Secure Coding Practice in Java: Automatic Detection, Repair, and Vulnerability Demonstration

Ying Zhang

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Science and Application

Na Meng, Chair
Danfeng (Daphne) Yao
Matthew Hicks
Muhammad Ali Gulzar
Francisco Servant

September 20, 2023
Blacksburg, Virginia

Keywords: API insecure usages, Static analysis, Repair, Cryptography, Supply chain attack, ChatGPT-4.0, Test generation.

Secure Coding Practice in Java: Automatic Detection, Repair, and Vulnerability Demonstration

Ying Zhang

ABSTRACT

The Java platform and third-party open-source libraries provide various Application Programming Interfaces (APIs) to facilitate secure coding. However, using these APIs securely is challenging for developers who lack cybersecurity training. Prior studies show that many developers use APIs insecurely, thereby introducing vulnerabilities in their software. Despite the availability of various tools designed to identify API insecure usage, their effectiveness in helping developers with secure coding practices remains unclear. This dissertation focuses on two main objectives: (1) exploring the strengths and weaknesses of the existing automated detection tools for API-related vulnerabilities, and (2) creating better tools that detect, repair, and demonstrate these vulnerabilities.

Our research started with *investigating the effectiveness of current tools in helping with developers' secure coding practices.* We systematically explored the strengths and weaknesses of existing automated tools for detecting API-related vulnerabilities. Through comprehensive analysis, we observed that most existing tools merely report misuses, without suggesting any customized fixes. Moreover, developers often rejected tool-generated vulnerability reports due to their concerns on the correctness of detection, and the exploitability of the reported issues. To address these limitations, the second work proposed Seader*, an example-based approach to detect and repair security-API misuses.* Given an exemplar ⟨insecure, secure⟩ code pair, Seader compares the snippets to infer any API-misuse template and corresponding fixing edit. Based on the inferred information, given a program, Seader performs interprocedural static analysis to search for security-API misuses and to propose customized fixes. The third work *leverages ChatGPT-4.0 to automatically generate security test cases.* These

test cases can demonstrate how vulnerable API usage facilitates supply chain attacks on specific software applications. By running such test cases during software development and maintenance, developers can gain more relevant information about exposed vulnerabilities, and may better create secure-by-design and secure-by-default software.

Secure Coding Practice in Java: Automatic Detection, Repair, and Vulnerability Demonstration

Ying Zhang

GENERAL AUDIENCE ABSTRACT

The Java platform and third-party open-source libraries provide various Application Programming Interfaces (APIs) to facilitate secure coding. However, using these APIs securely can be challenging, especially for developers who aren't trained in cybersecurity. Prior work shows that many developers use APIs insecurely, consequently introducing vulnerabilities in their software. Despite the availability of various tools designed to identify API insecure usage, it is still unclear how well they help developers with secure coding practices.

This dissertation focuses on (1) exploring the strengths and weaknesses of the existing automated detection tools for API-related vulnerabilities, and (2) creating better tools that detect, repair, and demonstrate these vulnerabilities. We first systematically evaluated the strengths and weaknesses of the existing automated API-related vulnerability detection tools. We observed that most existing tools merely report misuses, without suggesting any customized fixes. Additionally, developers often reject tool-generated vulnerability reports due to their concerns about the correctness of detection, and whether the reported vulnerabilities are truly exploitable. To address the limitations found in our study, the second work proposed a novel example-based approach, Seader, to detect and repair API insecure usage. The third work leverages ChatGPT-4.0 to automatically generate security test cases, and to demonstrate how vulnerable API usage facilitates the supply chain attacks to given software applications.

Dedication

Dedicated to my family

Acknowledgments

I am incredibly fortunate to have crossed paths with the finest and most fitting individuals at each juncture of my journey. Their company and love along the way have helped me become a better one. I am wholeheartedly grateful to all of them for their unwavering support.

First and foremost, I would like to express my sincere gratitude to my advisor, Dr. Na Meng. I will forever cherish the moment she offered me an opportunity to join her lab during that challenging summer. Over the past years, she has always been patient, providing timely feedback and guidance for any questions I have had. Her expertise guided my research perspective from static program analysis to software security and testing, helping me gain deeper insights into research and my own capabilities. She has been incredibly supportive and encouraged me to explore my research interests. Her rigorous and dedicated attitude toward research, optimism and confidence in confronting any challenges has illuminated my academic path and life and inspired me to chase higher.

I also want to sincerely thank my collaborator Dr. Danfeng (Daphne) Yao, for sharing her profound expertise in computer security and having invaluable discussions that significantly enriched my research.

My gratitude also goes to my committee members: Dr. Matthew Hicks, Dr. Muhammad Ali Gulzar, and Dr. Francisco Servant. Thank you for your discernment of my research topics and timely feedback that guided me to improve my work.

Special appreciation is extended to my friends and colleagues. They have generously shared research insights, provided technical support, and given encouragement throughout this journey.

I'm particularly thankful to my industrial mentors, Dr. Peng Li and Dr. Yu Ding. Their

vast experience in Rust, symbolic execution, testing, and application security has broadened my perspective on software security. They provided incisive guidance during my internship. Lastly, I want to thanks my family for their unconditional love and support. Even when the pandemic kept us apart for four years, their understanding and love never wavered and remain the greatest gifts I have in my life. A special mention to Lingxiang - his distinct perspective on software engineering provided many interesting angles to my research. His constant care has been a source of strength during the most challenging times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation - APIs Insecure Usage

The Java platform, along with numerous third-party Java frameworks, provides a variety of Application Programming Interfaces (APIs) for simplifying application development. To facilitate the development, developers often integrate various functionalities by calling Java APIs from popular open-source libraries and components [173]. The wide availability and extensive adoption of Java APIs have become integral to efficient software development.

However, these APIs are not easy to use for two reasons. First, many APIs are overly complicated but poorly documented [117, 144]. Second, developers lack the necessary cybersecurity training; they are unaware of the security implications of software configuration and coding options (e.g., the parameter values, calling sequences, or outdated dependencies with vulnerable APIs) [3, 7, 140]. Consequently, many developers use the APIs insecurely, consequently introducing vulnerabilities to their software [106, 159]. For instance, Fischer et al. found that cryptographic API insecure usage posted on StackOverflow was replicated in 196,403 Android applications on the Google Play Store [106]. Additionally, Snyk [14] and Williams et al. [173] showed that around 67% of real-world Java projects incorporate dependencies with known vulnerable API(s). This insecure API usage presents a high risk, as cyber-attackers can leverage API-related vulnerabilities to compromise systems built upon these libraries [104, 115].

To help developers identify the insecure usage of APIs, researchers and engineers created many tools to detect invocation of vulnerable APIs or insecure usage of APIs [30, 36, 41, 101, 107, 127, 157, 158, 165, 176]. These automatic tools vary from design to implementation. They leverage either program static analysis or machine learning-based approaches to check whether APIs are called with insecure parameter values or in wrong sequential orders. However, due to the lack of systematic evaluation of the tools' effectiveness from both the theoretical side and developers' perspectives, it is unknown what the strengths and weaknesses of existing tools to mitigate API insecure usages and whether we need to create new tools to better help with secure coding practices.

1.2 Dissertation Statement

For this dissertation, we 1) explored the strengths and weakness of current automatic tools for detecting API insecure usage, from both theoretical and practical perspectives, and 2) created new approaches to help developers eliminate insecure API usage. We intended to create an effective framework that can accurately locate API misuses, offer fix guidance to developers, and demonstrate API-relevant vulnerabilities. This framework can help engineers conduct secure Java coding practices. To achieve our goal, we ask the following questions:

- How effectively do the current tools detect API insecure usage?
- How can we help developers better address API insecure usage?

To investigate the research questions, we conducted an empirical study focusing on automatic detection tools that target cryptographic API insecure usage. This study reveals the following limitations of the current state-of-art from the developer's perspective:

1. Most existing tools use pre-defined rules, lacking extensibility in detection and lacking guidance for customized fixes. To overcome the limitations, we designed and implemented an example-based approach to detect and repair insecure usage of APIs. This approach adopted intra-procedural analysis to derive a vulnerability-repair pattern. Based on the inferred patterns, our approach performs inter-procedural static analysis on a given program to identify the API insecure usage, concretizes the corresponding abstract fixes, and suggests code replacements to developers.

2. Existing tools fail to provide demonstrations of the detected vulnerabilities related to API insecure usage. To overcome this limitation, we conducted a novel study to explore generating security tests for software applications (Apps) that depend on vulnerable libraries (Libs) with ChatGPT-4.0. We crafted distinct prompt templates and concretized them with a full or subset of vulnerable API-relevant information(e.g., vulnerable API(s), vulnerability ID, the API calling context in App, and a proof-of-concept exploit from the Lib), asking ChatGPT-4.0 to generate proof-of-concept security tests for various Apps.Wemanually validate whether the test cases (i) propagate vulnerabilities from libraries into Apps via calls to vulnerable library APIs and (ii) trigger abnormal App behaviors.

I will further introduce my research work in the next three sections.

1.3 Understanding Pros and Cons of the Current Automatic Vulnerability Detection Tools

To reveal the gap between state-of-the-art tools and developers' expectations, the empirical study explores the following research questions (1) how current tools are designed to detect

cryptographic API misuses[1], (2) how effectively the tools work to locate API misuses and (3) how developers perceive the usefulness of tools' outputs. We systematically and comprehensively compared the design approaches of various tools, focusing on aspects including API usage pattern representations, pattern-matching logic, input/output information, and public availability. Building on this, we evaluated the effectiveness of six publicly available, executable, and comparable tools. To delve deeper into the practical aspect, we conducted a novel study with developers: we applied these tools to 200 open-source Apache projects. We described the security vulnerabilities reported by our selected tools to the developers. By manually inspecting developers' feedback and related program context, we observed several limitations of the proposed tools:

Lack of support for detection extensibility and vulnerability repair guidance. Current tools typically identify cryptographic API secure usage with built-in rules. With security libraries' constant emergence and evolution, vulnerability detectors should have good extensibility to keep their pattern sets of API misuses up-to-date. Additionally, Most existing tools merely report misuses and do not suggest any customized fixes or provide vague and incomplete fixing suggestions. When developers lack the cybersecurity knowledge to understand the reported misuses, they may continue making mistakes when fixing those issues independently [170].

Lack of demonstration of the detection correctness. More Developers rejected tools' reports than accepted (30 vs. 9) due to their concerns about the correctness of tools reported issues. Developers need the demonstration of those reported vulnerabilities. Some of the API insecure usages detected by tools were in test code or irrelevant program contexts that developers believed would not cause security issues. As a result, developers rejected many of the bug reports from the tools.

[1]API misuses and API Insecure Usage are interchangeable in this dissertation.

1.4 Automatic Vulnerability Detection and Fix related to APIs Insecure Usage

To address the limitations of existing tools, we developed a novel data-driven tool called Seader for detecting vulnerabilities and suggesting fixes. There are two phases in Seader: pattern inference and pattern application. In Phase I, given ⟨insecure, secure⟩ code examples, Seader compares the two code snippets and identifies program changes from the example code pair. Next, based on those changes, Seader conducts intra-procedural analysis to derive a vulnerability-repair pattern. In Phase II, given a program, Seader loads patterns from the JSON file and conducts inter-procedural program analysis to match code with templates. For each code match, Seader suggests code replacements with correct code contexts to developers.

Seader allows users to specify examples of API misuse in specialized ways, such as defining misuses related to arbitrary constant parameters, parameters within certain ranges, and parameters with specific requirements. Seader strengthens the expressiveness of example-based pattern specification. We further applied Seader to two program datasets to evaluate its effectiveness in vulnerability detection and repair. When applied to the first dataset, Seader detected vulnerabilities with 95% precision, 72% recall, and 82% F-score. After applying Seader to the second dataset, we inspected 77 repairs output by Seader and found 76 of them correct.

1.5 Security Tests Generation with ChatGPT-4.0

To address the second limitation observed in the study, provide a demonstration of the detected vulnerabilities related to API insecure usage, and help developers gain more relevant

information about exposed vulnerabilities, we conducted a novel study to explore generating security tests for software Apps that depend on vulnerable Libs. Our study explored: 1) How effectively ChatGPT-4.0 can generate security tests. 2) How ChatGPT-4.0 performs when different combinations of information are included in the prompts we created. 3) How ChatGPT-4.0 works compared with existing tools for security test generation.

To answer these questions, firstly, we created a dataset including 25 libraries and 55 applications; each App depends on a version of a library with a known vulnerability. Secondly, we crafted distinct prompt templates with domain knowledge and concretized these prompt templates with a full or subset of vulnerable API descriptive information. The descriptive information includes vulnerable APIs, vulnerability ID, the API calling context within the application, and a proof-of-concept exploit test from the library. Using this prompt, we asked ChatGPT-4.0 to generate a test for the App by mimicking the given test. We then manually verified each generated security test to see if they 1) invoke vulnerable APIs made by Apps, (2) trigger the problematic behaviors of Apps, and (3) fail when reported vulnerabilities are not fixed. Thirdly, we applied two state-of-the-art tools—SIEGE and TRANSFER—to the dataset we had created and compared the performance of ChatGPT-4.0 against both of these tools.

