface for the existing static analysis tool *CogniCrypt* [Krü+19], which we have maintained and further developed throughout the research effort for this thesis. The systematic approach to derive the features and design of *SecAI* is based on *(i)* the large-scale study of usability criteria addressed by static analysis tools [NSB22] (*Contribution 1*, cf. Chapter 3) and the contributions made to the study of SAST tool adoption in industry by Amaral et al. [Ama+26]. The *(ii)* studies of Java API misuses in general [Sch+22a] (*Contribution 2*, cf. Chapter 4) and specifically in cryptography (cf. Chapter 5) were essential to increase the explainability of *Warning Messages* (cf. Section 3.3.1). We contributed the benchmark *(iii) CamBench* [Sch+22c] (*Contribution 3*, cf. Section 5.3) for the domain of static analysis tools for cryptographic API misuse detection.

Based on these foundational contributions to achieve the goal of this thesis, we developed new tooling. *CamBench* was used to evaluate *LLM-SAST*$_{Repair}$ [Sch+26b] (*Contribution 4*, cf. Chapter 6) and to train *FP*$_{Predictor}$ [OSB26b] (cf. Section 7.4.8), both of which are integrated as features in *SecAI*. More features, such as *Severity Score* (cf. Section 7.4.7) and *Visualization of Interconnected Errors* (cf. Section 7.4.3), were developed based on the two publications to which we contributed as a co-author, presented in Section 5.1 and Section 5.2, respectively. Presenting the implementation of *SecAI*, we aim to provide developers with a technical solution that better supports them in remedying Java cryptographic API misuse. So far, *SecAI* functions as a demonstrator that, in the future, will allow evaluating individual

usability features, feature combinations, and all features as full applications. This evaluation will require user studies with industry developers, which were out of scope for this thesis.

Developing *SecAI*, we learned that for building well-designed, usable analysis tools, a holistic approach is necessary. It is not sufficient to develop a sophisticated analysis algorithm without providing a usable interface that allows developers to act on analysis results; otherwise, even true positive reports may be ignored and thus turn into *effective false positives* [Sad+15] (cf. Section 2.2.3). In Chapter 3 (*Contribution 1*) we saw that static analysis tools that only provided a command-line interface generally performed worse in our evaluation than tools with a graphical user interface. Further, it is also insufficient to build beautiful, usable graphical user interfaces if the underlying analysis does not yield significant or relevant results. Therefore, it is also important to design the static analysis to address the needs that the static analysis tool, including its user interface, is supposed to address.

For instance, when we identified *error chains* to be common when using Java cryptographic APIs [Wic+22; Wic+24] (cf. Sections 5.1 and 5.2) because of the underlying design concept of *multi-object protocols* [Sch+22a] (*Contribution 2*, cf. Chapter 4), we had the idea to add a usability feature to *CogniCrypt* to visualize them. However, developing this feature required significant effort. First, we had to understand the issue and adapt the analysis to reason about error chains [Wic+24] (cf. Section 5.2). Once we achieved this, we learned that such interconnected errors are not limited to error chains but also appear as error trees (cf. Figure 5.2), which also require a well-designed graphical user interface feature. An expert interview (cf. Section 5.2.5) at this point indicated that providing developers with information about *root errors* in error chains helps them remediate relevant errors first. Finally, in *SecAI* [Sch+26c] (*Contribution 5*, cf. Chapter 7), we make use of *CogniCrypt's* adapted analysis to extract interconnected errors from the analysis report and present them in the *Visualization of Interconnected Errors* (cf. Section 7.4.3) feature.

Overall, in this thesis, we make multiple main contributions (cf. Chapters 3, 4, 6 and 7 and Section 5.3) that lay the foundation to systematically build usable static analysis tools and show this by systematically developing *SecAI* (*Contribution 5*, cf. Chapter 7), a new, usability-focused user interface for *CogniCrypt*, as the final main contribution, which is composed of all the previous ones. Future work may build on this to further evaluate usability features of static analysis tools, improve existing tools, and develop new tools. Finally, this may further help Java developers reduce misuse of cryptographic APIs.

9 Implementations and Data

While conducting the research presented in this thesis, we have created several datasets and prototypical implementations. We made these data and implementations available to enable other researchers to reproduce and extend our results. All projects contain detailed documentation and instructions to validate our results. An overview is shown in Table 9.1.

Table 9.1: Overview of implementations, datasets, benchmarks, and supplementary artifacts created in the context of this thesis.

Contribution	Chapter	Artifact	Availability
Contribution 1	Chapter 3	Usability study	figshare
Contribution 2	Chapter 4	*FUM* case study	figshare
–	Section 5.1	Empirical study	figshare
–	Section 5.2	*CogniCrypt* extension	GitHub
		Empirical study	figshare
Contribution 3	Section 5.3	*CamBench*	GitHub
Contribution 4	Chapter 6	*LLM-SAST*$_{Repair}$	figshare
Contribution 5	Chapter 7	*SecAI*	GitHub
	Section 7.4.8	*FP*$_{Predictor}$	figshare

Contribution 1, Usability Study — Chapter 3

In Chapter 3, we presented a large-scale tool survey of usability criteria addressed by static analysis tools. The results were summarized in Table 3.2. This artifact contains information on *all collected tools*, *tools by programming language*, *evaluation criteria*, and *all evaluation details*; it is available on figshare.

Usability Study

`https://doi.org/10.6084/m9.figshare.32594580`

Contribution 2, *FUM* — Chapter 4

In Chapter 4, we performed a case study (cf. Table 4.1) on *FUM* types that can be specified and detected with *CogniCrypt* [Krü+19] in a Java API misuse dataset derived from *MuBench* [Ama+16], as well as Li's [Li20] five most common API misuse patterns for each of their categories. Specifically, we used *CogniCrypt* in version 1.0.0 and *CrySL* [Krü+19] in version 2.0.0. The artifact [Sch+22b] is available on figshare and includes all protocols for each API misuse, as well as a discussion of the proposed improvements to *CrySL*.

***FUM* Artifact**

`https://doi.org/10.6084/m9.figshare.16832749`

Risk Model and Empirical Study — Section 5.1

In Section 5.1, we presented a risk model (cf. Table 5.1) which we used to classify the cryptographic API misuses detected in the study on cryptographic API misuses in business applications [Wic+22], and for the *Severity Score* feature (cf. Section 7.4.7) in *SecAI* (*Contribution 5*, cf. Chapter 7). The artifact containing all information on the study and risk model is available on figshare.

Empirical Study Artifact

`https://doi.org/10.6084/m9.figshare.21178243`.

Adaptation of *CogniCrypt* — Section 5.2

In Section 5.2, we presented *CogniCrypt*$_{SUBS}$ [Wic+24], an adaptation of *CogniCrypt* [Krü+19] that supports the detection of *error chains*, which was merged into the *CogniCrypt* release version 3.1.0. The scripts and data collected during the empirical study (Section 5.2.5), the benchmark (cf. [Wic+24]), and the design and anonymized data from our user study (Section 5.2.5) are available on figshare.

CogniCrypt$_{SUBS}$

`https://github.com/CROSSINGTUD/CryptoAnalysis/releases/tag/3.1.0`

Empirical Study Artifact

`https://doi.org/10.6084/m9.figshare.24473197`.

Contribution 3, *CamBench* — Section 5.3

In Section 5.3, we presented *CamBench*, a benchmark for evaluating static analysis tools for cryptographic API misuse (JCA). The created micro-benchmark consists of 623 cases (cf. Table 5.4) with metadata files. We used *CamBench* for the training of *FP*$_{Predictor}$ [OSB26b] (cf. Section 7.4.8) and the evaluation of *LLM-SAST*$_{Repair}$ [Sch+26b] (*Contribution 4*, cf. Chapter 6). It is available on GitHub.

CamBench

`https://github.com/CROSSINGTUD/CamBench/releases/tag/CamBench`

Contribution 4, *LLM-SAST$_{Repair}$* — Chapter 6

In Chapter 6, we introduced *LLM-SAST$_{Repair}$* for automatically repairing misuses of the Java cryptographic API. The implementation of *LLM-SAST$_{Repair}$* and the results of the evaluation experiments are available as an artifact [Sch+26a] on figshare.

LLM-SAST$_{Repair}$

`https://doi.org/10.6084/m9.figshare.32594145`

Contribution 5, *SecAI* — Chapter 7

In Chapter 7, we present *SecAI* and its feature *FP$_{Predictor}$* [OSB26b] in Section 7.4.8. The repository for *SecAI* contains the implementation, documentation, and supplementary material for self-assessment of the fulfillment of usability criteria, following the approach of Nachtigall et al. [NSB22] (*Contribution 1*, cf. Chapter 3). The artifact [OSB26a] for *FP$_{Predictor}$* includes its implementation, the experimentation pipeline, and results from manual inspection. The *SecAI* repository is available on GitHub, and the *FP$_{Predictor}$* artifact is available on figshare.

SecAI

`https://github.com/secure-software-engineering/CogniCryptSQPlugin/releases/tag/Licensed-version`

FP$_{Predictor}$

`https://doi.org/10.6084/m9.figshare.30389425`

Bibliography

[Abr+] Ajin Abraham, Magaofei, Matan Dobrushin, and Vincent Nadal. *Mobile Security Framework (MobSF)*. `https://github.com/MobSF/Mobile-Security-Framework-MobSF`. Last accessed: 2026-06-06 (cit. on p. 107).

[ARY19] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. "CryptoAPI-Bench: A Comprehensive Benchmark on Java Cryptographic API Misuses". In: *2019 IEEE Cybersecurity Development (SecDev)*. IEEE. 2019, pp. 49–61 (cit. on pp. xiii, xiv, 8, 14, 105–107, 111, 126, 128, 135, 139, 148–150, 166, 168).

[Aku+23] Vishwanath Akuthota, Raghunandan Kasula, Sabiha Tasnim Sumona, et al. "Vulnerability Detection and Monitoring Using LLM". In: *2023 IEEE 9th International Women in Engineering (WIE) Conference on Electrical and Computer Engineering (WIECON-ECE)*. 2023, pp. 309–314 (cit. on pp. 1, 5).

[Ali] Alibaba. *Qwen*. `https://chat.qwen.ai/`. Last accessed: 2026-06-06 (cit. on p. 136).

[All] Booz Allen. *AppCritique*. Last accessed: 2026-06-06. At the time of access, the AppCritique webpage was no longer available. During the literature review, the project appeared to be deprecated. (cit. on p. 107).

[Alp23] Alphabet. *Gemini*. `https://blog.google/innovation-and-ai/technology/ai/google-gemini-ai/`. Last accessed: 2026-06-06. 2023 (cit. on p. 164).

[Ama18] Sven Amann. "A Systematic Approach to Benchmark and Improve Automated Static Detection of Java-API Misuses". Dr.-Ing. dissertation. Department of Computer Science at the Technical University of Darmstadt, Mar. 2018 (cit. on pp. 7, 13, 48, 53–56, 58–68).

[Ama+16] Sven Amann, Sarah Nadi, Hoan A. Nguyen, Tien N. Nguyen, and Mira Mezini. "MuBench: A Benchmark for API-Misuse Detectors". In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 464–467 (cit. on pp. 7, 8, 48, 55, 69, 107, 108, 111, 112, 114, 116, 174).

[Ama+18] Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N Nguyen, and Mira Mezini. "A systematic evaluation of static api-misuse detectors". In: *IEEE Transactions on Software Engineering (TSE)*. Vol. 45. 12. IEEE, 2018 (cit. on pp. 48, 49, 55, 71, 106, 108, 112).

[Ama+26] Luis Amaral, Michael Schlichtig, Wagner Emanuel, et al. “From Legacy Designs to Vulnerability Fixes: Understanding SAST Adoption in Non-Technological Companies”. In: *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2026 (cit. on pp. xv, 6, 9, 153, 154, 156, 171).

[Ant+25] Gábor Antal, Bence Bogenfürst, Rudolf Ferenc, and Péter Hegedus. “Identifying Helpful Context for LLM-based Vulnerability Repair: A Preliminary Study”. In: *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. 2025, pp. 696–700 (cit. on p. 164).

[Apaa] Apache Software Foundation. *Apache Commons Collection*. `https://commons.apache.org/proper/commons-collections/`. Last accessed: 2026-06-06 (cit. on p. 52).

[Apab] Apache Software Foundation. *Apache Maven*. `https://maven.apache.org`. Last accessed: 2026-06-06 (cit. on pp. 39, 48, 79, 83, 94, 101, 105, 157).

[Apac] Apache Software Foundation. *CollatingIterator documentation*. `https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/iterators/CollatingIterator.html`. Last accessed: 2026-06-06 (cit. on p. 64).

[Art20] Philippe Arteau. *Find Security Bugs 1.11.0*. `https://find-sec-bugs.github.io/`. Last accessed: 2026-06-06. Oct. 2020 (cit. on pp. 1, 2, 26, 107, 108, 126).

[Avi+04] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. “Basic concepts and taxonomy of dependable and secure computing”. en. In: *IEEE Transactions on Dependable and Secure Computing* 1.1 (Jan. 2004), pp. 11–33 (cit. on pp. 85, 100).

[AWS] Amazon Web Services (AWS). *How can I check the integrity of an object uploaded to Amazon S3?* `https://repost.aws/knowledge-center/data-integrity-s3`. Last accessed: 2026-06-06 (cit. on p. 112).

[AP08] Nathaniel Ayewah and William Pugh. “A report on a survey and study of static analysis users”. In: *Proceedings of the 2008 workshop on Defects in large software systems*. 2008, pp. 1–5 (cit. on p. 27).

[Aye+08] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. “Using static analysis to find bugs”. In: *IEEE software* 25.5 (2008), pp. 22–29 (cit. on pp. 34, 35).

[Aye+07] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. “Evaluating static analysis defect warnings on production software”. In: *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2007, pp. 1–8 (cit. on p. 17).

[Ban] Bandit. `https://github.com/PyCQA/bandit`. Last accessed: 2026-06-06 (cit. on p. 26).

[Bar16] Titus Barik. “How should static analysis tools explain anomalies to developers?” In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 1118–1120 (cit. on p. 25).

[Bar+18] Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. “How should compilers explain problems to developers?” In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 633–643 (cit. on p. 29).

[Bar+14] Titus Barik, Kevin Lubick, Samuel Christie, and Emerson Murphy-Hill. “How developers visualize compiler messages: A foundational approach to notification construction”. In: *2014 Second IEEE Working Conference on Software Visualization*. IEEE. 2014, pp. 87–96 (cit. on p. 25).

[Bar+16] Titus Barik, Yoonki Song, Brittany Johnson, and Emerson Murphy-Hill. “From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration”. In: *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE. 2016, pp. 211–221 (cit. on p. 32).

[BKA11] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. “An Empirical Study of Object Protocols in the Wild”. In: *ECOOP 2011 – Object-Oriented Programming*. Ed. by Mira Mezini. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 2–26 (cit. on pp. 13, 54, 56, 60, 62).

[Bes+10] Al Bessey, Ken Block, Ben Chelf, et al. “A few billion lines of code later: using static analysis to find bugs in the real world”. In: *Communications of the ACM* 53.2 (2010), pp. 66–75 (cit. on pp. 11, 12, 17, 18, 20, 23, 25).

[BBA09] Kevin Bierhoff, Nels E Beckman, and Jonathan Aldrich. “Practical API protocol checking with access permissions”. In: *European Conference on Object-Oriented Programming*. Springer. 2009, pp. 195–219 (cit. on pp. 13, 54, 56, 60, 62).

[Bla+06] S. Blackburn, R. Garner, C. Hoffmann, et al. “The DaCapo Benchmarks: Java Benchmarking Development and Analysis”. In: *21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA)*. ACM, 2006 (cit. on pp. 8, 13, 105, 109, 111, 115).

[Ble98] Daniel Bleichenbacher. “Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1”. In: *Annual international cryptology conference*. Springer. 1998, pp. 1–12 (cit. on p. 78).

[Bod09] Eric Bodden. “Verifying finite-state properties of large-scale programs”. Available in print through ProQuest. Ph.D. thesis. McGill University, June 2009 (cit. on pp. 54, 62).

[BHL07] Eric Bodden, Laurie Hendren, and Ondřej Lhoták. “A staged static program analysis to improve the performance of runtime monitoring”. In: *European Conference on Object-Oriented Programming*. Springer. 2007, pp. 525–549 (cit. on pp. 56, 58, 63).

[BLH08] Eric Bodden, Patrick Lam, and Laurie Hendren. “Finding programming errors earlier by evaluating runtime monitors ahead-of-time”. In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 2008, pp. 36–47 (cit. on pp. 35, 165).

[BN18] Eric Bodden and Lisa Nguyen Quang Do. “Explainable static analysis”. In: *Software Engineering und Software Management 2018* (2018) (cit. on p. 38).

[Bon+21] Rodrigo Bonifácio, Stefan Krüger, Krishna Narasimhan, Eric Bodden, and Mira Mezini. *Dealing with Variability in API Misuse Specification*. Ed. by Anders Møller and Manu Sridharan. 2021 (cit. on pp. 6, 50).

[Bra+17] Alexandre Braga, Ricardo Dahab, Nuno Antunes, Nuno Laranjeiro, and Marco Vieira. “Practical evaluation of static analysis tools for cryptography: Benchmarking method and case study”. In: *IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2017 (cit. on pp. 8, 14, 105, 107).

[BMM10] M. Bruch, M. Mezini, and M. Monperrus. “Mining subclassing directives to improve framework reuse”. In: *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. 2010, pp. 141–150 (cit. on pp. 52, 66).

[BSI] German Federal Office for Information Security (BSI). *BSI TR-02102-1: "Cryptographic Mechanisms: Recommendations and Key Lengths"*. `https://www.bsi.bund.de/SharedDocs/Downloads/EN/BSI/Publications/TechGuidelines/TG02102/BSI-TR-02102-1.html`. Last accessed: 2026-06-06 (cit. on p. 113).

[Bui+23] Quang-Cuong Bui, Ranindya Paramitha, Duc-Ly Vu, Fabio Massacci, and Riccardo Scandariato. “APR4Vul: an empirical study of automatic program repair techniques on real-world Java vulnerabilities”. In: *Empirical Softw. Engg.* 29.1 (Dec. 2023) (cit. on pp. 126, 149).

[BFS17] Elisa Burato, Pietro Ferrara, and Fausto Spoto. “Security analysis of the OWASP benchmark with Julia”. In: *First Italian Conference on Cybersecurity (ITASEC)*. 2017 (cit. on p. 107).

[Cer+18] Ervina Cergani, Sebastian Proksch, Sarah Nadi, and Mira Mezini. “Investigating Order Information in API-Usage Patterns: A Benchmark and Empirical Study.” In: *ICSOFT*. 2018, pp. 91–102 (cit. on pp. 13, 54, 56, 60, 62).

[Cha+16] Alexia Chatzikonstantinou, Christoforos Ntantogian, Georgios Karopoulos, and Christos Xenakis. “Evaluation of Cryptography Usage in Android Applications”. In: *Proceedings of the 9th EAI International Conference on Bio-Inspired Information and Communications Technologies (Formerly BIONETICS)*. BICT’15. New York City, United States: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016, pp. 83–90 (cit. on pp. 55, 74).

[Che] Checkstyle. *Checkstyle*. `https://checkstyle.sourceforge.io/`. Last accessed: 2026-06-06 (cit. on pp. 2, 26, 126).

[CW07] Brian Chess and Jacob West. *Secure programming with static analysis*. Pearson Education, 2007 (cit. on pp. 2, 126).

[CB16] Maria Christakis and Christian Bird. “What developers want and need from program analysis: an empirical study”. In: *Proceedings of the 31st IEEE/ACM international conference on automated software engineering*. 2016, pp. 332–343 (cit. on pp. 2, 7, 12, 20, 34, 36–38, 44, 48, 90, 101, 110, 111, 154, 155, 158, 161, 163, 165).

[Coga] CogniCrypt developers. *CogniCrypt Eclipse plugin*. `https://github.com/eclipse-cognicrypt/CogniCrypt`. Last accessed: 2026-06-06 (cit. on pp. 6, 155).