Surprisingly, we observed, given prompts that cover all relevant vulnerable API-relevant information, ChatGPT-4.0 successfully generated tests for 40 out of 55 Apps, demonstrating 24 attacks. It outperformed two state-of-the-art security test generators, TRANSFER and SIEGE, by generating a significantly higher number of tests and achieving more exploits. ChatGPT-4.0 worked better when prompts described more on the vulnerabilities, possible exploits, and code context.

1.6 Contribution

This dissertation has the following contributions:

Clarified of the strengths and weaknesses of the existing automatic API insecure usage detection tools incorporated with developers' perspectives (TSE 2022)

- We analyzed the approach designs of existing detectors for cryptographic API misuses. Most tools represent patterns as built-in rules, conduct inter-procedural analysis, and only report detected API misuses.
- We conducted a novel study with developers by describing vulnerabilities reported by current tools. Developers wanted to address the issues but needed more detailed edit suggestions and configuration fixes.
- By manually inspecting developer feedback and code context, we found disagreements on the criteria used to recognize vulnerabilities. Developers needed tools to demonstrate exploits and skip irrelevant contexts like tests.

Built a novel security API misuses detection and fix tool that takes into account both extensibility and capability. (ICPC 2022)

- We developed Seader—a new approach that performs *intra-procedural analysis* to infer vulnerability-repair patterns from ⟨insecure, secure⟩ code examples, and *inter-procedural analysis* to match code with vulnerability templates. It then customizes abstract fixes to suggest repairs. Seader is the first tool to combine intra- with inter-procedural analysis in this way.

- Key features of SEADER include supporting specialized example specifications that enable defining misuse examples related to arbitrary constants, ranged constants, and alternative constants.
- We evaluated SEADER comprehensively and found it achieves higher vulnerability detection F-scores than three state-of-the-art tools. For repair suggestion, SEADER achieved 99% (76/77) accuracy.

Generate security tests for Applications with vulnerable dependencies demonstrating proof-of-concept exploits using ChatGPT-4.0. (FSE 2024 under submission)

- We conducted a novel study to explore the effectiveness of ChatGPT in vulnerability test generation for known vulnerabilities in Apps. Our findings demonstrated that ChatGPT can generate unit tests that reveal known vulnerabilities with few-shot prompts.
- Our evaluation showed that ChatGPT-4.0 surpasses existing tools in capability and effectiveness in security test generation.
- Our findings reveal that ChatGPT's effectiveness can be enhanced with detailed instructions and relevant background knowledge.

Dissertation Structure: The rest of this dissertation is organized as follows. Chapter 2: Introduces the research background and delves into related work to this dissertation. Chapter 3 details our first research project, which comprehensively evaluates the automatic detection tool for detecting API insecure usage. Chapter 4 presents our tool, SEADER, designed to extend the pattern set of API-misuse detectors and concretizes security expertise

as customized fixing edits for developers. Chapter 5 explores ChatGPT to generate security test cases and demonstrates how vulnerable library dependencies facilitate the supply chain attacks to given Apps. In the end, Chapter 6 summarizes the research contributions and future research directions.

Chapter 2

Related Work

2.1 Empirical study on APIs Insecure Usage

Several studies are relevant to our work [90, 92, 113, 114, 150, 170]. For instance, Gao et al. applied CogniCrypt to different versions of Android apps; they observed that app developers are generally unaware of cryptographic API misuses and hence usually do not fix such issues [113]. Gao et al. also performed another empirical study on the evolution of Android app vulnerabilities [114]. The researchers found that (1) most vulnerabilities could survive at least three app updates; (2) part of third-party libraries were the major contributors of most vulnerabilities; (3) all kinds of vulnerabilities were reintroduced by developers, while encryption-related ones were reintroduced most often; (4) some vulnerabilities may foreshadow malware. Our study is different from both studies, as it compares tools that automatically detect cryptographic API misuse and investigates developers' feedback on tool outputs.

Amann et al. [92] did a systematic evaluation of static API-misuse detectors. They qualitatively compared 12 existing detectors. They also applied four of the studied tools to MUBench, to evaluate detection capabilities and analyze the root causes for low precision and recall. However, none of the studied tools focus on cryptographic API misuse.

Afrose et al. [90] empirically compared four tools (FindSecBugs, also named as SpotBugs,

CryptoGuard, CogniCrypt, also known as CrySL, and Coverity) on two program benchmarks. They observed that CryptoGuard and CogniCrypt cover more rules and detect more API misuses than the other tools. None of the tools supports path-sensitive analysis. We observed a similar phenomenon concerning the effectiveness comparison among FindSecBugs, CryptoGuard, and CogniCrypt. However, our research is different mainly in two aspects. First, it involves a systematic review of a lot more tools (i.e., 20 tools), and characterizes the design of existing detectors for security API misuses. Second, we conducted a user study and obtained developers' valuable feedback on tool outputs.

Oyetoyan et al. [150] studied static application security testing (SAST) tools and explored developers' opinions on those tools. The researchers applied five open-source tools (i.e., SonarQube, FindSecBugs, Lapse+ [154], JLint [31], and FindBugs [120]) and a commercial tool to two program benchmarks: OWASP Benchmark and NIST Test Suite [1]. They also interviewed six developers to understand the desired features in SAST tools. They reported similar findings to ours, including (1) one tool is not enough to cover all weakness categories and (2) the capability of current tools is generally low. Our research is different in two aspects. First, we focused on cryptographic API misuses; Oyetoyan et al. focused on 13 weakness categories (e.g., code quality), many of which are irrelevant to security. Second, our user study with developers is more rigorous and representative. We interacted with more developers (47 vs. 6) on concrete API misuses and general repair suggestions.

Tupsamudre et al. [170] surveyed four SAST tools (FindSecBugs, SonarQube, CryptoGuard, and CogniCrypt) to explore (1) how tools detect password storage vulnerabilities, and (2) whether the tool-generated fixes comply with the guidelines by OWASP or NIST. The researchers found that none of the tools covered all vulnerabilities related to password storage, and the tools' suggestions were either imprecise or inconsistent with the latest guidelines. They also did a study with eight developers, asking each developer to replace insecure SHA-1-

based password storage implementation with the PBKDF2 solution suggested by tools. The results show that, in the absence of examples, developers chose insecure values for PBKDF2 parameters (salt, iteration count, key length). Thus, although the usage of PBKDF2 matches tools' suggestions, The resulting password storage code may be insecure in practice.

Our research corroborates the findings mentioned in Tupsamudre et al.'s work but is different in two ways. First, we studied more tools, conducted more experiments, and examined more API-misuse patterns; thus, our work has a wider and deeper scope. Second, there are more participants in our user study (47 vs. 8); they provided feedback on not only tool outputs but also future directions. Thus, we revealed more challenges and research opportunities.

2.2 Detection of Security-API Misuses

Tools were built to detect security-API misuses [30, 36, 103, 104, 106, 118, 127, 128, 132, 133, 159]. As shown in Table 4.1, most tools statically analyze programs based on hardcoded or built-in rules. Specifically, CryptoLint hardcoded six API misuse patterns. For each located potentially vulnerable API call (e.g., `Cipher.getInstance(v)`), CryptoLint conducts backward slicing to decide whether the used parameter value is insecure (e.g., `v="AES/ECB"`). CDRep reimplements the design of CryptoLint for misuse detection. It also repairs detected misuses leveraging manually created patch templates. Such tools are not easy to extend because tool builders or users have to modify tool implementation to expand the rulesets of vulnerabilities.

Fischer et al. [106] built a tool to detect misuses in two ways: machine learning and graph matching. Both methods detect vulnerabilities based on the similarity between given programs and labeled (in)secure code. However, this tool does not rigorously reason about misuse patterns; it cannot pinpoint the exact location of misused API in vulnerable code.

CogniCrypt [127] supports rule definition via a domain-specific language CrySL [128]. Each CrySL rule specifies **correct** API usage, and CogniCrypt detects misuses by scanning programs for rule violations. CogniCrypt has three limitations. First, manually prescribing rules with CrySL can be tedious and error-prone for tool users. Second, CrySL cannot express the API misuses related to constant placeholders and constants within certain ranges. Third, CogniCrypt does not customize fixes.

VuRLE [133] is most relevant to our work. VuRLE also detects and repairs vulnerabilities based on the $\langle I, S \rangle$ code examples provided by users.

Seader complements VuRLE in three ways. First, Seader infers each pattern from one instead of multiple code pairs, so it works well when users have only one pair. Second, Seader conducts inter-procedural analysis and adopts succinct info (security APIs and data dependencies) to match code with templates, while VuRLE uses intra-procedural analysis and tree matching. Thus, Seader can find more matches. Third, Seader supports specialized example definitions, and VuRLE does not. As the source code of VuRLE is not publicly available, we cannot empirically compare it with Seader.

2.3 Example-Based Program Transformation

Based on the insight that developers modify similar code in similar ways, researchers built tools to infer program transformations from exemplar code change examples and to manipulate code or suggest changes accordingly [94, 138, 139, 142, 155, 161, 175]. For instance, Given one or multiple code change examples, LASE [139] and REFAZER [161] infer a program transformation from the examples; they then use the transformation to locate similar code to edit and apply customized transformations to those locations. Given a set of vulnerable and patched code fragments $K = \{(A_1, A'_1), (A_2, A'_2), \ldots, (A_n, A'_n)\}$, SecureSync [155]

scans the source code of programs to find fragments, which are similar to vulnerable code A_i but dissimilar to the patched code $A'_i (i \in [1, n])$.

Sharing the same insight, we designed SEADER to detect and fix vulnerabilities based on code examples. However, SEADER is different from prior work for two reasons. First, SEADER infers a program transformation via *intra-procedural* analysis but conducts pattern matching via *inter-procedural* analysis. All the tools mentioned above are limited to intra-procedural analysis. Our unique design makes SEADER more powerful when it searches for pattern matches; it can find matches that go beyond the method boundary and span multiple Java methods. Second, SEADER supports three specialized ways of example specification, which can describe transformations that are not expressible via plain code examples. Based on our experience with security-API misuses, these unique specification methods are necessary and helpful.

2.4 Detection of Vulnerable Dependencies

People built a variety of tools to detect vulnerable dependencies in software projects [5, 10, 51, 55, 70, 76, 79, 160]. For instance, npm-audit [70], snyk-test [79], AuditJS [51], RetireJS [76], and gammaray [5] scan JS applications for their package/library dependencies, compare those packages as well as versions against the known package versions in vulnerability databases (e.g., NVD [71]), and report a vulnerability for each found match. OWASP Dependency-Check [10] and SwiftDependencyChecker [160] implement the same technique, to separately reveal vulnerable dependencies in Java and Swift programs. Dependabot [55] provides the same technical support for multiple programming languages, including C#, Go, Java, JS, PHP, Python, Ruby, Scala, and TypeScript.

However, recent studies and articles show that developers did not trust many of the vulnera-

bilities reported by these tools [22, 42, 124, 153]. For example, Some reported vulnerabilities are not exploitable, as they exist in development-only or test dependencies and never get deployed as parts of released software products [124, 129, 152]. Some vulnerability reports are false alarms because vulnerability databases (e.g., NVD) incorrectly mark library versions [153] or the vulnerable APIs are never called [42, 157]. Developers would like to see how vulnerabilities can be exploited, before fixing the reported vulnerabilities [22, 124]. Our research is motivated by developers' dissatisfaction with existing dependency checkers.

2.5 Security Test Generation

Various tools were built to generate security tests [45, 72, 91, 95, 96, 97, 102, 112, 116, 135, 141, 168, 174]. Specifically, Marback et al. [135] and Xu et al. [174] created approaches, to (partially) automate the procedure of generating security tests from threat models (e.g., threat trees or nets). Namely, these approaches first reveal potential attack paths by automatically traversing hand-crafted threat models, and then convert those paths to executable test code via tool automation or manual effort. However, these approaches require lots of manual effort and domain knowledge. If a user is unable to precisely model all potential threats/attacks or to accurately convert attack paths to executable tests, these approaches are ineffective in creating tests.

Traditional verification takes a program and a specification of safety as inputs and verifies whether the program satisfies the safety specification. Automatic exploit generation (AEG) [95, 96, 97, 112] twists program verification, by replacing typical safety properties with an exploitability property, and the verification process becomes one of finding a program path where the exploitability property holds. For instance, Ganapathy et al. [112] explore API-level exploitability with bounded model checking (BMC). An API-level exploit

means that an attacker can compromise the security of a software component, by calling a sequence of APIs allowed by the component. Ganapathy et al. feed BMC with (1) API specification, (2) safety condition, and (3) integer bound, to decide whether any API sequence shorter than the integer bound violates the safety policy. Brumley et al. [96] created a tool to compare a vulnerable program P against its patched version P', and apply static and/or dynamic symbolic execution to generate a malicious input that is accepted by P but rejected by P'.