[Cogb] CogniCrypt developers. *Crypto-API-Rules*. en. `https://github.com/CROSSINGTUD/Crypto-API-Rules`. Last accessed: 2026-06-06 (cit. on p. 16).

[Fb-] Fb-contrib. `http://fb-contrib.sourceforge.net`. Last accessed: 2026-06-06 (cit. on p. 26).

[Cpp] Cpplint. `https://github.com/cpplint/cpplint`. Last accessed: 2026-06-06 (cit. on p. 26).

[CSc] CScout. `https://www.spinellis.gr/cscout/`. Last accessed: 2026-06-06 (cit. on p. 26).

[DW09] Vidroha Debroy and W. Eric Wong. “Insights on Fault Interference for Programs with Multiple Bugs”. en. In: *2009 20th International Symposium on Software Reliability Engineering*. Mysuru, Karnataka, India: IEEE, Nov. 2009, pp. 165–174 (cit. on p. 100).

[DH09] U. Dekel and J. D. Herbsleb. “Improving API documentation usability with knowledge pushing”. In: *2009 IEEE 31st International Conference on Software Engineering*. 5000 Forbes Avenue, Pittsburgh, PA 15213 USA, 2009, pp. 320–330 (cit. on pp. 13, 52, 56, 60, 62, 65).

[Dev] DevSkim. `https://github.com/microsoft/DevSkim`. Last accessed: 2026-06-06 (cit. on p. 26).

[Dli] Dlint. `https://github.com/dlint-py/dlint`. Last accessed: 2026-06-06 (cit. on pp. 1, 2, 26).

[DB18] Lisa Nguyen Quang Do and Eric Bodden. “Gamifying static analysis”. In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2018, pp. 714–718 (cit. on p. 40).

[DB20] Lisa Nguyen Quang Do and Eric Bodden. “Explaining static analysis with rule graphs”. In: *IEEE Transactions on Software Engineering* (2020) (cit. on p. 25).

[DEB16] Lisa Nguyen Quang Do, Michael Eichberg, and Eric Bodden. “Toward an Automated Benchmark Management System”. In: *5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP)*. ACM, 2016 (cit. on pp. 8, 109–113).

[DWA20] Lisa Nguyen Quang Do, James R Wright, and Karim Ali. “Why do software developers use static analysis tools? a user-centered study of developer needs and motivations”. In: *IEEE Transactions on Software Engineering* 48.3 (2020), pp. 835–847 (cit. on pp. 3, 12, 20, 23, 32, 35, 36, 38–40, 42, 44).

[Dol+26] Greta Dolcetti, Vincenzo Arceri, Eleonora Iotti, et al. “Helping llms improve code generation using feedback from testing and static analysis”. In: *Discover Artificial Intelligence* 6.1 (2026), p. 314 (cit. on p. 149).

[Dye+13] Robert Dyer, Hoan Anh Nguyen, Hridesh Rajan, and Tien N Nguyen. “Boa: A language and infrastructure for analyzing ultra-large-scale software repositories”. In: *ACM 35th International Conference on Software Engineering (ICSE)*. ACM, 2013 (cit. on p. 108).

[Ecla] Eclipse Foundation. *Eclipse IDE*. `https://www.eclipse.org/downloads/`. Last accessed: 2026-06-06 (cit. on pp. 39, 52, 98, 99).

[Eclb] Eclipse Foundation. *Eclipse JFace documentation*. `https://wiki.eclipse.org/JFace`. Last accessed: 2026-06-06 (cit. on p. 52).

[Ege+13] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. “An empirical study of cryptographic misuse in Android applications”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. Nov. 2013, pp. 73–84 (cit. on pp. 1, 3, 8, 13, 48, 54, 55, 76, 77, 83, 101, 104, 108, 116, 119).

[Err] Error Prone. `https://errorprone.info`. Last accessed: 2026-06-06 (cit. on p. 26).

[ESL] ESLint. `https://eslint.org`. Last accessed: 2026-06-06 (cit. on p. 26).

[Fah+12] Sascha Fahl, Marian Harbach, Thomas Muders, et al. “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*. 2012, pp. 50–61 (cit. on p. 77).

[Fan+24] Chongzhou Fang, Ning Miao, Shaurya Srivastav, et al. “Large language models for code analysis: Do {LLMs} really do their job?” In: *33rd USENIX Security Symposium (USENIX Security 24)*. 2024, pp. 829–846 (cit. on p. 126).

[fLP13] fasmat, Adam Langley, and Rob Pike. *Issue 7860047: code review 7860047: crypto/cipher: Added BlockMode for ECB Encryption and D... - Code Review*. `https://codereview.appspot.com/7860047/#ps1`. Last accessed: 2026-06-06. June 2013 (cit. on p. 101).

[FMS18] Johannes Feichtner, David Missmann, and Raphael Spreitzer. “Automated binary analysis on ios: A case study on cryptographic misuse in ios applications”. In: *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*. 2018, pp. 236–247 (cit. on p. 101).

[Fer] Niels Ferguson. *Authentication weaknesses in GCM*. `https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/comments/cwc-gcm/ferguson2.pdf`. Last accessed: 2026-06-06. NIST (cit. on p. 78).

[FG25] Ehsan Firouzi and Mohammad Ghafari. “Can generative AI detect and fix real-world cryptographic misuses?” In: *Journal of Systems and Software* (2025), p. 112650 (cit. on pp. 8, 126, 127, 135, 136, 139).

[FIR23] FIRST.Org, Inc. *Common Vulnerability Scoring System (CVSS) v4.0 Specification Document*. https://www.first.org/cvss/specification-document. Last accessed: 2026-06-06. Nov. 2023 (cit. on p. 76).

[Fla] Flawfinder. https://dwheeler.com/flawfinder/. Last accessed: 2026-06-06 (cit. on p. 26).

[Fu+22] Michael Fu, Chakkrit Tantithamthavorn, Trung Le, Van Nguyen, and Dinh Phung. "VulRepair: a T5-based automated software vulnerability repair". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 935–947 (cit. on pp. 126, 150).

[Gaj+25] Jugal Gajjar, Kamalasankari Subramaniakuppusamy, Relsy Puthal, and Kaustik Ranaware. "SecureFixAgent: A Hybrid LLM Agent for Automated Python Static Vulnerability Repair". In: *International Conference on Machine Learning and Applications, ICMLA 2025, Boca Raton, FL, USA, December 3–5, 2025*. IEEE, 2025, pp. 1390–1395. arXiv: 2509.16275 [cs.CR] (cit. on p. 150).

[Gaj+17] Jyoti Gajrani, Meenakshi Tripathi, Vijay Laxmi, et al. "sPECTRA: a Precise framEwork for analyzing CrypTographic vulneRabilities in Android apps". In: *2017 14th IEEE Annual Consumer Communications & Networking Conference (CCNC)*. IEEE. 2017, pp. 854–860 (cit. on p. 1).

[Gao+19] Jun Gao, Pingfan Kong, Li Li, Tegawendé F Bissyandé, and Jacques Klein. "Negative Results on Mining Crypto-API Usage Rules in Android Apps". In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 388–398 (cit. on pp. 6, 50, 75, 79, 80, 86).

[Gita] Git. *Git*. https://git-scm.com/. Last accessed: 2026-06-06 (cit. on p. 55).

[Gitb] GitHub. *CodeQL*. https://github.com/github/codeql/. Last accessed: 2026-06-06 (cit. on pp. 135, 148).

[Gitc] GitHub. *GitHub*. https://github.com. Last accessed: 2026-06-06 (cit. on pp. 48, 56, 75, 79, 93, 113, 163, 164).

[Gon+25] Jianian Gong, Nachuan Duan, Ziheng Tao, et al. "How Well Do Large Language Models Serve as End-to-End Secure Code Agents for Python?" In: *Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering*. 2025, pp. 1004–1013 (cit. on p. 149).

[Goo+16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. Vol. 1. 2. MIT press Cambridge, 2016 (cit. on p. 166).

[Goo] Google. *Android API reference*. https://developer.android.com/reference. Last accessed: 2026-06-06 (cit. on p. 55).

[Gor+18] Peter Leo Gorski, Luigi Lo Iacono, Dominik Wermke, et al. "Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse". In: *14th Symposium on Usable Privacy and Security (SOUPS)*. USENIX Association, Aug. 2018 (cit. on p. 104).

[Gra] Gradle Inc. *Gradle*. `https://gradle.org`. Last accessed: 2026-06-06 (cit. on pp. 79, 94, 157).

[Gu+19] Z. Gu, J. Wu, J. Liu, M. Zhou, and M. Gu. "An Empirical Study on API-Misuse Bugs in Open-Source C Programs". In: *2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC)*. Vol. 1. Milwaukee, WI, USA: IEEE, 2019, pp. 11–20 (cit. on pp. 7, 48, 55).

[GE09] Philip J. Guo and Dawson Engler. "Linux Kernel Developer Responses to Static Analysis Bug Reports". In: *USENIX annual technical conference, 2009*. 2009 (cit. on pp. 90, 101).

[HG09] M. Hamill and K. Goseva-Popstojanova. "Common Trends in Software Fault and Failure Data". en. In: *IEEE Transactions on Software Engineering* 35.4 (July 2009), pp. 484–496 (cit. on p. 100).

[Har+10] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. "What would other programmers do: suggesting solutions to error messages". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2010, pp. 1019–1028 (cit. on pp. 20, 33).

[HGN20] Mohammadreza Hazhirpasand, Mohammad Ghafari, and Oscar Nierstrasz. "Java Cryptography Uses in the Wild". In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM '20. Bari, Italy: Association for Computing Machinery, 2020 (cit. on pp. 1, 6, 17, 50, 83, 86, 116, 119).

[HW09] Sarah Heckman and Laurie Williams. "A model building process for identifying actionable static analysis alerts". In: *2009 International Conference on Software Testing Verification and Validation*. IEEE. 2009, pp. 161–170 (cit. on p. 31).

[HW11] Sarah Heckman and Laurie Williams. "A systematic literature review of actionable alert identification techniques for automated static code analysis". In: *Information and Software Technology* 53.4 (2011), pp. 363–387 (cit. on p. 31).