AEG often suffers from scalability challenges (e.g., path explosion and the NP-hardness of solving SMT queries in general). To overcome the limitations, people also proposed fuzzing (i.e., fuzz testing) tools to generate security tests [45, 72, 102, 116, 168]. Fuzzing injects invalid, malformed, or unexpected inputs into a system to reveal software defects and vulnerabilities [58]. Among current fuzzing tools, SAGE [116] is a white-box fuzzer, which starts with a well-formed input, symbolically executes the program-under-test, and gathers constraints on inputs from conditional branches encountered along the execution. The collected constraints are then systematically negated and solved with a constraint solver, whose solutions are mapped to new inputs that exercise different program execution paths. This process is repeated to sweep many feasible execution paths of the program while checking properties simultaneously. AFL [45] employs compile-time instrumentation and genetic algorithms to discover test cases that trigger new paths. OSS-Fuzz [72] provides a massively distributed fuzzing environment for widely-used open-source projects.

However, fuzzing also suffers from challenges. It is unable to explore deep paths; an inefficient initial seed can incur high runtime overheads because the mutations-to-generate depend on that seed. To overcome the limitations of both program verification and fuzzing, Alshmrany et al. [91] and Metta et al. [141] separately combined BMC with fuzzers to generate security tests. Our research is different from all the work mentioned above in two aspects. First, it

adopts ChatGPT to generate test cases. Second, it mimics the exemplar test for a vulnerable library, to similarly generate a security test for client Apps built on top of that library.

2.6 ChatGPT-Based Research

Because ChatGPT-4.0 was launched very recently, only a few studies were recently done to assess its capability in programming or assisting programmers [122, 145, 166, 169]. For instance, Nascimento et al. [145] chose four LeetCode questions to create prompts for ChatGPT-4.0. Jalil et al. [122] checked how well ChatGPT-4.0 performs when answering the common questions in a popular software testing curriculum. Sobania et al. [166] evaluated ChatGPT on the standard bug fixing benchmark set—QuixBugs; they found it to fix 31 out of 40 bugs, outperforming the state-of-the-art. Tian et al. [169] assessed ChatGPT-4.0's capability in code generation, program repair, and code summarization. Nikolaidis et al. [148] evaluated ChatGPT-4.0 and Copilot using LeetCode problems. Chen et al. [98] created GPTutor, a ChatGPT-4.0-powered programming tool, to provide code explanation for developers in the Visual Studio IDE. Our study complements all research work mentioned above, because we applied ChatGPT-4.0 to perform a totally different task: exploit generation.

Chapter 3

Empirical Study of Automatic Detection Tools on Security API Misuses[1]

Tools were recently built to scan Java applications to detect *the specialized category of security vulnerabilities—misuses of cryptographic APIs* [103, 107, 108, 131, 155, 158]. However, it is unclear what are the strengths and weaknesses of these tools, how well they help developers improve existing secure coding practices, and how we can design better approaches. Therefore, for this Chapter, we conducted a novel empirical study to explore (1) how current tools are designed to detect cryptographic API misuses, (2) how effectively the tools work to locate API misuses, and (3) how developers perceive the usefulness of tools' outputs.

Specifically, there are four steps in our study method. First, we searched in the ACM digital library for state-of-the-art tools, which analyze Java-based applications for any API misuses relevant to JCA and JSSE. Second, we compared the approach design of tools in terms of API usage pattern representations, pattern-matching logic, input/output infos, and public availability. Third, among all explored tools, we identified six publicly available, executable, and comparable tools: CogniCrypt [127], CryptoGuard [158], CryptoTutor [165], FindSecBugs [30], SonarQube [36], and Xanitizer [41]. To empirically compare tools' effec-

[1]In this chapter, API Misuses and API Insecure Usage are interchangeable.

tiveness, we applied the six tools to three public program benchmarks: CryptoBench [8], MUBench [93], and OWASP Benchmark [34]. Based on the ground truth of cryptographic API misuses and manual validation, we evaluated tools' precision, recall, and F-score rates. Fourth, to assess the relevance of tool outputs, we also applied the 6 tools to another dataset of 200 Apache projects, and filed 57 pull requests (PRs) to seek for developers' feedback. After sending our descriptions to the owners of 35 projects, we received 47 responses and analyzed the information to learn about developers' opinions. Our research explores the following research questions (RQs):

RQ1: *How are current tools designed to detect cryptographic API misuses?* We found that current tools represent misuse patterns as either hardcoded rules in tool implementation, Java code snippets, or templates described with domain-specific languages (DSL). These tools match Java programs against known misuse patterns via static program analysis, clone detection, or machine learning to reveal API misuses. Among different design options, most tools adopt hardcoded rules and inter-procedural static analysis probably because such a design can effectively locate misuses.

RQ2: *How effectively do current tools work to locate cryptographic API misuses?* The six experimented tools focus on slightly different pattern sets and achieved distinct trade-offs between precision and recall. Specifically, CryptoGuard outperformed other tools on CryptoBench, getting 85% F-score; Xanitizer acquired the highest F-score when being applied to OWASP Benchmark and MUBench (i.e., 100% and 72%). There is no tool universally better than the others.

RQ3: *How do developers perceive the usefulness of tools' outputs?* According to the 47 responses we received, most developers (i.e., 30) rejected the reported vulnerabilities, fewer developers (i.e., 17) wanted to address the reported issues, and even fewer developers (i.e., 9) replaced API misuses by following tool-generated guidance. The tools' reports usually did not

change developers' coding practices. We identified three factors that prevent developers from addressing reported issues. First, the fixing suggestions are vague and incomplete. Second, developers need evidence of security exploits enabled by those vulnerabilities. Third, some detected misuses were in test code or security-irrelevant program contexts, and developers believed those issues to cause no security consequence.

In the following sections, we will first introduce the misuse patterns of cryptographic APIs (Section 3.1). Then we will present our literature survey, which describes the existing API-misuse detectors to answer RQ1 (Section 3.2). Next, we will describe our empirical evaluation of six state-of-the-art tools to explore RQ2 (Section 3.3). Finally, we will explain our study with developers to investigate RQ3 (Section 3.4).

3.1 Misuse Patterns of Cryptographic APIs

The Java platform provides two important frameworks to enable security implementation: JCA and JSSE. JCA provides APIs to implement concepts of cryptography such as digital signatures, message digests, certificates and their validation, encryption, key generation and management, and secure random number generation [123]. JSSE enables secure internet communications; it includes APIs for creation of secure channels, data encryption, server authentication, message integrity, and optional client authentication [12].

Among all APIs defined in JCA and JSSE, there are 13 Java types (i.e., classes or interfaces) frequently mentioned in the API-misuse patterns summarized by prior research [103, 104, 107, 108, 118, 128, 158]. As shown in Table 3.1, 10 Java types are from JCA and the other 3 types are from JSSE. Each of these Java types has one or more method APIs that are prone to misuse, each API may be misused in one or multiple ways, and each API misuse pattern is considered a code vulnerability. To succinctly represent all API misuse patterns in Table 3.1,

Table 3.1: The insecure and secure usage patterns of method APIs related to 13 Java classes/interfaces

Java Type API	**Framework**	**Insecure**	**Secure**	**CWE Category**
`Cipher`	JCA	The passed name of a requested transformation is RC2, RC4, RC5, DES, DESede, 3DES, AES/ECB, RSA with NoPadding or Blowfish.	The parameter value is AES/GCM, AES/CCM, AES/CFB, AEC/CBC, or {RSA, RSA/ECB, RSA/None} with OAEP padding.	CWE-327: Use of a Broken or Risky Cryptographic Algorithm
`HostnameVerifier`	JSSE	Allow all hostnames.	Disallow the hostnames that do not pass validation.	CWE-295: Improper Certificate Validation
`IvParameterSpec`	JCA	Create an initialization vector (IV) with a constant.	Create an IV with an unpredictable random value.	CWE-330: Use of Insufficiently Random Values
`KeyPairGenerator`	JCA	Create an RSA key pair whose key size < 2048 bits, or create an ECC key pair whose key size < 224 bits.	RSA key size >= 2048 bits, or ECC key size >= 224 bits.	CWE-326: Inadequate Encryption Strength
`KeyStore`	JCA	When loading a keystore from a given input stream, the provided password is a hardcoded constant non-null value.	The password is retrieved from some external source (e.g., database or file)	CWE-798: Use of Hard-coded Credentials
`MessageDigest`	JCA	The used hash algorithm is MD2, MD5, SHA-1, or SHA-224.	The parameter value is SHA-256, SHA-512 or SHA-3.	CWE-327
`PBEKeySpec`	JCA	Create a PBEKey based on a constant salt.	Set the salt to an unpredictable random value.	CWE-330
`PBEParameterSpec`	JCA	Create a parameter set for password-based encryption (PBE) by setting salt size < 64 bits, iteration count <1000, or a constant salt.	Set salt size >= 64 bits, iteration count >=1000. Set the salt to an unpredictable random value.	CWE-326, CWE-330
`SecretKeyFactory`	JCA	Create a secret key with the algorithm DES, DESede, ARCFOUR, or PBEWithMD5AndDES.	Create a secret key with AES or PBEWithHmacSHA256AndAES_128.	CWE-327
`SecretKeySpec`	JCA	Create a secret key with a hardcoded constant.	Create a secret key with a raw key (i.e., credential) retrieved from some external source or an unpredictable random value dynamically generated.	CWE-798
`SecureRandom`	JCA	Use `Random` to generate random values, set `SecureRandom` to use a constant seed, or invoke `SecureRandom.setSeed()` before calling any `nextXXX()` method (e.g., `nextInt()`).	Replace `Random` with `SecureRandom`. If `setSeed(...)` is called, use a parameter that is generated by `nextBytes()`.	CWE-330
`SSLContext`	JSSE	Use the protocol SSL, SSLv2.0, SSLv3.0, TLSv1.0, or TLSv1.1.	Use the protocol TLSv1.2 or TLSv1.3.	CWE-757: Selection of Less-Secure Algorithm During Negotiation
`TrustManager`	JSSE	Trust all clients and servers.	Check clients or servers.	CWE-295

we list all Java type APIs (the container classes/interfaces of methods) instead of the misused method APIs, and summarize the misuse patterns as well as related correct usage. In Table 3.1, column **Insecure** describes API misuse patterns, while **Secure** summarizes the correct usage patterns with security guarantees. The **Common Weakness Enumeration (CWE)** [23] is a category system for software weaknesses and vulnerabilities. To facilitate understanding of API misuse patterns, we map the patterns to six CWE categories to explain their security implications:

1. CWE-327: Use of a Broken or Risky Cryptographic Algorithm. When the methods APIs of three Java types (i.e., `Cipher.getInstance(...)`, `MessageDigest.getInstance(...)` and `SecretKeyFactory.getInstance(...)`) get invoked with improper parameters (e.g., `"DES"` and `"MD5"`), security experts consider those invocations to be vulnerable. This is because the parameter values indicate usage of broken or risky cryptographic algorithms, and the usage may result in the exposure of sensitive information [26]. For instance, `"DES"` on line 10 in Listing 3.1 implies using the symmetric-key algorithm DES (Data Encryption Standard), which is proven insecure [164]. Thus, the method invocation on line 10 is considered an instance of API misuse.

```
1  private static byte[] desKey = "12345678".getBytes();
2  private static byte[] iv = "12345678".getBytes();
3  public static void insecureEncrypt(String in) {
4    try {
5   // Declare an IV parameter with constant (CWE-330).
6      IvParameterSpec ivSpec = new IvParameterSpec (iv);
7   // Create a secret key with a hardcoded constant (CWE-798).
8      SecretKey key = new SecretKeySpec(desKey, "DES");
9   // Declare a DES cipher although DES is provenly insecure (CWE-327).
10     Cipher c=Cipher.getInstance("DES/CBC/PKCS5Padding");
11     c.init(Cipher.ENCRYPT_MODE, key, ivSpec);
12     ... } ... }
```

Listing 3.1: A code snippet that misuses three APIs

2. CWE-295: Improper Certificate Validation. When the method APIs `HostnameVerifier.verify(...)`, `TrustManager.checkClientTrusted(...)`, and `TrustManager.checkServerTrusted(...)` get overridden with (almost) empty code implementation, security experts consider those overridden methods to be vulnerable. This is because with naïve or empty code implementation, a software program does not validate, or incorrectly validates, hostnames and/or certificates; it may allow an attacker to spoof a trusted entity by interfering in the communication path between the host and client [24]. For instance, Listing 3.2 shows the empty bodies for `checkXXX(...)` methods of `TrustManager`. Such naïve method overriding actually voids the intended protection mechanism offered by JSSE.

```
1 private static TrustManager createTrustAllManager() {
2   return new X509TrustManager() {
3 // Override checkClientTrusted (...) to have empty body (CWE-295).
4     @Override
5     public void checkClientTrusted(...) throws CertificateException {}
6  // Override checkServerTrusted (...) to have empty body (CWE-295).
7     @Override
8     public void checkServerTrusted(...) throws CertificateException {}
9   ... };}
```

Listing 3.2: Insecure method overriding for `TrustAllManager`

3. CWE-330: Use of Insufficiently Random Values. When some methods of four Java types (i.e., `IvParameterSpec`, `PBEKeySpec`, `PBEParameterSpec`, and `SecureRandom`) get called with constants or predictable random values, the method calls are considered insecure. This is because when software generates predictable values in a context requiring unpredictability, it may be possible for an attacker to guess the next value that will be generated, and use this guess to impersonate another user or access sensitive information [27]. Line 6 of Listing 3.1 presents an exemplar API misuse, which creates an `IvParameterSpec` object with a constant array derived from the string literal `"12345678"`.

4. CWE-326: Inadequate Encryption Strength. When certain methods of two Java types (i.e., `KeyPairGenerator` and `PBEParameterSpec`) are invoked, if the parameter values are constants within specific value ranges, the invocations are considered vulnerable. This is because when generating keys or creating parameters used for password-based encryption (PBE), if a program specifies relatively low numbers for key lengths, salt sizes, or iteration counts, the leveraged encryption scheme is theoretically sound but not strong enough for the level of protection required. The weak encryption scheme can be subjected to brute force attacks that have a reasonable chance of succeeding using current attack methods and resources [25]. Listing 3.3 shows an exemplar misuse of `KeyPairGenerator` APIs, which initializes a generator to create RSA key pairs with the 1024-bit key size; however, the key size should be no smaller than 2048.

```
1 KeyPairGenerator gen=KeyPairGenerator.getInstance("RSA");
2 // The RSA key size should be at least 2048 (CWE-326).
3 gen.initialize(1024);
4 KeyPair kp = gen.generateKeyPair();
```

Listing 3.3: An exemplar misuse of the `KeyPairGenerator` APIs

5. CWE-798: Use of Hardcoded Credentials. When a program calls `KeyStore.load(...)` or `SecretKeySpec(...)` with a hardcoded constant as the credential parameter, the method invocation is treated vulnerable. The reason is that hardcoded credentials typically create a significant hole that allows an attacker to bypass the authentication that has been configured by the software administrator [29]. Line 8 in Listing 3.1 shows an exemplar API misuse of this category. In the example, `SecretKeySpec(...)` is invoked with `desKey`, which credential comes from a hardcoded constant `"12345678"`.

6. CWE-757: Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade'). When a program calls `SSLContext.getInstance(...)` with any of the following

parameters: `"SSL"`, `"SSLv2"`, `"SSLv3"`, `"TLSv1.0"`, and `"TLSv1.1"`, the invocation is treated insecure. Secure Socket Layer (SSL) and TLS (Transport Layer Security) are standard protocols for keeping an internet connection secure and safeguarding the transmitted data [149]. As a successor of SSL, TLS is more secure. Security exerts recommend to enforce TLS 1.2 as the minimum protocol version and to disallow older versions like TLS 1.0. Failure to do so could open the door to downgrade attacks: a malicious actor who is able to intercept the connection could modify the requested protocol version and downgrade it to a less secure version [28, 39].

Summary: We defined Table 3.1 to include only the API misuse patterns *frequently* mentioned by literature for three reasons. First, different literatures sometimes define conflicting patterns, so focusing on the most common ones can avoid arguable cases. Second, these patterns enable us to empirically compare the effectiveness of different tools on the same benchmarks. Third, these patterns are representative, so our study based on them can reflect developers' general perception of security-API misuses.

3.2 A Literature Survey of Current Tools

Tools were built to automatically detect Java cryptographic API misuses. These tools typically start with certain representations of misuse patterns, adopt different techniques to scan programs for pattern matches, and generate reports when matches are found. Table 3.2 shows an overview of existing detectors. This list is complete to the best of our knowledge. We searched for literatures published in 2020, with keywords "secure API misuse Java" in the ACM digital library. We then read the retrieved papers together with their references to identify all relevant tools. For each tool, this table summarizes the pattern representation, pattern-matching strategy, and output; it also characterizes two properties: availability and

Table 3.2: Overview of existing detectors for security API misuses

Name	**Availability**	**Input Format**			**Pattern Representation**		**Pattern-Matching Strategy**			**Output**	
		Java	JAR	APK	Built-in Rules	Other	Intra-	Inter-	Other	Misuse	Repair
MalloDroid [104]	✓			✓	✓		✓			✓	
CryptoLint [103]				✓	✓			✓		✓	
BinSight [143]				✓	✓			✓		✓	
CDRep [132]				✓	✓			✓		✓	✓
CryptoTutor [165]	✓	✓			✓			✓		✓	✓
CMA [162]				✓	✓			✓	✓	✓	
CryptoChecker [151]		✓			✓		-	-	-	✓	
Amandroid [172]	✓			✓	✓			✓		✓	
CogniCrypt [127]	✓	✓	✓	✓		✓		✓		✓	
Hotfixer [146]		✓	✓	✓		✓		✓		✓	✓
Fischer et al.'s tool [107]		✓		✓		✓	✓		✓	✓	
Xu et al.'s tool [176]				✓		✓		✓	✓	✓	
CryptoGuard [158]	✓		✓	✓	✓			✓		✓	
VuRLE [133]		✓				✓	✓			✓	✓
Vulvet [111]				✓	✓			✓		✓	✓
AndroBugs [19]	✓			✓	✓		✓			✓	
FindSecBugs [30]	✓		✓		✓			✓		✓	
MobSF [33]	✓	✓		✓	✓	✓			✓	✓	
SonarQube [36]	✓	✓			✓			✓		✓	
Xanitizer [41]	✓	✓	✓		✓			✓		✓	

"-" means the information is not publicly available.

the input format. **Availability** describes whether a tool is publicly available or just has its methodology described in literature. **Input format** reflects whether a tool can analyze Java source code, JAR files, or APK files. Among the 20 tools studied, there are 15 research prototypes from academia, and 5 tools from industry or open source communities.

3.2.1 Research Prototypes from Academia

Most tools that fall into this category scan the APK files of Android apps. Among the 15 tools, 10 tools hardcode patterns as built-in rules probably due to the simplicity of such representations; another 5 tools represent patterns with code snippets or templates written in a domain-specific language. 11 tools conduct inter-procedural static analysis for pattern matching, perhaps due to the relatively higher precision of such analysis than intra-procedural analysis and other techniques. Five tools suggest customized repairs.

MalloDroid [104] scans the decompiled code of Android apps to detect potential vulnerabilities related to SSL. It uses intra-procedural static analysis to i) extract networking API calls and valid HTTP(S) URLs, ii) check the validity of SSL certificates for all extracted HTTPS hosts, and iii) identify apps that validate certificates inadequately (CWE-295).

CryptoLint [103] is similar to MalloDroid, because it also extends Androguard [20]—a tool to decompile APK files into Dalvik bytecode and to statically analyze the bytecode. However, CryptoLint hardcodes six rules:

1. Do not use ECB mode for encryption (CWE-327).
2. Do not use a non-random IV for CBC encryption (CWE-330).
3. Do not use constant encryption keys (CWE-798).
4. Do not use constant salts for PBE (CWE-330).

5. Do not use fewer than 1,000 iterations for PBE (CWE-326).

6. Do not use static seeds to seed `SecureRandom(...)` (CWE-330).

For each located potentially vulnerable API call (e.g., `Cipher.getInstance(v)`), CryptoLint conducts inter-procedural backward slicing to decide whether the used parameter value is insecure (e.g., `v="AES/ECB"`). Although its design has been followed by later tools, CryptoLint is not publicly available.

BinSight [143] reimplements CryptoLint. However, its design seems better as it maps Java classifiers to their container software (i.e., an Android app or a third-party library) in a semi-automated way. When Android apps are obfuscated and identifiers are renamed, it is challenging to correctly map detected vulnerabilities to the original code or software libraries. To overcome this challenge, BinSight implements several heuristics to automate identifier mapping.

CDRep [132] automates both the detection and repair of security API misuses for Android apps. CDRep reimplements the design of CryptoLint for vulnerability detection. To repair vulnerabilities, CDRep leverages seven manually created patch templates, with each template usable to fix one misuse pattern.

CryptoTutor [165] helps students locate and repair cryptographic API misuses in Java code. Similar to CDRep, CryptoTutor applies program slicing and inter-procedural data flow analysis to locate misuses. Its built-in rules are also related to the vulnerabilities of CWE-327, CWE-330, CWE-798, and CWE-326. However, CryptoTutor focuses on a larger pattern set; in addition to the misuses examined by CryptoLint, CryptoTutor also checks API misuses that

1. use weak hash functions (e.g., MD5),

2. use weak encryption algorithms (e.g., DES),

3. use weak random number generators (e.g., `Random(...)`, and

4. use short-length keys or salts for encryption.

CryptoTutor repairs vulnerabilities by editing abstract syntax trees (ASTs) for code transformation. Once API misuses are located in the code submitted by a student, CryptoTutor provides coding feedback to help the student understand why the program is incorrect.

Crypto Misuse Analyzer (CMA) [162] scans Dalvik bytecode of Android apps and checks for API misuses related to CWE-327, CWE-295, CWE-330, CWE-326, and CWE-798. CMA first uses inter-procedural static analysis to identify all execution paths that may invoke certain cryptographic APIs. Based on the analysis result, CMA instruments code to perform dynamic analysis, log execution profiles, and record how cryptographic APIs are invoked at runtime. Finally, CMA matches execution profiles with predefined API-misuse models, to decide whether any API is misused.

CryptoChecker [151] also detects cryptographic API misuses based on built-in rules. Before hardcoding rules into CryptoChecker, Paletov et al. first built a rule inference tool called DiffCode. There are three steps in DiffCode. First, DiffCode mines code changes from GitHub repositories based on their usage of particular crypto APIs (e.g., `SecertKeySpec`). Second, to filter out irrelevant changes from the mined corpus, DiffCode represents invoked APIs and related parameter values with directed acyclic graphs (DAGs). By comparing DAGs, DiffCode extracts API usage changes and then clusters similar changes to infer API misuse patterns. The mined rules are related to CWE-327, CWE-330, CWE-326, and CWE-798. However, CryptoChecker's pattern set is much smaller than that of CryptoTutor. It is unclear what technique CryptoChecker adopts to match patterns.

Amandroid [40, 172] is a general-purpose static analysis framework, to decide points-to

information for all objects in a flow- and context- sensitive way across Android app components. This technique seems more accurate than prior work. As the researchers noted, the event-driven nature and inter-component communication (ICC) of Android apps make traditional analysis insufficient and imprecise, and require additional processing to connect the control flow graphs of some seemingly irrelevant functions. Wei et al. [172] demonstrated that Amandroid can be easily extended to find API misuses that adopt ECB mode for encryption (CWE-327).

CogniCrypt [127] supports developers to properly use APIs in two ways. First, for some common tasks (e.g., data encryption), CogniCrypt generates code from high-level task descriptions in English. Second, CogniCrypt takes in rules defined in a domain-specific language (DSL)—CrySL [128]—to detect API misuses. Each CrySL rule has five mandatory sections:

(1) OBJECTS declares Java objects;

(2) EVENTS lists all security APIs involved;

(3) ORDER uses a regular expression to define correct API call sequences;

(4) CONSTRAINTS defines constraints on objects; and

(5) ENSURES defines predicates on the relationship among objects.

Because CogniCrypt translates CrySL rules into context-sensitive, flow-sensitive, and demand-driven static analysis, users can extend the tool capability by defining new rules. For detected API misuses, CogniCrypt offers fixing guidance (e.g., replacing an insecure parameter value with a secure one). Its pattern set is related to CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757.

Hotfixer [146] adopts CogniCrypt to detect API misuses, and applies fixes at runtime without stopping the program execution. To dynamically update software, Hotfixer transforms handcrafted software patches into hotfixes that are usable by Java agents, and checks the program execution status before applying any patch. For instance, if a method is running and a redefinition of that method is suggested as a security patch, then the method's old implementation will continue running until the execution finishes. Hotfixer ensures that only future runs of that method will execute the new implementation.

Fischer et al.'s tool [107] detects misuses in two ways: machine learning and clone detection. Specifically, their first approach uses *tf-idf* to generate features from source code. It trains a support vector machine (SVM) with an annotated dataset of code snippets that use APIs securely or insecurely. The trained model predicts whether a code snippet misuses any security API. Their second approach converts both known vulnerable Java code and Android apps to the same representation, i.e., the internal representation (IR) of WALA [16]—a widely used program analysis framework. The approach scans Android apps for clones (i.e., similar code) of the known vulnerable code, by finding isomorphic subgraphs in IR-based program dependency graphs (PDG). Both tools focus on misuse patterns related to CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757.

Xu et al.'s tool [176] is similar to that of Fischer et al., as it also detects misuses via machine learning. The approach first analyzes the Dalvik bytecode of APK files to extract all possible API invocation sequences from Android apps, and uses CogniCrypt to label secure and insecure call sequences. With the labeled dataset, the tool trains (1) a Hidden Markov Model (HMM) to predict how likely a given API sequence is secure, and (2) an n-gram model to further locate the misused API(s) in a problematic call sequence. As this tool adopts CogniCrypt to label training data, the pattern set it learns overlaps with, but can be no larger than that of CogniCrypt.