[Heg] Hegel. `https://hegel.js.org`. Last accessed: 2026-06-06 (cit. on p. 26).

[Hei+11] Lars Heinemann, Florian Deissenboeck, Mario Gleirscher, Benjamin Hummel, and Maximilian Irlbeck. "On the extent and nature of software reuse in open source java projects". In: *International Conference on Software Reuse*. Springer. 2011, pp. 207–222 (cit. on pp. 12, 47).

[Hel+20] Dominik Helm, Florian Kübler, Michael Reif, Michael Eichberg, and Mira Mezini. "Modular collaborative program analysis in OPAL". en. In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Virtual Event USA: ACM, Nov. 2020, pp. 184–196 (cit. on pp. 94, 113, 156).

[HJZ13] K. Herzig, S. Just, and A. Zeller. "It's not a bug, it's a feature: How misclassification impacts bug prediction". In: *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA, 2013, pp. 392–401 (cit. on pp. 48, 55).

[Hod22] Rashina Hoda. “Socio-Technical Grounded Theory for Software Engineering”. In: *IEEE Trans. Software Eng.* 48.10 (2022), pp. 3808–3832 (cit. on p. xv).

[HH06] D. Hou and H.J. Hoover. “Using SCL to specify and check design intent in source code”. In: *IEEE Transactions on Software Engineering* 32.6 (2006), pp. 404–423 (cit. on p. 48).

[HP04] David Hovemeyer and William Pugh. “Finding Bugs is Easy”. In: *SIGPLAN Not.* 39.12 (Dec. 2004), pp. 92–106 (cit. on pp. 54, 56).

[HL06] Michael Howard and Steve Lipner. *The security development lifecycle: SDL, a process for developing demonstrably more secure software*. Secure software development series. Redmond, Washington: Microsoft Press, 2006 (cit. on p. 2).

[Hug] Hugging Face. *Hugging Face*. `https://huggingface.co/`. Last accessed: 2026-06-06 (cit. on p. 136).

[Imt+19] Nasif Imtiaz, Akond Rahman, Effat Farhana, and Laurie Williams. “Challenges with responding to static analysis tool alerts”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. 2019, pp. 245–249 (cit. on pp. 20, 34, 37).

[Inf] InferSharp. `https://github.com/microsoft/infersharp`. Last accessed: 2026-06-06 (cit. on p. 26).

[Isl20] Md Johirul Islam. “Towards understanding the challenges faced by machine learning software developers and enabling automated solutions”. Ph.D. thesis. Iowa State University, 2020 (cit. on pp. 7, 48, 54, 55).

[JAM25] Niveen O. Jaffal, Mohammed Alkhanafseh, and David Mohaisen. “Large Language Models in Cybersecurity: A Survey of Applications, Vulnerabilities, and Defense Techniques”. en. In: *AI* 6.9 (Sept. 2025). Publisher: Multidisciplinary Digital Publishing Institute, p. 216 (cit. on pp. 126, 127, 150).

[Jas+04] Monique WM Jaspers, Thiemo Steen, Cor Van Den Bos, and Maud Geenen. “The think aloud method: a guide to user interface design”. In: *International journal of medical informatics* 73.11-12 (2004), pp. 781–795 (cit. on p. 98).

[Jet] JetBrains. *IntelliJ IDEA*. `https://www.jetbrains.com/idea/download/`. Last accessed: 2026-06-06 (cit. on p. 39).

[Ji+23] Ziwei Ji, Tiezheng Yu, Yan Xu, et al. “Towards Mitigating LLM Hallucination via Self Reflection”. In: *Findings of the Association for Computational Linguistics: EMNLP 2023*. Ed. by Houda Bouamor, Juan Pino, and Kalika Bali. Singapore: Association for Computational Linguistics, Dec. 2023, pp. 1827–1843 (cit. on p. 126).

[JCX19] Z. Jin, K. Y. Chee, and X. Xia. “What Do Developers Discuss about Biometric APIs?” In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 348–352 (cit. on p. 54).

[Joh15] Brittany Johnson. “Adapting program analysis tool notifications to the individual developer”. In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2015, pp. 291–292 (cit. on pp. 28, 36).

[Joh+15] Brittany Johnson, Rahul Pandita, Emerson Murphy-Hill, and Sarah Heckman. "Bespoke tools: adapted to the concepts developers know". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 878–881 (cit. on pp. 28, 33, 35, 36).

[Joh+16] Brittany Johnson, Rahul Pandita, Justin Smith, et al. "A cross-tool communication study on program analysis tool notifications". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 73–84 (cit. on pp. 28, 42).

[Joh+13] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why don't software developers use static analysis tools to find bugs?" In: *2013 35th International Conference on Software Engineering (ICSE)*. IEEE. 2013, pp. 672–681 (cit. on pp. 2, 7, 11, 17, 20, 23, 25–29, 32–34, 36–40, 44, 48, 81, 90, 101, 111, 154).

[JJE14] René Just, Darioush Jalali, and Michael D. Ernst. "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: Association for Computing Machinery, 2014, pp. 437–440 (cit. on pp. 48, 55).

[Kar+24] Kadiray Karakaya, Stefan Schott, Jonas Klauke, et al. "SootUp: A redesign of the soot static analysis framework". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2024, pp. 229–247 (cit. on pp. 156, 159, 166).

[Kas+13] Alok Kumar Kasgar, Mukesh Kumar Dhariwal, Neeraj Tantubay, and Hina Malviya. "A review paper of message digest 5 (MD5)". In: *International Journal of Modern Engineering & Management Research*. Vol. 1. 4. 2013 (cit. on p. 112).

[Kav+24] Arya Kavian, Mohammad Mehdi Pourhashem Kallehbasti, Sajjad Kazemi, Ehsan Firouzi, and Mohammad Ghafari. "LLM security guard for code". In: *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*. 2024, pp. 600–603 (cit. on pp. 127, 163).

[Kec+19] Maria Kechagia, Xavier Devroey, Annibale Panichella, Georgios Gousios, and Arie van Deursen. "Effective and Efficient API Misuse Detection via Exception Propagation and Search-Based Testing". In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, pp. 192–203 (cit. on pp. 54, 56).

[KS14] Maria Kechagia and Diomidis Spinellis. "Undocumented and Unchecked: Exceptions That Spell Trouble". In: *Proceedings of the 11th Working Conference on Mining Software Repositories*. MSR 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 312–315 (cit. on pp. 54, 56).

[Kha+20] Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. "An Empirical Study on the Use and Misuse of Java 8 Streams." In: *23rd International Conference, FASE 2020*. Dublin, Ireland: ETAPS 2020, Apr. 2020, pp. 97–118 (cit. on pp. 54–56, 60, 62, 64, 65).

[Khe+26a] Mugdha Khedkar, Michael Schlichtig, Nihad Atakishiyev, and Eric Bodden. "Between law and code: challenges and opportunities for automating privacy assessments". In: *Automated Software Engineering* 33.2 (2026), p. 56 (cit. on p. xv).

[KSB24] Mugdha Khedkar, Michael Schlichtig, and Eric Bodden. "Advancing android privacy assessments with automation". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering Workshops*. ACM, 2024, pp. 218–222 (cit. on p. xv).

[KSB26] Mugdha Khedkar, Michael Schlichtig, and Eric Bodden. "Source Code-Driven GDPR Documentation: Supporting RoPA with Assessor View." In: *2026 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. IEEE, 2026 (cit. on p. xvi).

[Khe+25] Mugdha Khedkar, Michael Schlichtig, Santhosh Mohan, and Eric Bodden. *Visualizing Privacy-Relevant Data Flows in Android Applications*. 2025. arXiv: 2503.16640 [cs.CR] (cit. on p. xvi).

[Khe+26b] Mugdha Khedkar, Michael Schlichtig, Mohamed Soliman, and Eric Bodden. "Challenges in Android Data Disclosure: An Empirical Study". In: *2026 IEEE/ACM 13th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 2026 (cit. on p. xvi).

[Kho+08] Yit Phang Khoo, Jeffrey S Foster, Michael Hicks, and Vibha Sazawal. "Path projection for user-centered static analysis tools". In: *Proceedings of the 8th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 2008, pp. 57–63 (cit. on p. 27).

[Kit04] Barbara Kitchenham. "Procedures for performing systematic reviews". In: *Keele, UK, Keele University* 33 (2004), pp. 1–26 (cit. on p. 51).

[Kit+09] Barbara Kitchenham, O Pearl Brereton, David Budgen, et al. "Systematic literature reviews in software engineering–a systematic literature review". In: *Information and software technology* 51.1 (2009), pp. 7–15 (cit. on p. 23).

[Kit96] Barbara Ann Kitchenham. "Evaluating software engineering methods and tool part 1: The evaluation context and evaluation methods". In: *ACM SIGSOFT Software Engineering Notes* 21.1 (1996), pp. 11–14 (cit. on p. 23).

[KR03] Vlastimil Klima and Tomas Rosa. "Side channel attacks on CBC encrypted messages in the PKCS# 7 format". In: *Cryptology ePrint Archive* (2003) (cit. on p. 78).

[KM04] Andrew J Ko and Brad A Myers. "Designing the whyline: a debugging interface for asking questions about program behavior". In: *Proceedings of the SIGCHI conference on Human factors in computing systems*. 2004, pp. 151–158 (cit. on p. 41).

[Kra99] Douglas Kramer. "API Documentation from Source Code Comments: A Case Study of Javadoc". In: *Proceedings of the 17th Annual International Conference on Computer Documentation*. SIGDOC '99. New Orleans, Louisiana, USA: Association for Computing Machinery, 1999, pp. 147–153 (cit. on p. 58).

[KAB20] Stefan Krüger, Karim Ali, and Eric Bodden. “CogniCryptGEN: generating code for the secure usage of crypto APIs”. In: *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*. CGO ’20. San Diego, CA, USA: Association for Computing Machinery, 2020, pp. 185–198 (cit. on p. 5).

[Krü+23] Stefan Krüger, Michael Reif, Anna-Katharina Wickert, et al. “ Securing Your Crypto-API Usage Through Tool Support - A Usability Study ”. In: *2023 IEEE Secure Development Conference (SecDev)*. Los Alamitos, CA, USA: IEEE Computer Society, Oct. 2023, pp. 14–25 (cit. on p. 6).

[Krü+19] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. “CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs”. In: *IEEE Transactions on Software Engineering* 47.11 (2019), pp. 2382–2400 (cit. on pp. x, xi, xiii–xv, 2, 4–6, 8, 9, 11, 14–17, 26, 48, 69, 71, 74–77, 79, 80, 83, 86–88, 94, 101, 104, 105, 107, 108, 126, 128, 131, 134, 135, 139, 148, 150, 153–159, 161–166, 168, 169, 171, 174, 175).

[Krü20] Krüger, Stefan. “CogniCrypt - The Secure Integration of Cryptographic Software”. Dr. rer. nat. dissertation. University Paderborn, Oct. 2020 (cit. on pp. 3, 5, 6, 54, 55, 62, 63, 65).

[Lan92] William Landi. “Undecidability of static analysis”. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 1.4 (1992), pp. 323–337 (cit. on pp. 2, 126).

[Laz+14] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. “Why Does Cryptographic Software Fail? A Case Study and Open Problems”. In: *Proceedings of 5th Asia-Pacific Workshop on Systems*. APSys ’14. Beijing, China: Association for Computing Machinery, 2014 (cit. on pp. 1, 3, 7, 48, 54, 55, 76, 104).

[LPR19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. “Automated program repair”. In: *Commun. ACM* 62.12 (Nov. 2019), pp. 56–65 (cit. on pp. 8, 125, 127–129, 149).

[Li+18] H. Li, S. Li, J. Sun, et al. “Improving API Caveats Accessibility by Mining API Caveats Knowledge Graph”. In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2018, pp. 183–193 (cit. on pp. 13, 53, 56).

[Li+26] Junjie Li, Fazle Rabbi, Cheng Cheng, et al. “An exploratory study on fine-tuning large language models for secure code generation”. In: *Empirical Software Engineering* 31.4 (2026), p. 81 (cit. on p. 149).

[Li+17] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, et al. “Static analysis of android apps: A systematic literature review”. In: *Information and Software Technology*. Vol. 88. 2017 (cit. on pp. 114, 116).

[Li+22] Wenqing Li, Shijie Jia, Limin Liu, et al. “CryptoGo: Automatic Detection of Go Cryptographic API Misuses”. In: *Proceedings of the 38th Annual Computer Security Applications Conference*. ACSAC ’22. New York, NY, USA: Association for Computing Machinery, Dec. 2022, pp. 318–331 (cit. on p. 101).

[Li20] Xia Li. “An Integrated Approach for Automated Software Debugging via Machine Learning and Big Code Mining”. Ph.D. thesis. The University of Texas at Dellas, 2020 (cit. on pp. 7, 48, 49, 54, 56, 60, 61, 64, 66, 68, 69, 174).

[LDN25] Ziyang Li, Saikat Dutta, and Mayur Naik. “IRIS: LLM-assisted static analysis for detecting security vulnerabilities”. In: *International Conference on Learning Representations*. Vol. 2025. 2025, pp. 35735–35758 (cit. on pp. 126, 127).

[LBP22] Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. “An empirical study on the effectiveness of static C code analyzers for vulnerability detection”. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, July 2022, pp. 544–555 (cit. on pp. 85, 93, 100).

[Liu+24] Zhijie Liu, Yutian Tang, Xiapu Luo, Yuming Zhou, and Liang Feng Zhang. “No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT”. In: *IEEE Transactions on Software Engineering* 50.6 (2024), pp. 1548–1584 (cit. on p. 126).

[Liv+15] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, et al. “In defense of soundiness: A manifesto”. In: *Communications of the ACM* 58.2 (2015) (cit. on p. 112).

[Luc+12] Lucia, Ferdian Thung, David Lo, and Lingxiao Jiang. “Are faults localizable?” en. In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. Zurich: IEEE, June 2012, pp. 74–77 (cit. on p. 100).

[Luo+18] Tianyue Luo, Jingzheng Wu, Mutian Yang, et al. “MAD-API: Detection, Correction and Explanation of API Misuses in Distributed Android Applications”. In: *Artificial Intelligence and Mobile Services – AIMS 2018*. Ed. by Marco Aiello, Yujiu Yang, Yuexian Zou, and Liang-Jie Zhang. Cham: Springer International Publishing, 2018, pp. 123–140 (cit. on pp. 7, 48, 55, 56, 60, 63).

[Lv+20] Tao Lv, Ruishi Li, Yi Yang, et al. “RTFM! Automatic Assumption Discovery and Verification Derivation from Library Document for API Misuse Detection”. In: *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’20. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 1837–1852 (cit. on pp. 52, 53, 56, 61).

[MR13] W. Maalej and M. P. Robillard. “Patterns of Knowledge in API Reference Documentation”. In: *IEEE Transactions on Software Engineering* 39.9 (Sept. 2013), pp. 1264–1282 (cit. on pp. 13, 52, 56, 58).

[Mar] Daniel Marjamaki. *Cppcheck: A Tool for Static C/C++ Code Analysis*. `https://sourceforge.net/projects/cppcheck/`. Last accessed: 2026-06-06 (cit. on p. 26).

[McC] McCabe. `https://github.com/PyCQA/mccabe`. Last accessed: 2026-06-06 (cit. on p. 26).

[McC76] Thomas J McCabe. “A complexity measure”. In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320 (cit. on p. 117).

[Men+13] Vinícius Rafael Lobo de Mendonça, Cássio Leonardo Rodrigues, Fabrízzio Alphonsus A de MN Soares, and Auri Marcelo Rizzo Vincenzi. “Static analysis techniques and tools: A systematic mapping study”. In: *ICSEA* (2013) (cit. on pp. 2, 126).

[Met24] Meta. *Llama*. `https://ai.meta.com/blog/meta-llama-3/`. Last accessed: 2026-06-06. 2024 (cit. on pp. 135, 136).

[Mik+13] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. “Efficient Estimation of Word Representations in Vector Space”. In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2–4, 2013, Workshop Track Proceedings*. Ed. by Yoshua Bengio and Yann LeCun. 2013. arXiv: `1301.3781 [cs.CL]` (cit. on p. 166).

[MKP23] Amir M Mir, Mehdi Keshani, and Sebastian Proksch. “On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem”. In: *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2023, pp. 201–211 (cit. on p. 48).

[MR17] Joydeep Mitra and Venkatesh-Prasad Ranganath. “Ghera: A repository of android app vulnerability benchmarks”. In: *13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*. ACM, 2017 (cit. on pp. 8, 14, 105, 107, 108).

[Mita] The MITRE Corporation. *Common Weakness Enumeration (CWE)*. `https://cwe.mitre.org/`. Last accessed: 2026-06-06 (cit. on pp. 29, 161).

[Mitb] The MITRE Corporation. *Common Weakness Scoring System (CWSS): Scoring the Severity of Software Weaknesses*. `https://cwe.mitre.org/cwss/`. Last accessed: 2026-06-06. 2014 (cit. on p. 76).

[Mon+12] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. “What should developers be aware of? An empirical study on the directives of API documentation”. In: *Empirical Software Engineering* 17.6 (Dec. 2012), pp. 703–737 (cit. on pp. 7, 13, 49–53, 55–58, 60–66).

[Mou+25] Yutao Mou, Xiao Deng, Yuxiao Luo, Shikun Zhang, and Wei Ye. “Can You Really Trust Code Copilot? Evaluating Large Language Models from a Code Security Perspective”. In: *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2025, pp. 17349–17369 (cit. on p. 149).

[MuB] MuBench Dataset. *Software Technology Group, TU Darmstadt*. `https://github.com/stg-tud/MUBench/`. Last accessed: 2026-06-06 (cit. on p. 55).

[MCJ17] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. “Bayesian specification learning for finding API usage errors”. In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 2017, pp. 151–162 (cit. on p. 48).

[MS16] Tukaram Muske and Alexander Serebrenik. “Survey of approaches for handling static analysis alarms”. In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE. 2016, pp. 157–166 (cit. on pp. 34, 35, 37, 41).

[MBB18] Ildar Muslukhov, Yazan Boshmaf, and Konstantin Beznosov. “Source attribution of cryptographic API misuse in android applications”. In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 2018, pp. 133–146 (cit. on pp. 48, 83).

[MWI17] Ahmad Mustafa, Wan MN Wan-Kadir, and I Ibrahim. “Comparative evaluation of the state-of-art requirements-based test case generation approaches”. In: *International Journal on Advanced Science, Engineering and Information Technology*. Vol. 7. 4-2. 2017 (cit. on p. 114).

[Myp] Mypy. `http://mypy-lang.org`. Last accessed: 2026-06-06 (cit. on p. 26).

[NDB19] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. “Explaining Static Analysis-A Perspective”. In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*. IEEE. 2019, pp. 29–32 (cit. on pp. 3, 7, 12, 23, 33, 45).

[NSB22] Marcus Nachtigall, Michael Schlichtig, and Eric Bodden. “A large-scale study of usability criteria addressed by static analysis tools”. In: *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2022. Virtual, South Korea: Association for Computing Machinery, 2022, pp. 532–543 (cit. on pp. ix, 2, 3, 5–7, 9, 154, 155, 158, 159, 167, 169, 171, 176).

[Nad+16] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. “Jumping through Hoops: Why Do Java Developers Struggle with Cryptography APIs?” In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE ’16. Austin, Texas: Association for Computing Machinery, 2016, pp. 935–946 (cit. on pp. 1, 2, 7, 12, 48, 54, 55, 58, 59, 104, 125, 164).

[Ngu+17] Duc Cuong Nguyen, Dominik Wermke, Yasemin Acar, et al. “A stitch in time: Supporting android developers in writingsecure code”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 1065–1077 (cit. on pp. 41, 48, 107).

[Ngu+15] Tam The Nguyen, Hung Viet Pham, Phong Minh Vu, and Tung Thanh Nguyen. “Recommending API usages for mobile apps with Hidden Markov Model”. In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 795–800 (cit. on p. 48).

[NVN19] Tam The Nguyen, Phong Minh Vu, and Tung Thanh Nguyen. *API Misuse Correction: A Statistical Approach*. 2019. arXiv: `1908.06492 [cs.SE]` (cit. on pp. 13, 53, 56, 58, 60–62, 64).