CryptoGuard [9, 158] extends Soot [15, 171]—a widely used program analysis framework—to statically analyze Java bytecode. It focuses on vulnerabilities of CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757. To achieve high precision rates when detecting vulnerabilities, CryptoGuard conducts both backward and forward slicing in a context- and field- sensitive way. However, because eight of CryptoGuard's rules are about the usage of constant values, naïvely applying existing slicing techniques can falsely report constants that are covered by program slices but totally irrelevant to security. Therefore, CryptoGuard defines refinement algorithms to remove false alarms based on the domain knowledge of cryptography.

VuRLE [133] detects and fixes vulnerabilities. There are two phases in VuRLE: learning and repair. In Phase I, given vulnerable programs and corresponding repaired code, VuRLE first extracts edits by comparing the ASTs of each ⟨vulnerable, repaired⟩ code pair; it represents each edit as a sequence of AST edit operations (e.g., node insertion). Next, VuRLE clusters similar edits based on the longest common subsequences (LCSs) between edit operations; for each cluster, VuRLE generalizes a ⟨template, edit⟩ pair. Here, the template abstractly represents a vulnerable code pattern, while the edit pattern represents the repair. In Phase II, given a vulnerable program, VuRLE scans code for matches of any inferred template. For each template match, VuRLE customizes the corresponding edit pattern, and applies the customized changes to repair vulnerabilities. Its pattern set is about CWE-327, CWE-295, CWE-798, and some categories outside our research scope (e.g., resource leakage).

Vulvet [111] extends Soot to statically analyze the Dalvik bytecode of Android apps; it detects and fixes cryptographic API misuses as well as other types of vulnerabilities (e.g., ICC-related). The patterns Vulvet focuses on are related to CWE-327, CWE-295, CWE-330, and CWE-798. Vulvet automatically resolves vulnerabilities by instrumenting patches to the Jimple code of Android apps, where Jimple is an internal representation of Soot.

AndroBugs [19] is an open-source framework to scan Android apps for vulnerabilities. Among all the vulnerability categories AndroBugs considers, two categories are within our research scope: CWE-295 and CWE-798. AndroBugs implements a naïve string-match method, to detect API misuses that match certain regular expressions (regex).

FindSecBugs [30] is the SpotBugs [167] plugin for security audits of Java web applications. Here, SpotBugs is an open-source tool that statically analyzes Java bytecode for software bugs. FindSecBugs performs inter-procedural static program analysis to find API misuses related to CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757. The misuse patterns in FindSecBugs are hardcoded as built-in rules; however, the software architecture provides extensible interfaces for developers to easily add or remove rules. For each detected vulnerability, FindSecBugs can provide general guidance on repairs by showing code examples; it does not suggest concrete repairs applicable to any specific program context.

MobSF [33] is an open-source, automated, all-in-one mobile application (Android/iOS/Windows) pen-testing, manual analysis, and security assessment framework. It performs both static and dynamic analysis to detect a variety of vulnerabilities. Among the categories listed in Table 3.1, MobSF locates API misuses related to CWE-327, CWE-295, CWE-330, and CWE-798. To identify improper certificate validation (CWE-295), MobSF hardcodes built-in rules and performs dynamic program analysis. To reveal other API misuses, MobSF holds an independent YAML file to represent misuse patterns with regular expressions (regex), and conducts regex-based string match.

SonarQube [36] is an open-source tool that conducts inter-procedural static analysis to detect bugs, code smells, and security vulnerabilities. SonarQube hardcodes built-in rules for vulnerabilities of CWE-327, CWE-295, CWE-330, CWE-326, and CWE-757. For each detected vulnerability, SonarQube presents general guidance on repairs by showing (1) secure code examples and (2) relevant CWE entries.

Xanitizer [41] is a closed-source commercial tool for the security audit of Java web applications. We were able to use Xanitizer by requesting for the user license. According to our experience with the tool, Xanitizer conducts inter-procedural static analysis to reveal API misuses of CWE-327, CWE-295, CWE-330, CWE-326, CWE-798, and CWE-757. It scans not only Java code and JAR files, but also configuration files and templates for rendering the HTML output. For each detected vulnerability, Xanitizer offers a high-level repair suggestion (e.g., "specify crypto-provider") together with relevant CWE entries.

Finding 1: *For RQ1, existing tools are different in terms of their availability, input formats, pattern representations, pattern-matching strategies, and outputs. Most tools represent patterns as built-in rules, conduct inter-procedural analysis, and report detected API misuses as outputs.*

3.3 Empirical Comparison of Tools

To empirically compare the tools listed in Table 3.2, we first tried to download all tools and deploy them to our desktop, and then applied the successfully deployed ones to existing datasets. The configuration of our desktop includes (1) OS: Linux Mint 20, (2) CPU: i7-8700, (3) memory size: 32 GB, and (4) JVM heap size: 30 GB. In this section, we will first introduce the experimented tools (Section 3.3.1) and evaluation datasets (Section 3.3.2). Then we will explain our evaluation metrics (Section 3.3.3) and results (Section 3.3.4).

3.3.1 Tools Used in Experiments

Within the tools listed in Table 3.2, nine tools are unavailable and do not support any free trial. Although BinSight is open-source, we could not compile or run it, neither did

Table 3.3: The API misuse patterns covered by each tool

Java Type API	CogniCrypt	CryptoGuard	CryptoTutor	FindSecBugs	SonarQube	Xanitizer
`Cipher`	✓	✓	✓	✓	✓	✓
`HostnameVerifier`		✓		✓	✓	✓
`IvParameterSepc`	✓	✓	✓	✓	✓	✓
`KeyPairGenerator`	✓	✓		✓	✓	✓
`KeyStore`	✓	✓		✓		✓
`MessageDigest`	✓	✓	✓	✓	✓	✓
`PBEKeySpec`	✓	✓		✓	✓	✓
`PBEParameterSpec`	✓	✓	✓		✓	✓
`SecrectKeyFactory`	✓		✓			✓
`SecrectKeySpec`	✓	✓	✓	✓		✓
`SecureRandom`	✓	✓	✓	✓	✓	✓
`SSLContext`	✓	✓		✓	✓	✓
`TrustManager`	✓	✓	✓	✓	✓	✓

"✓" means a Java class/interface has at least one method-API misuse pattern covered by a tool

the authors respond to our email requests. Thus, we still consider BinSight unavailable. Among the remaining 10 tools, MalloDroid, Amandroid, AndroBugs, and MobSF are only applicable to Android apps. Because our evaluation datasets include only six Android apps (see Section 3.3.2), which are insufficient to evaluate any tool, we decided not to experiment with these four tools. Finally, we have six tools usable in the empirical comparison of tools' detection capability: CogniCrypt, CryptoGuard, CryptoTutor, FindSecBugs, SonarQube, and Xanitizer. The first three tools are from academia, while the last three come from industry. Table 3.4 shows the tool versions we adopted. As shown in Table 3.3, among the six tools, Xanitizer covers the most misuse patterns while CryptoTutor covers the fewest.

Table 3.4: The tool versions adopted

Tool	**Version or Commit Id on Github**
CogniCrypt	2.7.1
CryptoGuard	Release_04.05.03_2020-11-25-02-42
CryptoTutor	v202107
FindSecBugs	1.10.1
SonarQube	8.5.1.38104
Xanitizer	5.1.3

3.3.2 Benchmark Datasets

To evaluate the effectiveness of tools, we searched extensively online for third-party benchmarks that label programs based on their correct or incorrect usage of security APIs. We found three datasets:

(1) CryptoBench [8, 88] includes 171 handcrafted programs that use the APIs of JCA and JSSE. In particular, 136 of the programs have cryptographic API misuses, while the other 35 programs use APIs correctly. To precisely comprehend API usage, a tool needs to do intra-procedural analysis for 40 programs, and perform inter-procedural analysis for the other 131 programs. Among the 136 vulnerable programs, only 129 programs contain the

API misuses we focus on (see Table 3.1).

(2) MUBench [18, 93] is a benchmark of API-misuse detectors. Among the different versions of MUBench, we downloaded a recent version created in 2019 [6], which contains instances of cryptographic API misuses collected from 62 Java programs. These programs include 6 Android apps and 56 non-Android applications. We managed to compile 37 out of the 56 Java applications into JAR files, which correspond to 149 labeled instances of API misuses. Therefore, we used these 149 instances as ground truth in our evaluation.

(3) OWASP Benchmark [34, 35] is a Java test suite designed to evaluate the effectiveness of automated vulnerability detection. It gathers the vulnerabilities recently reported on CWE [23], and has been recommended as an evaluation dataset for Application Security Testing tools. We downloaded the latest version (v1.2) of this benchmark, which includes 2,740 Java programs. Because not all programs involve security APIs, we focused on the data of three categories: weak cryptography, weak hashing, and weak randomness. In this way, our experiment includes 975 programs from the original dataset, containing 477 programs with labeled misuses of security APIs and 498 programs with correct uses.

Actually not every benchmark covers all the API misuses summarized by prior work. To ensure rigorous evaluation of tools, we manually inspected the security APIs labeled in each benchmark. We present the mapping between benchmarks and security APIs in Table 3.5. As shown in the table, CryptoBench covers the usage of most APIs (i.e., 11 Java types). The data of OWASP Benchmarks is only relevant to three Java classes: `Cipher`, `MessageDigest`, and `SecureRandom`. We chose to use existing benchmark datasets instead of creating new ones for two reasons. First, these benchmarks are public and were manually crafted by different groups of people, which makes our empirical comparison representative, and easy to reproduce by other people. Second, some of the benchmarks (e.g., OWASP Benchmark and MUBench) are widely accepted and have great industrial impacts, which enables our

Table 3.5: The security APIs covered by each benchmark

Java Type API	CryptoBench	MUBench	OWASP Benchmark
`Cipher`	✓	✓	✓
`HostnameVerifier`	✓		
`IvParameterSepc`	✓	✓	
`KeyPairGenerator`	✓		
`KeyStore`	✓		
`MessageDigest`	✓	✓	✓
`PBEKeySpec`	✓		
`PBEParameterSpec`	✓	✓	
`SecrectKeyFactory`		✓	
`SecrectKeySpec`	✓	✓	
`SecureRandom`	✓		✓
`SSLContext`			
`TrustManager`	✓		

"✓" means that a Java class/interface has at least one method API called by a program benchmark.

empirical results to better characterize the state-of-the-art tools and inspire future research.

3.3.3 Evaluation Metrics

We used four metrics to measure tool effectiveness: precision, recall, F-score, and runtime overhead.

Precision (P) measures among all reported misuses, how many of them are actual misuses (i.e., true positives):

$$P = \frac{\text{\# of true misuses detected}}{\text{Total \# of detected misuses}}. \tag{3.1}$$

Given a reported set of misuses S_1, suppose that the labeled set of misuses (i.e., ground truth) is S_2. Theoretically, we can automatically evaluate precision based on the intersection between two sets, i.e., $P = |S_1 \cap S_2|/|S_1|$. Such automatic evaluation requires the ground truth (i.e., S_2) to be complete. Namely, the labeled set of misuses in each benchmark should cover all actual misuses existing in codebases.

Table 3.6: The precision, recall, and F-score of tools measured based on CryptoBench (%)

Java Type API	CogniCrypt			CryptoGuard			CryptoTutor			FindSecBugs			SonarQube			Xanitizer		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
`Cipher(36)`	85	78	81	85	97	91	67	22	33	62	36	46	43	17	24	83	83	83
`HostnameVerifier(1)`	-	0	-	-	0	-	-	0	-	100	100	100	50	100	67	100	100	100
`IvParameterSepc(8)`	71	63	67	88	88	88	0	0	-	80	100	89	-	0	-	89	100	94
`KeyPairGenerator(5)`	83	100	91	80	80	80	-	0	-	-	0	-	-	0	-	83	100	91
`KeyStore(7)`	75	86	80	88	100	93	-	0	-	100	29	44	-	0	-	100	29	44
`MessageDigest(24)`	94	67	78	86	100	92	67	26	36	67	33	44	80	83	82	83	83	83
`PBEKeySpec(8)`	63	63	63	86	75	80	-	0	-	100	25	40	100	50	67	73	100	84
`PBEParameterSpec(14)`	77	71	74	85	79	81	67	14	24	-	0	-	-	0	-	-	0	-
`SecrectKeySpec(8)`	78	88	82	100	38	55	50	13	20	67	25	36	-	0	-	50	13	20
`SecureRandom(15)`	100	7	13	86	80	83	50	13	21	100	7	13	100	60	75	-	0	-
`TrustManager(3)`	-	0	-	-	0	-	-	0	-	100	100	100	100	100	100	100	100	100
Overall (129)	**81**	**64**	**72**	**86**	**84**	**85**	**61**	**15**	**24**	**73**	**31**	**43**	**75**	**33**	**46**	**83**	**60**	**70**

"-" means that a tool does not report any misuse for certain Java type APIs, so the related precision and F score values cannot be calculated.

Because CryptoBench and OWASP are manually crafted datasets with injected API misuses, their ground truth sets are complete. However, MUBench consists of software from the real world, and the labeled set was crafted based on manual inspection or tool results. The ground truth of MUBench is incomplete and thus unusable for automatic evaluation. To correctly compute precision of tools on MUBench, we manually inspected all reported misuses and decided whether they were true positives based on our security knowledge. Namely, given a reported API misuse, if our manual inspection of the program context confirms the misuse, we consider the report to be a true positive; otherwise, it is a false positive.