[NISa] National Institute of Standards and Technology (NIST). *SP 800-175A: Guideline for Using Cryptographic Standards in the Federal Government: Directives, Mandates and Policies*. https://csrc.nist.gov/publications/detail/sp/800-175a/final. Last accessed: 2026-06-06 (cit. on p. 113).

[NISb] National Institute of Standards and Technology (NIST). *SP 800-175B Rev. 1 Guideline for Using Cryptographic Standards in the Federal Government: Cryptographic Mechanisms*. https://csrc.nist.gov/publications/detail/sp/800-175b/rev-1/final. Last accessed: 2026-06-06 (cit. on p. 113).

[Nod] NodeJSScan. https://github.com/ajinabraham/nodejsscan. Last accessed: 2026-06-06 (cit. on p. 26).

[NSA22] NSA Media Relations. *NSA Releases Guidance on How to Protect Against Software Memory Safety Issues*. en-US. https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/. press release. Last accessed: 2026-06-06. Nov. 2022 (cit. on p. 2).

[Nun+24] Ana Nunez, Nafis Tanveer Islam, Sumit Kumar Jha, and Peyman Najafirad. *AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing*. 2024. arXiv: 2409.10737 [cs.SE] (cit. on pp. 126, 127).

[OAS20] OASIS. *Static Analysis Results Interchange Format (SARIF) Version 2.1.0*. https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html. Last accessed: 2026-06-06. Feb. 2020 (cit. on pp. 8, 134, 135, 148, 151, 157).

[OSB26a] Tom Ohlmer, Michael Schlichtig, and Eric Bodden. *Artifact for $FP_{Predictor}$ – False Positive Prediction for Static Analysis Reports*. https://doi.org/10.6084/m9.figshare.30389425. June 2026 (cit. on pp. 166, 176).

[OSB26b] Tom Ohlmer, Michael Schlichtig, and Eric Bodden. "FP-Predictor – False Positive Prediction for Static Analysis Reports". In: *2026 IEEE/ACM 2nd International Workshop on Advancing Static Analysis for Researchers and Industry Practitioners in Software Engineering (STATIC)*. 2026 (cit. on pp. xiv, 3, 5, 8, 9, 165, 169, 171, 175, 176).

[Oll] Ollama. *Ollama*. https://ollama.ai. Last accessed: 2026-06-06 (cit. on p. 164).

[Ope] OpenAI. *GPT-4o*. https://openai.com/index/hello-gpt-4o/. Last accessed: 2026-06-06 (cit. on pp. 126, 131, 136, 164).

[Oraa] Oracle. *AlgorithmParameters documentation*. https://docs.oracle.com/javase/8/docs/api/java/security/AlgorithmParameters.html. Last accessed: 2026-06-06 (cit. on p. 61).

[Orab] Oracle. *Date documentation*. https://docs.oracle.com/javase/6/docs/api/java/util/Date.html. Last accessed: 2026-06-06 (cit. on p. 64).

[Orac] Oracle. *FileReader documentation*. https://docs.oracle.com/javase/8/docs/api/java/io/FileReader.html. Last accessed: 2026-06-06 (cit. on pp. 51, 54, 59).

[Orad] Oracle. *HashMap documentation*. https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html. Last accessed: 2026-06-06 (cit. on p. 65).

[Orae] Oracle. *InputStreamReader documentation*. https://docs.oracle.com/javase/8/docs/api/java/io/InputStreamReader.html. Last accessed: 2026-06-06 (cit. on p. 61).

[Oraf] Oracle. *Iterator documentation*. https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html. Last accessed: 2026-06-06 (cit. on pp. 58, 62).

[Orag] Oracle. *Java Class Library (JCL) documentation*. https://docs.oracle.com/javase/8/docs/api/allclasses-frame.html. Last accessed: 2026-06-06 (cit. on pp. 48, 52).

[Orah] Oracle. *Java Cryptography Architecture (JCA) Reference Guide*. en. https://docs.oracle.com/javase/8/docs/technotes/guides/security/crypto/CryptoSpec.html. Last accessed: 2026-06-06 (cit. on p. 14).

[Orai] Oracle. *javax.swing documentation*. https://docs.oracle.com/javase/8/docs/api/javax/swing/package-summary.html. Last accessed: 2026-06-06 (cit. on pp. 65, 67).

[Oraj] Oracle. *KeyStore documentation*. https://docs.oracle.com/javase/8/docs/api/java/security/KeyStore.html. Last accessed: 2026-06-06 (cit. on p. 64).

[Orak] Oracle. *Object documentation*. https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html. Last accessed: 2026-06-06 (cit. on p. 65).

[Oral] Oracle. *String documentation*. https://docs.oracle.com/javase/8/docs/api/java/lang/String.html. Last accessed: 2026-06-06 (cit. on pp. 57, 63).

[Oram] Oracle. *Timestamp documentation*. https://docs.oracle.com/javase/6/docs/api/java/sql/Timestamp.html. Last accessed: 2026-06-06 (cit. on pp. 56, 64).

[Oran] Oracle. *URLDecoder documentation*. https://docs.oracle.com/javase/8/docs/api/java/net/URLDecoder.html. Last accessed: 2026-06-06 (cit. on p. 63).

[Owa] OWASP Dependency Check. https://owasp.org/www-project-dependency-check/. Last accessed: 2026-06-06 (cit. on p. 26).

[Oye+18] Tosin Daniel Oyetoyan, Bisera Milosheska, Mari Grini, and Daniela Soares Cruzes. "Myths and facts about static application security testing tools: an action research at Telenor digital". In: *International Conference on Agile Software Development*. Springer, Cham. 2018, pp. 86–103 (cit. on p. 38).

[Pic+21] Luca Piccolboni, Giuseppe Di Guglielmo, Luca P Carloni, and Simha Sethumadhavan. "Crylogger: Detecting crypto misuses dynamically". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1972–1989 (cit. on pp. 83, 101).

[PMD] PMD. *PMD*. https://pmd.github.io/. Last accessed: 2026-06-06 (cit. on pp. 1, 2, 26, 107, 126).

[Pra+12] Michael Pradel, Ciera Jaspan, Jonathan Aldrich, and Thomas R Gross. "Statically checking API protocol conformance with mined multi-object specifications". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE. 2012, pp. 925–935 (cit. on p. 48).

[Pum] Puma Scan. https://www.pumascan.com/. Last accessed: 2026-06-06 (cit. on p. 26).

[Pyc] Pycodestyle. https://pycodestyle.pycqa.org/en/latest/. Last accessed: 2026-06-06 (cit. on p. 26).

[PyD] PyDev. https://www.pydev.org. Last accessed: 2026-06-06 (cit. on p. 26).

[Pyd] Pydocstyle. https://github.com/PyCQA/pydocstyle. Last accessed: 2026-06-06 (cit. on p. 26).

[Pyf] Pyflakes. https://pypi.org/project/pyflakes/. Last accessed: 2026-06-06 (cit. on p. 26).

[Pyl] Pylint. https://pylint.org. Last accessed: 2026-06-06 (cit. on p. 26).

[Pyra] Pyre-Check. https://pyre-check.org. Last accessed: 2026-06-06 (cit. on p. 26).

[Pyrb] Pyright. https://github.com/microsoft/pyright. Last accessed: 2026-06-06 (cit. on p. 26).

[Pyt] Pytype. https://github.com/google/pytype. Last accessed: 2026-06-06 (cit. on p. 26).

[QWR18] Lina Qiu, Yingying Wang, and Julia Rubin. "Analyzing the analyzers: Flowdroid/iccta, amandroid, and droidsafe". In: *27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2018 (cit. on pp. 114, 116).

[Rad] Radon. https://pypi.org/project/radon/. Last accessed: 2026-06-06 (cit. on p. 26).

[Rah+19] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, et al. "CryptoGuard: High Precision Detection of Cryptographic Vulnerabilities in Massive-sized Java Projects". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. Nov. 2019, pp. 2455–2472 (cit. on pp. xiii, xv, 2, 6, 8, 14, 17, 48, 75–80, 83, 104, 105, 107–109, 126, 128, 134, 135, 139, 147, 148, 150).

[RM20] Venkatesh-Prasad Ranganath and Joydeep Mitra. "Are free android app security analysis tools effective in detecting known vulnerabilities?" In: *Empirical Software Engineering*. Vol. 25. 1. Springer, 2020 (cit. on pp. 106, 107).

[Rei+17] Michael Reif, Michael Eichberg, Ben Hermann, and Mira Mezini. “Hermes: assessment and creation of effective test corpora”. In: *6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis (SOAP)*. ACM, 2017 (cit. on p. 114).

[Ren+20] Xiaoxue Ren, Jiamou Sun, Zhenchang Xing, Xin Xia, and Jianling Sun. “Demystify Official API Usage Directives with Crowdsourced API Misuse Scenarios, Erroneous Code Examples and Patches”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ICSE ’20. Seoul, South Korea: Association for Computing Machinery, June 2020, pp. 925–936 (cit. on p. 53).

[Rep00] Thomas Reps. “Undecidability of Context-Sensitive Data-Dependence Analysis”. In: *ACM Trans. Program. Lang. Syst.* 22.1 (Jan. 2000) (cit. on p. 112).

[Res] Reshift. Last accessed: 2026-06-06. At the time of access, the original webpage was no longer available and redirected to an unrelated sports live-scoring website. (cit. on p. 26).

[RIG] RIGS IT GmbH. *Xanitizer*. en-US. `https://www.testingtoolsguide.net/tools/xanitizer/`. Last accessed: 2026-06-06. At the time of access, the RIGS IT webpage was no longer available. (cit. on p. 107).

[RH] Joaquín Rinaudo and Juan Heguiabehere. *Marvin Static Analyzer*. `https://github.com/programa-stic/Marvin-static-Analyzer`. Last accessed: 2026-06-06 (cit. on p. 107).

[RY20] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020 (cit. on p. 2).

[Rob+13] Martin P. Robillard, Eric Bodden, David Kawrykow, Mira Mezini, and Tristan Ratchford. “Automated API Property Inference Techniques”. In: *IEEE Transactions on Software Engineering* 39.5 (May 2013), pp. 613–637 (cit. on pp. 49, 56, 58–60, 62–64).

[Ros] Roslynator. `https://github.com/JosefPihrt/Roslynator`. Last accessed: 2026-06-06 (cit. on p. 26).

[SDL80] Frank E Saal, Ronald G Downey, and Mary A Lahey. “Rating the ratings: Assessing the psychometric quality of rating data.” In: *Psychological bulletin* 88.2 (1980), pp. 413–428 (cit. on p. 24).

[Sad+18] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. “Lessons from building static analysis tools at google”. In: *Communications of the ACM* 61.4 (2018), pp. 58–66 (cit. on pp. 20, 31, 38, 39, 42).

[Sad+15] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. “Tricorder: Building a program analysis ecosystem”. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Vol. 1. IEEE. 2015, pp. 598–608 (cit. on pp. 11, 12, 18, 20, 38, 75, 81, 83, 90, 93, 101, 172).

[SSD15] M. A. Saied, H. Sahraoui, and B. Dufour. “An observational study on API usage constraints and their documentation”. In: *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Montreal, QC, Canada, Mar. 2015, pp. 33–42 (cit. on pp. 53, 56, 58, 60, 63, 64).

[Sau+25] Rebecca Saul, Hao Wang, Koushik Sen, and David Wagner. *SCGAgent: Recreating the Benefits of Reasoning Models for Secure Code Generation with Agentic Workflows*. 2025. arXiv: `2506.07313 [cs.CR]` (cit. on p. 149).

[Sca+19] Simone Scalabrino, Gabriele Bavota, Mario Linares-Vásquez, Michele Lanza, and Rocco Oliveto. “Data-driven solutions to detect api compatibility issues in android: an empirical study”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE. Montreal, QC, Canada, May 2019, pp. 288–298 (cit. on pp. 54, 56).

[Sch24] Michael Schlichtig. “Building a Framework to Improve the User Experience of Static Analysis Tools”. In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. ICSE-Companion ’24. Lisbon, Portugal: Association for Computing Machinery, 2024, pp. 165–169 (cit. on p. ix).

[Sch+26a] Michael Schlichtig, Aashish Prajapati, Ashwin Prasad Shivarpatna Venkatesh, et al. *Artifact for LLM-SAST$_{Repair}$*. `https://doi.org/10.6084/m9.figshare.32594145`. June 2026 (cit. on pp. 128, 132, 136, 139, 147, 148, 176).

[Sch+26b] Michael Schlichtig, Aashish Prajapati, Ashwin Prasad Shivarpatna Venkatesh, et al. “Can SAST Tools Support LLMs in Automatically Repairing Cryptographic API Misuses?” Unpublished manuscript. 2026 (cit. on pp. xiii, 3, 5, 6, 8, 9, 163, 164, 169, 171, 175).

[Sch+22a] Michael Schlichtig, Steffen Sassalla, Krishna Narasimhan, and Eric Bodden. “FUM - A Framework for API Usage constraint and Misuse Classification”. In: *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2022, pp. 673–684 (cit. on pp. x, 1, 3, 5, 6, 8, 9, 84, 85, 116, 117, 130, 163, 171, 172).

[Sch+22b] Michael Schlichtig, Steffen Sassalla, Krishna Narasimhan, and Eric Bodden. *FUM - A Framework for API Usage constraint and Misuse Classification*. `https://doi.org/10.6084/m9.figshare.16832749`. Jan. 2022 (cit. on pp. 69, 174).

[Sch+26c] Michael Schlichtig, Markus Schmidt, Tom Ohlmer, et al. “*SecAI* — Improving the Usability of Static Application Security Testing”. Unpublished manuscript. 2026 (cit. on pp. xiii, 3, 4, 7, 9, 172).

[Sch+22c] Michael Schlichtig, Anna-Katharina Wickert, Stefan Krüger, Eric Bodden, and Mira Mezini. *CamBench – Cryptographic API Misuse Detection Tool Benchmark Suite*. 2022. arXiv: `2204.06447 [cs.SE]` (cit. on pp. xii–xiv, 3, 5, 6, 8, 9, 126, 128, 131, 135, 139, 140, 148–150, 166, 168, 171).

[SR] Michael Scovetta and Daniel Ruf. *Yasca*. `https://github.com/scovetta/yasca`. Last accessed: 2026-06-06. When we accessed the GitHub page, the project was deprecated. (cit. on p. 107).

[Sec] Security Code Scan. https://security-code-scan.github.io. Last accessed: 2026-06-06 (cit. on pp. 1, 2, 26).

[Sem] Semgrep. https://semgrep.dev. Last accessed: 2026-06-06 (cit. on p. 26).

[Shu+14] Shao Shuai, Dong Guowei, Guo Tao, Yang Tianchang, and Shi Chenjie. "Modelling analysis and auto-detection of cryptographic misuse in android applications". In: *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*. IEEE. 2014, pp. 75–80 (cit. on pp. 48, 116, 119).

[SFM25] André Silva, Sen Fang, and Martin Monperrus. "Repairllama: Efficient representations and fine-tuned adapters for program repair". In: *IEEE Transactions on Software Engineering* (2025) (cit. on pp. 8, 126, 127, 150, 151).

[SDM20] Justin Smith, Lisa Nguyen Quang Do, and Emerson Murphy-Hill. "Why can't johnny fix vulnerabilities: A usability evaluation of static analysis tools for security". In: *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*. 2020, pp. 221–238 (cit. on pp. 2, 3, 7, 12, 20, 21, 28, 31, 38, 42, 45, 154).

[Smi+15] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. "Questions developers ask while diagnosing potential security vulnerabilities with static analysis". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 248–259 (cit. on pp. 32, 33).

[Smi+18] Justin Smith, Brittany Johnson, Emerson Murphy-Hill, Bill Chu, and Heather Richter Lipford. "How developers diagnose potential security vulnerabilities with a static analysis tool". In: *IEEE Transactions on Software Engineering* 45.9 (2018), pp. 877–897 (cit. on p. 28).

[SOG] Senior Officials Group Information Systems Security (SOG-IS). *SOG-IS Crypto Evaluation Scheme Agreed Cryptographic Mechanisms*. https://www.sogis.eu/uk/supporting_doc_en.html. Last accessed: 2026-06-06 (cit. on p. 113).

[Sona] SonarSource S.A. *Sonar Plugin API*. https://docs.sonarsource.com/sonarqube/latest/extension-guide/developing-a-plugin/plugin-basics/. Last accessed: 2026-06-06 (cit. on pp. 9, 153, 154, 156, 157, 166–168).

[Sonb] SonarSource S.A. *SonarQube - Code Quality Tool*. https://www.sonarsource.com/products/sonarqube/. Last accessed: 2026-06-06 (cit. on pp. xiv, 2, 3, 9, 26, 107, 126, 154–159, 161–163, 166, 169).

[Sonc] SonarSource S.A. *SonarQube for IDE*. https://www.sonarsource.com/products/sonarlint/. Last accessed: 2026-06-06 (cit. on pp. 26, 160).

[Sond] SonarSource S.A. *SonarQube rules*. https://docs.sonarsource.com/sonarqube-server/quality-standards-administration/managing-rules/rules. Last accessed: 2026-08-18 (cit. on p. 157).

[Sone] SonarSource S.A. *SonarSource - about*. https://www.sonarsource.com/company/about/. Last accessed: 2026-06-06 (cit. on pp. 153, 156).

[Sonf] SonarSource S.A. *User Guide to Connected Mode of SonarQube for IDE*. https://docs.sonarsource.com/sonarqube-server/user-guide/connected-mode. Last accessed: 2026-06-06 (cit. on p. 160).

[SAB17] Johannes Späth, Karim Ali, and Eric Bodden. "IDE ^{al} : efficient and precise alias-aware dataflow analysis". en. In: *Proceedings of the ACM on Programming Languages* 1.OOPSLA (Oct. 2017), pp. 1–27 (cit. on pp. 17, 88).

[Spä+16] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. "Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java". en. In: *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Vol. 56. LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016, 22:1–22:26 (cit. on p. 88).

[Spi+20] Diomidis Spinellis, Zoe Kotti, Konstantinos Kravvaritis, Georgios Theodorou, and Panos Louridas. "A dataset of enterprise-driven open source software". In: *Proceedings of the 17th International Conference on Mining Software Repositories*. 2020, pp. 533–537 (cit. on pp. 75, 76, 79).

[Spo] Spotbugs. *Spotbugs*. https://spotbugs.github.io/. Last accessed: 2026-06-06 (cit. on pp. 1, 2, 26, 107).

[Spo16] Fausto Spoto. "The Julia Static Analyzer for Java". In: *Static Analysis*. Ed. by Xavier Rival. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016 (cit. on p. 107).

[Sta] JavaScript Standard Style. https://standardjs.com. Last accessed: 2026-06-06 (cit. on pp. 1, 2, 26).

[sta] statcounter. https://gs.statcounter.com/os-market-share/desktop/worldwide. Last accessed: 2026-06-06 (cit. on p. 43).

[Ste+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. "The first collision for full SHA-1". In: *Annual international cryptology conference*. Springer. 2017, pp. 570–596 (cit. on p. 78).

[Sto+21] Andrew van der Stock, Brian Glas, Neil Smithline, and Torsten Gigler. *OWASP Top 10:2021*. en. https://owasp.org/Top10/. Last accessed: 2026-06-06. Sept. 2021 (cit. on p. 108).

[SHA15] Joshua Sushine, James D. Herbsleb, and Jonathan Aldrich. "Searching the State Space: A Qualitative Study of API Protocol Usability". In: *2015 IEEE 23rd International Conference on Program Comprehension*. 2015, pp. 82–93 (cit. on pp. 7, 13, 48, 54, 56, 60, 62, 63, 85).

[Syn] Synopsys Inc. *Coverity Scan - Static Analysis*. https://scan.coverity.com/. Last accessed: 2026-06-06 (cit. on p. 107).

[Tai96] Antero Taivalsaari. "On the Notion of Inheritance". In: *ACM Comput. Surv.* 28.3 (Sept. 1996), pp. 438–479 (cit. on p. 64).