Recall (R) measures among all known API misuses in benchmarks, how many of them are detected by a tool:

$$R = \frac{\text{\# of true misuses detected}}{\text{Total \# of known true misuses}}. \tag{3.2}$$

Given a reported set of misuses S_1, suppose that the labeled set of misuses is S_2. We evaluated recall using $|S_1 \cap S_2|/|S_2|$.

F-score (F) is the harmonic mean of P and R, to reflect a trade-off between the two metrics:

$$F = \frac{2 \times P \times R}{P + R}. \tag{3.3}$$

F varies within [0, 1]. The higher F scores are desirable, because they demonstrate better trade-offs between precision and recall. Suppose that we have 100 known API misuses in a codebase; a tool reports 120 misuse instances, with 80 of them being true misuses. Then $P = 80/120 = 67\%$, $R = 80/100 = 80\%$, $F = 2 \times 80\% \times 67\%/(80\% + 67\%) = 73\%$.

Runtime Overhead measures the time cost of each tool. The lower overhead, the better.

Table 3.7: Time cost comparison between tools (seconds)

Tool	CryptoBench	MUBench	OWASP
CogniCrypt	7	133	10,773
CryptoGuard	11	1,918	9,045
CryptoTutor	22	530	-*
FindSecBugs	4	59	20,352
SonarQube	28	387	2,188
Xanitizer	95	724	490,729

* CryptoTutor does does not run successfully with the OWASP benchmark.

Table 3.8: The precision, recall, and F-score of tools measured based on MUBench (%)

Java Type API	CogniCrypt			CryptoGuard			FindSecBugs			SonarQube			Xanitizer		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
`Cipher(50)`	92	66	77	100	68	81	100	66	80	100	76	86	86	78	82
`IvParameterSpec(13)`	88	62	72	100	31	47	69	46	55	-	0	-	65	46	54
`MessageDigest(31)`	89	58	70	100	81	89	77	68	72	86	55	67	85	77	81
`PBEParameterSpec(7)`	79	100	88	100	100	100	-	0	-	100	29	44	100	57	73
`SecretKeyFactory(9)`	84	100	91	-	0	-	-	0	-	-	0	-	82	100	90
`SecretKeySpec(39)`	48	62	54	100	8	14	100	3	5	-	0	-	100	28	44
Overall (149)	**77**	**66**	**71**	**100**	**49**	**66**	**84**	**41**	**55**	**93**	**38**	**54**	**85**	**62**	**72**

Table 3.9: The precision, recall, and F-score of tools measured based on OWASP Benchmark (%)

Java Type API	CogniCrypt			CryptoGuard			FindSecBugs			SonarQube			Xanitizer		
	P	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**	**P**	**R**	**F**
`Cipher(130)`	83	100	91	83	100	91	100	100	100	83	100	91	100	100	100
`MessageDigest(129)`	100	69	82	100	69	82	100	69	82	-	0	-	100	100	100
`SecureRandom(218)`	-	0	-	100	100	100	100	100	100	-	0	-	100	100	100
Overall (477)	**89**	**46**	**61**	**94**	**92**	**93**	**100**	**92**	**96**	**83**	**27**	**41**	**100**	**100**	**100**

3.3.4 Results Based on Benchmarks

Table 3.7 shows the runtime overheads of five tools (except CryptoTutor). When running tools other than CryptoTutor, we managed to launch tools via commands and apply each tool to every program benchmark, in order to measure the time cost. To test the project, we first import the project into the Eclipse with the plugin, select this project, then trigger the execution of CryptoTutor. However, CryptoTutor an interactive coding assistance tool that is delivered as an Eclipse plugin. To launch the tool, we have to first launch a new instance of Eclipse, import all programs into the IDE for each benchmark, and manually right-click all programs to trigger the execution of CryptoTutor. Such frequent manual interference makes it impossible to precisely measure the tool's time cost.

Among the five tools with runtime overheads captured, Xanitizer spent the most time when being applied to CryptoBench and OWASP (i.e., 95 and 490,729 seconds); CryptoGuard got the highest time cost when being applied to MUBench (i.e., 1,918 seconds). Two reasons can explain the observed time differences among tools. First, these tools adopt distinct static analysis techniques (e.g., inter-procedural vs. intra-procedural) to match patterns, and some techniques are more time-consuming than others. Table 3.7 shows the runtime overheads of five tools (except CryptoTutor). When running tools other than CryptoTutor, we managed to launch tools via commands and apply each tool to every program benchmark, in order to measure the time cost. However, CryptoTutor an interactive coding assistance tool that is delivered as an Eclipse plugin. To launch the tool, we have to first launch a new instance of Eclipse, import all programs into the IDE for each benchmark, and manually right-click all programs to trigger the execution of CryptoTutor. Such frequent manual interference makes it impossible to precisely measure the tool's time cost.

Among the five tools with runtime overheads captured, Xanitizer spent the most time when

being applied to CryptoBench and OWASP (i.e., 95 and 490,729 seconds); CryptoGuard got the highest time cost when being applied to MUBench (i.e., 1,918 seconds). Two reasons can explain the observed time differences among tools. First, these tools adopt distinct static analysis techniques (e.g., inter-procedural vs. intra-procedural) to match patterns, and some techniques are more time-consuming than others.

Table 3.7 shows the measured runtime overheads. Within the six tools, Xanitizer spent the most time when being applied to CryptoBench and OWASP (i.e., 95 and 490,729 seconds); CryptoGuard got the highest time cost when being applied to MUBench (i.e., 1,918 seconds). Two reasons can explain the observed time differences among tools. First, these tools adopt distinct static analysis techniques (e.g., inter-procedural vs. intra-procedural) to match patterns, and some techniques are more time-consuming than others.

Second, tools focus on different vulnerability patterns, although some patterns are irrelevant to cryptographic APIs. Since we were unable to revise tools to disable all patterns irrelevant to our investigation, the measured costs are higher than the actual time costs incurred by automatic detection of cryptographic APIs.

Finding 2 (for RQ2): *The measured time costs imply that given hundreds of programs to scan, the experimented tools usually respond within six hours (18,000 seconds).*

In terms of detection capability, CryptoGuard achieved the highest F score (85%) among all tools when being applied to CryptoBench. Meanwhile, Xanitizer acquired the highest F scores (i.e., 72% and 100%) when being applied to MUBench and OWASP datasets. In the following subsections, we will further discuss the precision, recall, and F-score of tools on each dataset.

CryptoBench Table 3.6 shows the evaluation results of different tools on CryptoBench.

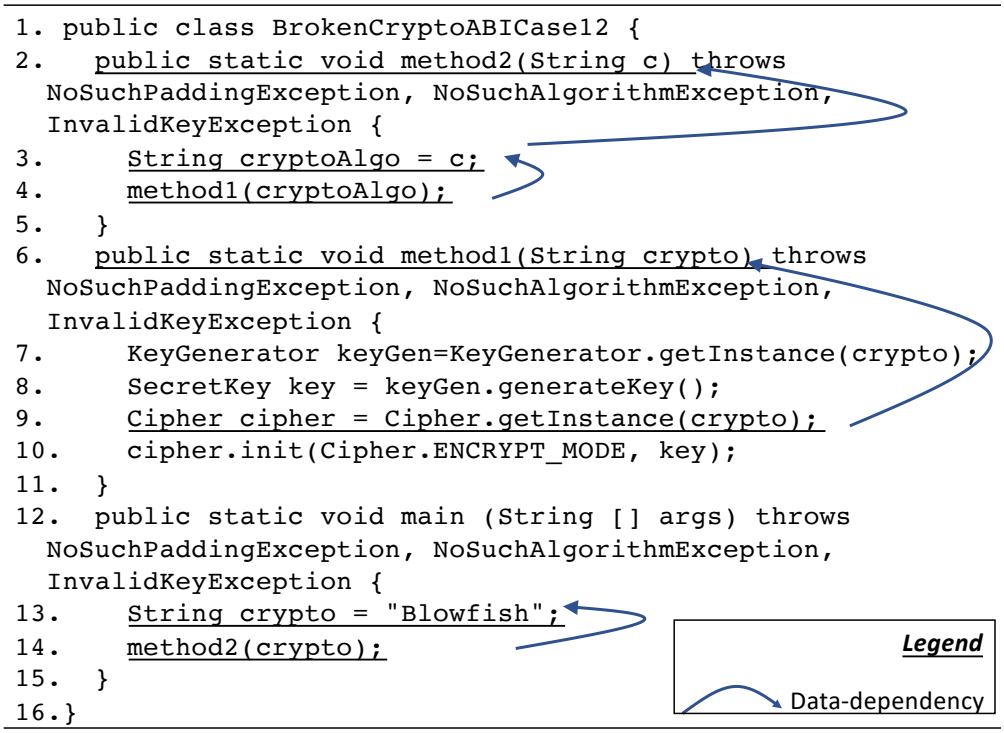

Figure 3.1: A program in which SonarQube and FindSecBugs could not find the API misuse

Each row in the table corresponds to API misuses related to one Java type (e.g., `Cipher`); each number mentioned in the first column (e.g., 36) counts the labeled cryptographic API misuse instances for a Java type. Among the 11 Java types covered by CryptoBench, CryptoTutor and SonarQube separately reported API misuses for 6 Java types; the other tools reported API misuses related to 9 Java types. There are only two Java types whose API misuses are commonly detected by all tools: `Cipher` and `MessageDigest`; CryptoGuard consistently outperformed the others when handling these common cases. Our observation indicates that it is promising to improve vulnerability detection by combining the results of different tools, although we have not seen any hybrid approach built in this way.

The overall F-score comparison among tools is CryptoGuard>CogniCrypt>Xanitizer> SonarQube>FindSe- cBugs>CryptoTutor. CryptoTutor obtained much lower measurements than other tools for two reasons. First, it has implementation issues, which prevented the

tool from correctly identifying API misuses in many cases. Second, CrytoTutor scans code for fewer patterns than other tools, and could not reveal the API misuses beyond its pattern set. SonarQube and FindSecBugs worked worse than CogniCrypt, CryptoGuard, and Xanitizer for two reasons. First, SonarQube and FindSecBugs have smaller pattern sets. Second, both tools apply intra-procedural instead of inter-procedural analysis to locate some API misuses (e.g., using constant IV values), although the inter-procedural program analysis is more desirable. As CryptoBench has many programs that require sophisticated inter-procedural analysis, neither SonarQube nor FindSecBugs handled well those programs.

Fig. 3.1 shows a program whose API misuse was not detected by either SonarQube or FindSecBugs. On line 9, `Cipher.getInstance(...)` is invoked with parameter `crypto`, whose actual value "`Blowfish`" implies the adoption of a provenly insecure algorithm. To facilitate understanding, in Fig. 3.1, we underlined all statements involved in the backward slice of that API invocation (i.e., lines 2–4, 6, 9, 13–14); we also marked the data-dependencies between statements with arrowed blue curves. As implied by the data dependencies, a tool has to conduct inter-procedural backward slicing in order to reveal the API misuse. Nevertheless, SonarQube and FindSecBugs failed to do that.

MUBench We applied five tools (except CryptoTutor) to MUBench, because CryptoTutor is quite unique. When running tools other than CryptoTutor, we managed to launch tools via commands, apply each tool to every program benchmark, and automatically process the output text (or files) to compute tools' P/R/F values. However, CryptoTutor is an interactive coding assistance tool that is delivered as an Eclipse plugin. To launch the tool, we have to first launch a new instance of Eclipse, import all programs into the IDE for each benchmark, manually right-click all programs to launch CryptoTutor, read outputs in GUI panes, and manually inspect subject programs as needed to compute P/R/F values. Such frequent manual interference can be extremely time-consuming, when there are hundreds of

open-source programs in MUBench. As we could not afford the manual effort, we did not do the experiment of applying CryptoTutor to MUBench.

Our results are shown in Table 3.8. We observed similar phenomena in this table and Table 3.6. Among the six Java types covered by MUBench, SonarQube revealed API misuses for the fewest Java types (i.e., three). Only two types have API misuses commonly detected by all tools: `Cipher` and `MessageDigest`. SonarQube achieved the highest F score (i.e., 86%) for misuses of `Cipher`'s method APIs, and CryptoGuard achieved the highest F score (i.e., 89%) for `MessageDigest`-related misuses. No tool consistently outperformed others. The overall F-score comparison is Xanitizer>CogniCrypt>CryptoGuard>FindSecBugs>Sona- rQube. FindSecBugs and SonarQube obtained much lower F scores than CogniCrypt, CryptoGuard, and Xanitizer.

OWASP As shown in Table 3.9, OWASP Benchmark covers a lot fewer Java types than the two benchmarks. Among the three Java types covered, SonarQube only detected API misuses for `Cipher`. Xanitizer worked perfectly to report misuses without any incorrect result. The overall F-score comparison among tools is Xanitizer>FindSecBugs>CryptoGuard>CogniCrypt>Sona-rQube. This comparison seems contradictory with what we observed in Tables 3.6 and 3.8. Namely, compared with CryptoGuard and CogniCrypt, FindSecBugs worked better on this dataset but worse on the other datasets.

To understand the contradiction, we further inspected the codebases of FindSecBugs and OWASP benchmark. We realized that in the benchmark, there are multiple programs having the same code underlined in Fig. 3.2. The common code calls `Cipher.getInstance(...)` with parameter `algorithm`, whose actual string value "`AES/CCM/NoPadding`" implies the adoption of a secure algorithm. However, CryptoGuard and CogniCrypt are not rigorous enough to determine that `algorithm` does not hold the value of "`AES/ECB/PKCS5Padding`". Consequently, they falsely inferred that "`AES/ECB/PKCS5Padding`" is passed to `Cipher.`

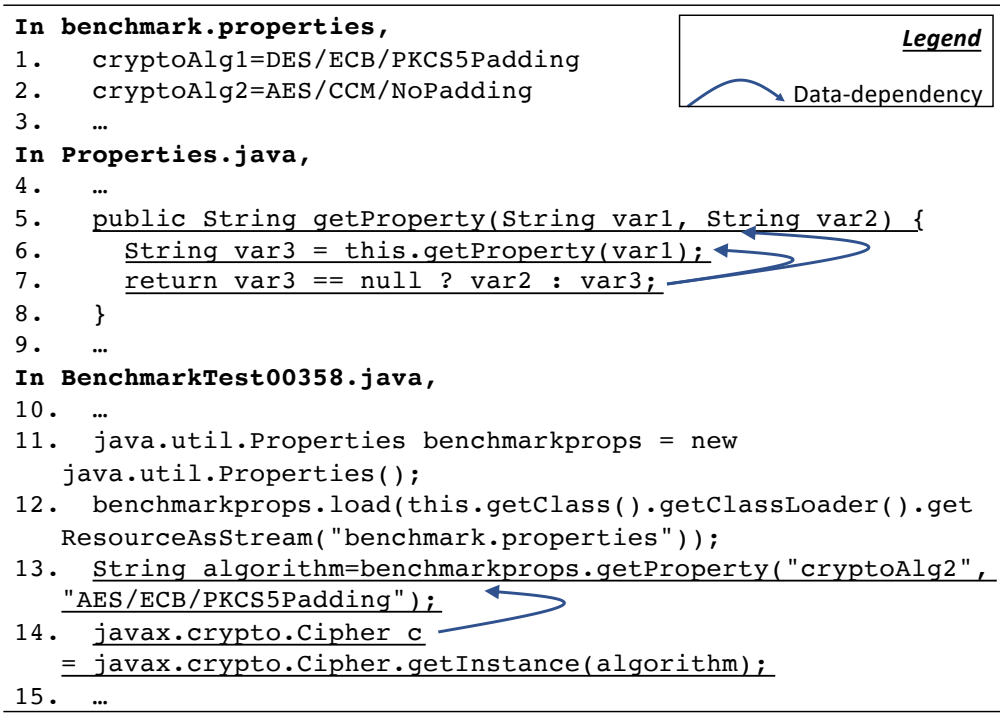

Figure 3.2: A program that is falsely reported to be vulnerable by CryptoGuard and CogniCrypt

`getInstance(...)`, and reported API misuses. In comparison, FindSecBugs nicely handled these programs and did not report any false positive.

Finding 3 (for RQ2): *No tool consistently worked best. However, CogniCrypt, CryptoGuard, and Xanitizer always outperformed SonarQube, probably due to their sophisticated inter-procedural analysis and larger pattern sets.*

Among the experimented tools, we observed the highest F-score that they can achieve on CryptoBench is 85% (by CryptoGuard); the highest measurement on MUBench is 72% (by Xanitizer). These numbers imply that there is still improvement space for new approaches to detect misuses with higher F-scores. Additionally, we noticed that each adopted benchmark only covers at most 11 of the 13 Java types frequently involved in cryptographic API misuses. It means that to better assess the effectiveness of different tools, we also need new benchmarks that cover various API misuses related to all Java types.

3.4 User Study

To understand how existing tools help with developers' secure coding practices, we performed a user study. We reported cryptographic API misuses found in open-source projects to owner developers, to seek for their feedback. Specifically, we ranked all Apache projects on GitHub in the descending order of their popularity (i.e., star counts). We then scanned the source code of top-ranked projects to find 200 projects that use any of the 13 Java types listed in Table 3.1. We chose to explore Apache projects because (1) they are usually well maintained, and (2) the project developers are experienced and often respond to pull requests (PRs).

Next, we applied all 5 experimented tools to the 200 Apache project to reveal cryptographic API misuses. Because the vulnerability reports by different tools contain true API misuses, together with false ones and other security issues out of scope, we needed to manually refine those reports before contacting developers for their feedback. To reduce our manual effort, in each tool's outputs, we sampled 15 projects. As we used 5 tools, in total we sampled 75 projects based on the reported vulnerabilities.

According to our manual analysis, the tools reported 416 true positives among the sampled projects. As it is infeasible for developers to respond to all instances, we further sampled 57 instances by taking a couple of steps. In Step 1, we classified all instances based on the Java types they are associated with. In Step 2, we randomly chose seven unique instances for each Java type to file PRs, in order to get developers' feedback on different kinds of misuses. When there are insufficient instances reported for any Java type (e.g., `IvParameterSpec`), we included all instances. In each PR, we specified (a) the code location of an instance, (b) the security implication, (c) one or two CWE entries showing the potential exploits, and (d) tool-generated guidance on fixes. When developers asked us to file issue reports, we also created issues to describe the above-mentioned information.

Table 3.10: Summary of developers' responses to 57 PRs

Java Type API	PRs filed	Developers' Feedback		
		Positive	Negative	No Response
`Cipher`	7	5	2	0
`HostnameVerifier`	2	1	1	0
`IvParameterSepc`	1	1	0	0
`KeyPairGenerator`	3	1	1	1
`KeyStore`	7	0	6	1
`MessageDigest`	7	1	6	0
`PBEParameterSpec`	5	4	0	1
`SecretKeyFactory`	3	0	1	2
`SecretKeySpec`	4	0	2	2
`SecureRandom`	7	0	5	2
`SSLContext`	4	1	2	1
`TrustManager`	7	3	4	0
Total	**57**	**17**	**30**	**10**

Based on our interactions with developers so far, we have classified developers' opinions into three categories: positive feedback, negative feedback, and no response. Surprisingly, developers rejected 53% of PRs (i.e., 30/57), agreed with us for 30% of PRs (i.e., 17/57), and did not respond for 18% of PRs (i.e., 10/57). PRs related to `Cipher` got the highest positive rate (i.e., 5/7). However, PRs related to another four Java types received zero positive response, including `KeyStore`, `SecretKeyFactory`, `SecretKeySpec`, and `SecureRandom`. Such comparison implies that developers considered certain vulnerability reports to be more important than others.

Finding 4 (for RQ3): *Developers would like to address the security issues mentioned in 17 PRs, but rejected 30 PRs. This phenomenon implies that developers are usually negative towards the reported security-API misuses.*

Among the 17 PRs with positive feedback, developers finally accepted 9 PRs. They mentioned two challenges posed by the other eight PRs:

Challenge 1. Incomplete fixing suggestion. In four PRs, tool-generated guidance does not offer all needed information for repairs. For instance, a project invokes

(a) An incomplete fixing suggestion by existing tools

```
1.    public static String encrypt(String text, SecretKey key) {
2.      …
3. -    Cipher cipher = Cipher.getInstance(“AES/ECB/PKCS5PADDING”);
4. +    Cipher cipher = Cipher.getInstance(“AES/CBC/PKCS5PADDING”);
5.      cipher.init(Cipher.ENCRYPT_MODE, key);
6.      … }
```

(b) The complete fixing suggestion that developers need

```
1. + private static IvParameterSpec iv = genIV();
2.   public static String encrypt(String text, SecretKey key) {
3.     …
4. -   Cipher cipher = Cipher.getInstance(“AES/ECB/PKCS5PADDING”);
5. +   Cipher cipher = Cipher.getInstance(“AES/CBC/PKCS5PADDING”);
6. -   cipher.init(Cipher.ENCRYPT_MODE, key);
7. +   cipher.init(Cipher.ENCRYPT_MODE, key, iv);
8.     … }
9. + public static IvParameterSpec genIV ( ) {
10.+   SecureRandom se = new SecureRandom ( ) ;
11.+   byte[] raw = new byte[16];
12.+   se.nextBytes(raw);
13.+   IvParameterSpec iv = new IvParameterSpec(raw);
14.+   return iv;
15.+ }
```

Figure 3.3: The incomplete and complete fixing suggestions for the misuse `Cipher.getInstance("AES/ECB/PKCS5PADDING")`

`Cipher.getInstance("AES/ECB/PKCS5PADDING")`, which misuse was located by tools and developers were recommended to replace the parameter with `"AES/CBC/PKCS5PADDING"`. However, naïvely applying this edit can cause a runtime error, because the substitute parameter requires for an additional initialization vector (IV) parameter when the cipher is used for encryption/decryption. With more details, Fig. 3.3 contrasts an incomplete fixing suggestion implied by existing tools and the complete suggestion that developers need. As reflected by Fig. 3.3 (b), when replacing the insecure parameter value with a secure one, developers must remember to also update a related API call `cipher.init(...)` (lines 6–7) and create an IV parameter for that updated call (line 1 and lines 9–15). As current tools do not guarantee the comprehensiveness or completeness of their coding suggestions, developers hesitate to modify code based on the partial information.

Challenge 2. Complex repair procedures. Some developers concurred with the revealed vulnerabilities in six PRs, but could not cope with the complexity of secure solutions. For instance, when implementing `TrustManager`, developers understood that they should not blindly trust all clients and servers, neither should they have empty implementation for the interface methods `checkClientTrusted(...)` and `checkServerTrusted(...)`. However, to remove the vulnerability, they have to not only revise code in order to check the certificates of both clients and servers, but also download certificate files to local machines and properly configure a set of local files. This process is challenging and time-consuming, and developers have almost zero tool assistance for non-code artifact configuration.

Finding 5 (for RQ3): *For 8 PRs, developers were willing to address the reported vulnerabilities but could not do that. They need tools to provide more detailed suggestions on repairing edits and non-code artifact configuration.*

Developers rejected 30 PRs for follwing reasons:

Reason 1. No exploit demo. For four PRs, developers do not trust the described API misuses or related security implications; they required actual security attacks to demonstrate the security exploit. Particularly in one PR, we pinpointed the insufficient key length of an RSA key pair, provided guidance on fixes, and included CWE-327 [17] as a reference. However, developers still need more convincing reasons before accepting the PR. They replied, "*we were unable to identify any security impact. As such, this has been marked as Not Applicable. If you still believe this to be valid, please submit a new report which includes detailed information demonstrating and exploiting the security impact for this issue*".

Reason 2. False positives without actual security impact. Among the 26 PRs, developers believed that the reported vulnerabilities in 10 PRs exist only in outdated code (i.e., archived files or repositories), or in test suites that will not be included into the released software products. As the reported vulnerabilities will not influence their software products, developers do not want to fix the reported misuses. For example, some `TrustManager` implementation was intentionally designed to trust all incoming connections in test code; developers mentioned that they understood the listed concerns in PRs. However, they have two reasons not to fix the reported issues: (1) all these test files will not be shipped with the projects; (2) the trust-all mechanism was implemented on purpose.

For the remaining 16 PRs, developers considered the reported vulnerabilities to be totally irrelevant, as the API misuses are not located in security-sensitive software implementation. 11 of these PRs are about the usage of `MessageDigest` and `Random`. Although tools consider "`MD5`" and "`SHA-1`" as insecure parameters to use when calling `MessageDigest.getInstance(...)`, developers totally disagreed on that. They defended that in their circumstances, these APIs were called not for cryptographic hashing or signature requests; instead, the APIs were used only to generate checksums for data integrity checks. Thus, they do not believe the usage of MD5 or SHA-1 to be security-sensitive. Listing 3.4 shows a

scenario where MD5 is used to generate message digests [4]. Concerning the code, developers explained, "*From a cursory check through Phoenix code, I see two uses of MD5: 1. An 'MD5' SQL function that allows users to generate MD5 fingerprints of columns. 2. To compare columns of primary and index tables against each other in the index scrutiny tool. Neither of these needs a cryptographic hash*".

```
1 public MD5Function(List<Expression> children) throws SQLException {
2   super(children);
3   try {
4     messageDigest = MessageDigest.getInstance("MD5");
5   } catch (NoSuchAlgorithmException e) {
6     throw new RuntimeException(e);
7   }
8 }
```

Listing 3.4: A scenario where `MD5` is irrelevant to security [4]

Similarly, for `Random`, although tools consider the API to be a cryptographically weak random value generator and consistently suggest `SecureRandom` as a secure substitute, developers disagreed. For instance, the code in Listing 3.5 uses `Random` to randomly pick an endpoint as the target for gossip (lines 18–22). The developers responded "*The gossiper does not use any random calls for cryptography*". This response indicates that developers only used randomly generated values for purposes irrelevant to cryptography, so they did not believe that their API usage is vulnerable.