[TB25] Karl Tamberg and Hayretdin Bahsi. "Harnessing Large Language Models for Software Vulnerability Detection: A Comprehensive Benchmarking Study". In: *IEEE Access* 13 (2025), pp. 29698–29717 (cit. on p. 149).

[Tem+10] Ewan Tempero, Craig Anslow, Jens Dietrich, et al. “The Qualitas Corpus: A curated collection of Java code for empirical studies”. In: *Asia Pacific Software Engineering Conference (APSEC)*. IEEE, 2010 (cit. on pp. 8, 110, 112, 114).

[Tho+16] Tyler W Thomas, Heather Lipford, Bill Chu, Justin Smith, and Emerson Murphy-Hill. “What questions remain? an examination of how developers understand an interactive static analysis tool”. In: *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*. 2016 (cit. on pp. 20, 27, 28, 32).

[TIO] TIOBE Software. *TIOBE-Index: The Java Programming Language*. `https://www.tiobe.com/tiobe-index/java/`. Last accessed: 2026-06-06 (cit. on pp. 22, 49).

[THZ19] Ayal Tirosh, Mark Horvarth, and Dionisio Zumerle. *Magic Quadrant for Application Security Testing*. Tech. rep. Gartner, Inc, 2019 (cit. on p. 22).

[Tor+23] Adriano Torres, Pedro Costa, Luis Amaral, et al. “Runtime Verification of Crypto APIs: An Empirical Study”. In: *IEEE Transactions on Software Engineering* 49.10 (2023), pp. 4510–4525 (cit. on pp. 14, 105, 107, 139, 140).

[Tri+14] Omer Tripp, Salvatore Guarnieri, Marco Pistoia, and Aleksandr Aravkin. “Aletheia: Improving the usability of static security analysis”. In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. 2014, pp. 762–774 (cit. on pp. 35, 165).

[Val+10] Raja Vallée-Rai, Phong Co, Etienne Gagnon, et al. “Soot: a Java bytecode optimization framework”. en. In: *CASCON First Decade High Impact Papers on - CASCON ’10*. Toronto, Ontario, Canada: ACM Press, 2010, pp. 214–224 (cit. on pp. 17, 88, 156).

[VBS+94] Maarten W Van Someren, Yvonne F Barnard, Jacobijn AC Sandberg, et al. “The think aloud method: a practical approach to modelling cognitive processes”. In: *London: AcademicPress* 11.6 (1994) (cit. on p. 98).

[Vas+18] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, et al. “Context is king: The developer perspective on the usage of static analysis tools”. In: *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2018, pp. 38–49 (cit. on pp. 20, 29).

[Vau02] Serge Vaudenay. “Security flaws induced by CBC padding—applications to SSL, IPSEC, WTLS...” In: *International Conference on the Theory and Applications of Cryptographic Techniques*. Springer. 2002, pp. 534–545 (cit. on p. 78).

[Vis] VisualCodeGrepper. `https://github.com/nccgroup/VCG`. Last accessed: 2026-06-06 (cit. on pp. 26, 107).

[VSD] VSDiagnostics. `https://github.com/Vannevelj/VSDiagnostics`. Last accessed: 2026-06-06 (cit. on p. 26).

[Vul] Vulture. `https://pypi.org/project/vulture/`. Last accessed: 2026-06-06 (cit. on p. 26).

[WRO14] Fengguo Wei, Sankardas Roy, and Xinming Ou. “Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps”. In: *2014 ACM SIGSAC conference on Computer and Communications security (CCS)*. ACM, 2014 (cit. on pp. 107, 109).

[wem] wemake-python-styleguide. `https://github.com/wemake-services/wemake-python-styleguide`. Last accessed: 2026-06-06 (cit. on p. 26).

[Whi+23] Jules White, Quchen Fu, Sam Hays, et al. “A Prompt Pattern Catalog to Enhance Prompt Engineering with ChatGPT”. In: *Proceedings of the 30th Conference on Pattern Languages of Programs (PLoP 2023)*. The Hillside Group, 2023. arXiv: `2302.11382 [cs.SE]` (cit. on p. 133).

[Wic] Dave Wichers. *OWASP Benchmark | OWASP Foundation*. en. `https://owasp.org/www-project-benchmark/`. Last accessed: 2026-06-06 (cit. on pp. 8, 107, 108).

[Wic25] Anna-Katharina Wickert. “Mitigating Security Risks by Understanding Security-related API-misuses and Advancing Detection of Misuses”. Dr.-Ing. dissertation. Technical University of Darmstadt, Germany, 2025 (cit. on p. 116).

[Wic+21] Anna-Katharina Wickert, Lars Baumgärtner, Florian Breitfelder, and Mira Mezini. “Python crypto misuses in the wild”. In: *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 2021, pp. 1–6 (cit. on pp. 101, 104).

[Wic+22] Anna-Katharina Wickert, Lars Baumgärtner, Michael Schlichtig, Krishna Narasimhan, and Mira Mezini. “To Fix or Not to Fix: A Critical Study of Crypto-misuses in the Wild”. In: *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. 2022, pp. 315–322 (cit. on pp. xi, 1, 5, 9, 74, 79, 85, 86, 90, 93, 116, 119, 164, 172, 174).

[Wic+19] Anna-Katharina Wickert, Michael Reif, Michael Eichberg, Anam Dodhy, and Mira Mezini. “A Dataset of Parametric Cryptographic Misuses”. In: *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. Montreal, QC, Canada, May 2019, pp. 96–100 (cit. on pp. 8, 14, 105, 107, 116, 119).

[Wic+24] Anna-Katharina Wickert, Michael Schlichtig, Marvin Vogel, et al. “Supporting Error Chains in Static Analysis for Precise Evaluation Results and Enhanced Usability”. In: *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 2024, pp. 693–704 (cit. on pp. xi, 5, 9, 84, 97, 117, 131, 162, 163, 169, 172, 175).

[WA19] Chamila Wijayarathna and Nalin Asanka Gamagedara Arachchilage. “Using cognitive dimensions to evaluate the usability of security APIs: An empirical investigation”. In: *Information and Software Technology* 115 (2019), pp. 5–19 (cit. on p. 79).

[Wil] Wily. `https://github.com/tonybaloney/wily`. Last accessed: 2026-06-06 (cit. on p. 26).

[Woh+12] Claes Wohlin, Per Runeson, Martin Höst, et al. *Experimentation in software engineering*. Vol. 236. Springer, 2012 (cit. on p. 147).

[Won+16] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. "A Survey on Software Fault Localization". en. In: *IEEE Transactions on Software Engineering* 42.8 (Aug. 2016), pp. 707–740 (cit. on p. 100).

[Wu+17] Jingzheng Wu, Shen Liu, Shouling Ji, et al. "Exception beyond Exception: Crashing Android System by Trapping in" Uncaught Exception"". In: *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE. 2017, pp. 283–292 (cit. on p. 78).

[Wu+21] Zonghan Wu, Shirui Pan, Fengwen Chen, et al. "A Comprehensive Survey on Graph Neural Networks". In: *IEEE Transactions on Neural Networks and Learning Systems* 32.1 (2021), pp. 4–24 (cit. on pp. 165, 166).

[Xen] Xenon. `https://xenon.readthedocs.io/en/latest/`. Last accessed: 2026-06-06 (cit. on p. 26).

[XZ22] Chunqiu Steven Xia and Lingming Zhang. "Less training, more repairing please: revisiting automated program repair via zero-shot learning". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 959–971 (cit. on p. 150).

[Xie+11] Jing Xie, Bill Chu, Heather Richter Lipford, and John T Melton. "Aside: Ide support for web application security". In: *Proceedings of the 27th Annual Computer Security Applications Conference*. 2011, pp. 267–276 (cit. on p. 41).

[xo] xo. `https://github.com/xojs/xo`. Last accessed: 2026-06-06 (cit. on p. 26).

[XYSEC] XYSEC Labs. *Devknox - Security Plugin for Android Studio*. `https://devknox.io/`. Last accessed: 2026-06-06 (cit. on p. 107).

[Yam+14] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. "Modeling and discovering vulnerabilities with code property graphs". In: *2014 IEEE symposium on security and privacy*. IEEE. 2014, pp. 590–604 (cit. on p. 165).

[ZAP] OWASP ZAP. *OWASP ZAP*. `https://github.com/zaproxy/zaproxy`. Last accessed: 2026-06-06 (cit. on p. 107).

[Zha+19] Li Zhang, Jiongyi Chen, Wenrui Diao, et al. "{CryptoREX}: Large-scale analysis of cryptographic misuse in {IoT} devices". In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. 2019, pp. 151–164 (cit. on pp. 8, 104).

[Zha+23] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. "A Survey of Learning-based Automated Program Repair". In: *ACM Trans. Softw. Eng. Methodol.* 33.2 (Dec. 2023) (cit. on p. 129).

[Zha+24] Quanjun Zhang, Chunrong Fang, Bowen Yu, et al. "Pre-Trained Model-Based Automated Software Vulnerability Repair: How Far are We?" In: *IEEE Transactions on Dependable and Secure Computing* 21.4 (July 2024), pp. 2507–2525 (cit. on pp. 126, 150).

[Zha+22] Ying Zhang, Md Mahir Asef Kabir, Ya Xiao, Danfeng Yao, and Na Meng. “Automatic detection of Java cryptographic API misuses: Are we there yet?” In: *IEEE Transactions on Software Engineering* 49.1 (2022), pp. 288–303 (cit. on pp. 116, 119).

[Zha+26] Wayne Xin Zhao, Kun Zhou, Junyi Li, et al. “A survey of large language models”. In: *Frontiers of Computer Science* 20.12 (2026), p. 2012627 (cit. on p. 164).

[ZM19] H. Zhong and H. Mei. “An Empirical Study on API Usages”. In: *IEEE Transactions on Software Engineering* 45.4 (Dec. 2019), pp. 319–334 (cit. on pp. 51, 56, 60, 62).

[Zhu+15] Jun Zhu, Bill Chu, Heather Lipford, and Tyler Thomas. “Mitigating access control vulnerabilities through interactive static analysis”. In: *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*. 2015, pp. 199–209 (cit. on p. 35).

List of Figures

List of Tables

Listings