Finding 6 (for RQ3): *Developers rejected 30 PRs because they disagreed upon the criteria tools adopted to recognize vulnerabilities. They need tools to demonstrate security exploits of vulnerabilities, and to skip issues located in test cases, archived code, and security-irrelevant context.*

```
1  public class Gossiper implements IFailureDetectionEventListener, GossiperMBean
2  {
3    private final Random random = new Random();
4    ...
5    /**
6     * Returns true if the chosen target was also a seed. False otherwise
7     *
8     * @param message
9     * @param epSet   a set of endpoint from which a random endpoint is chosen.
10    * @return true if the chosen endpoint is also a seed.
11    */
12   private boolean sendGossip(Message<GossipDigestSyn> message, Set<InetAddressAndPort> epSet) {
13     List<InetAddressAndPort> liveEndpoints = ImmutableList.copyOf(epSet);
14     int size = liveEndpoints.size();
15     if (size < 1)
16       return false;
17     /* Generate a random number from 0 -> size */
18     int index = (size == 1) ? 0 : random.nextInt(size);
19     InetAddressAndPort to = liveEndpoints.get(index);
20     ...
21     boolean isSeed = seeds.contains(to);
22     GossiperDiagnostics.sendGossipDigestSyn(this, to);
23     return isSeed;
24   }
25 }
26
```

Listing 3.5: A case where `Random` is irrelevant to security [13]

3.5 Threats to Validity

3.5.1 Threats to External Validity

Our empirical findings may be limited to the misuse patterns we focused on, the tools we experimented with, the datasets used, and the developers who responded to our PRs. To mitigate this limitation, we intentionally included the misuse patterns frequently mentioned

in literature, ran as many tools as possible, randomly sampled the most popular 200 Apache projects, and proactively discussed with developers on filed PRs. In the future, we will include more patterns, expand our datasets, and file more PRs.

3.5.2 Threats to Construct Validity

CryptoBench and OWASP Benchmark solely have crafted code with injected API misuse instances; they may not represent actual API misuses in real-world software. MUBench contains cryptographic API misuses in open-source programs; however, the ground truth of labeled misuses seems incomplete, and they may not represent API misuses in closed-source software. Therefore, our tool evaluation results may not reflect these tools' actual effectiveness in the real world. In the future, we will construct more comprehensive benchmarks using more real-world programs.

3.5.3 Threats to Internal Validity

In the user study, we manually checked tools' outputs, removed false alarms, and only sampled true misuses to file PRs. It is possible that our manual analysis is subject to human bias. To mitigate the problem, we had two authors inspect each sampled instance. In this way, we ensured that every filed PR contains a true misuse based on the pattern set defined in literature; when developers considered any PRs to be false positives, it indicates the discrepancy between developers' belief and research literature.

3.6 Summary

With the existence of tools that detect cryptographic API misuses in Java programs, some people believed that the research problem is well solved. Our work intended to assess current tools in different aspects and to reveal the gaps between existing work and developers' needs. Namely, we explored the question: *Are existing tools good enough to help developers eliminate cryptographic API misuses?*

Our quantitative and qualitative analysis of existing tools revealed several interesting findings. First, there is no tool consistently outperforming other tools. Currently, the most advanced tools detect API misuses using inter-procedural program analysis. However, developers still need better detectors, which conduct more accurate inter-procedure analysis and perform context-aware analysis to report API misuses in security-focused implementation. Second, although some tools provide general guidance on misuse repairs, they are insufficient to help developers correctly remove misuses. More detailed and customized repairing suggestions are still desperately needed. Third, although some tools explain reported API misuses by citing vulnerabilities described on CWE, such citations are sometimes unconvincing to developers. It will be better if future tools can automatically synthesize program-specific attacks and detail the procedure of security exploits.

Our study shows that the problem of cryptographic API misuse detection is far from being well solved. Next, we will build a novel tool to handle detection extensibility and suggest customized repairs to help developers secure the code practice better.

Chapter 4

Example-Based Vulnerability Detection and Repair in Java Code[1]

As the investigation result shown in Chapter 3, existing tools are insufficient to help developers eliminate security-API misuses. Table 4.1 summarizes both capability and extensibility of the mainstream techniques, and compares the tools with our new approach SEADER. As shown in the table, existing tools usually represent cryptographic API misuses as built-in rules [30, 36, 103, 132, 159]; users cannot easily extend these tools to detect more API-related vulnerabilities. As more security libraries emerge and evolve, we believe that vulnerability detectors should have good extensibility to keep their pattern sets of API-misuses up-to-date. Although CogniCrypt [127] offers a domain-specific language (DSL), CrySL [128], for users to prescribe the usage templates of cryptographic APIs, users need to spend lots of time learning CrySL and crafting templates. VuRLE [133] infers templates from user-provided code examples. However, its algorithm does not observe the unique characteristics of security API-misuses (e.g., using an integer within certain range); thus, VuRLE cannot always detect or fix misuses effectively. Additionally, most existing tools merely report misuses, without suggesting any customized fixes. When developers lack the cybersecurity knowledge to understand the reported misuses, they may continue making mistakes when trying to fix those issues independently [170]. Although CDRep [132] and VuRLE [133] can suggest

[1] In this chapter, API Misuses and API Insecure Usage are interchangeable.

Table 4.1: Comparison of SEADER against the existing detectors for security-API misuses

Tool	API-Misuse Representation			Misuse-Matching Strategy			Output	
	Built-in Rule	Template	Other	Intra-procedural Analysis	Inter-procedural Analysis	Other	Misuse	Repair
CryptoLint [103]	✓				✓		✓	
CDRep [132]	✓				✓		✓	✓
CogniCrypt [127]		✓			✓		✓	
CryptoGuard [159]	✓				✓		✓	
FindSecBugs [30]	✓				✓		✓	
Fischer et al.'s tool [106]			✓	✓		✓	✓	
SonarQube [36]	✓				✓		✓	
VuRLE [133]		✓		✓			✓	✓
SecureSync [155]			✓	✓			✓	
SEADER		✓			✓		✓	✓

customized fixes, they are separately limited by (1) the inextensible hardcoded pattern set and (2) the intra-procedural analysis adopted for template matching.

To persuade developers into removing detected API misuses, it is important to provide actionable suggestions on how to correctly use those APIs in developers' circumstances. Developers found existing repair guidance to be insufficient. Therefore, to better help developers, we still need tools to suggest both code solutions and related non-code configurations. Each suggested code solution should be complete: it not only replaces problematic API calls with correct ones, but also adjusts related code for correct program syntax and semantics. Each non-code configuration should be clear enough for developers to follow in order to properly manipulate their local file system.

To overcome the limitations of existing systems, we introduce SEADER (short for "security-API misuse detection and repair")—our new approach for vulnerability detection and repair from a data-driven perspective. As shown in Figure 4.1, there are two phases in SEADER: pattern inference and pattern application. In Phase I, suppose that a domain expert (e.g., security researcher) provides

- ***I***—insecure code with certain security-API misuse, and
- ***S***—the secure counterpart showing the correct API usage.

SEADER compares the two code snippets and detects program changes that can transform ***I*** to ***S***. Next, based on those changes, SEADER conducts *intra-procedural* analysis to derive a vulnerability-repair pattern. Each pattern has two parts: (i) a vulnerable code template together with matching-related information, and (ii) the abstract fix. SEADER stores all inferred patterns into a JSON file. In Phase II, given a program ***P***, SEADER loads patterns from the JSON file, and conducts *inter-procedural* program analysis to match code with

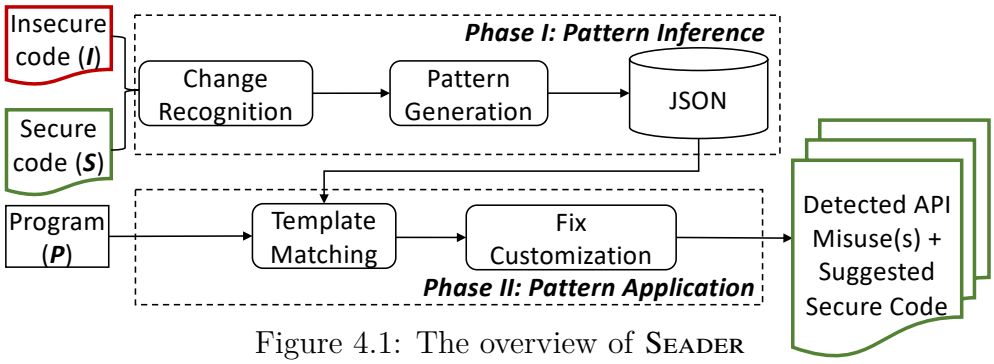

Figure 4.1: The overview of SEADER

any template. For each code match, SEADER concretizes the corresponding abstract fix, and suggests code replacements to developers.

According to the existing API-misuse patterns mentioned in prior work [106, 159], there are three unique kinds of security-API misuses that are hard to express with plain code examples, and are thus difficult to infer for existing program differencing-based approaches (e.g., VuRLE and SecureSync). Such misuses are about API invocations with (i) constants instead of random values, (ii) multiple alternative specialized constants, or (iii) constants in certain value ranges (see Section 4.2.5 and Table 3.1 for more details). To facilitate users to describe these patterns via code examples, we defined three novel specialized ways of example specification, and developed SEADER to specially infer patterns from those examples.

For evaluation, we crafted 28 ⟨insecure, secure⟩ code pairs based on the API-misuse patterns summarized by prior research. After SEADER inferred patterns from those pairs, we further applied SEADER to two program datasets to evaluate its effectiveness in vulnerability detection and repair. When applied to the first dataset, SEADER detected vulnerabilities with 95% precision, 72% recall, and 82% F-score. After applying SEADER to the second dataset, we inspected 77 repairs output by SEADER and found 76 of them correct.

4.1 A Motivating Example

This section overviews our approach with several code examples. Prior work shows that the security of symmetric encryption schemes depends on the secrecy of shared key [103]. Thus, developers should not generate secret keys from constant values hardcoded in programs [106]. Suppose a security expert **Alex** wants to detect and fix such vulnerabilities using Seader. Alex needs to craft (1) an insecure code example to show the API misuse, and (2) a secure example for the correct API usage. As shown in Figure 4.2, the insecure code ***I*** invokes the constructor of `SecretKeySpec` by passing in a constant array. Here, `ByteLiterals.CONSTANT\_ARRAY` is the specialized way that Seader requires users to adopt when they represent any byte-array constant. Meanwhile, the secure code ***S*** invokes the same API with `key`—a generated unpredictable value.

Given the two examples, Seader generates abstract syntax trees (ASTs) and compares them for any AST edit operation. For Figure 4.2, Seader creates an expression update and multiple statement insertions. The update operation replaces `ByteLiterals.CONSTANT\_ARRAY` with `key`. Next, based on the updated expression in ***I***, Seader conducts data-dependency analysis to find any security API that uses the expression, and treats it as a **critical API**. Such critical APIs are important for Seader to later detect similar vulnerabilities in other codebases. Afterwards, Seader generalizes a **vulnerability-repair pattern** from the examples by abstracting away concrete variable/method names and edit-irrelevant code. As shown in Figure 4.3, the generalized pattern has two parts: the vulnerability template (T) together with matching-related data, and an abstract fix (F). Such pattern generalization ensures the transformation applicable to codebases with distinct program contexts.

With a pattern inferred from the provided code pair, Alex can further apply Seader to an arbitrary program ***P***, to detect and fix any occurrence of the described vulnerability. In par-

Insecure code (*I*)

```
1  void test() {
2    SecretKey sekey= new SecretKeySpec(ByteLiterals.CONSTANT_ARRAY, "AES");
3  }
```

Secure code (*S*)

```
1  // store the key as a field for reuse purpose
2  byte[] key = keyInit();
3
4  // create a key based on an unpredictable random value
5  public byte[] keyInit() {
6    try {
7      KeyGenerator keyGen=KeyGenerator.getInstance("AES");
8      keyGen.init(256);
9      SecretKey secretKey = keyGen.generateKey();
10     byte[] keyBytes= secretKey.getEncoded();
11     return keyBytes;
12   } catch (Exception e) {
13     e.printStackTrace();
14     return null;
15   }
16 }
17
18 void test() {
19   SecretKey sekey= new SecretKeySpec(key, "AES");
20 }
```

Figure 4.2: A pair of examples to show the vulnerability and repair relevant to secret key creation

Vulnerable code template (T)

SecretKey v_0 = new SecretKeySpec(ByteLiterals.CONSTANT_ARRAY, "AES");

Matching-related data:

critical API: javax.crypto.spec.SecretKeySpec.SecretKeySpec(byte[], String)
other security APIs: {}

Abstract fix (F)

Replace the matched statement with:

SecretKey v_0 = new SecretKeySpec(v_1, "AES");

Add these lines before the container method of the matched statement:

```
1  // store the key as a field for reuse purpose
2  byte[] $v_1$ = $m_0$();
3
4  // create a key based on an unpredictable random value
5  public byte[] $m_0$() {
6      try {
7          KeyGenerator $v_4$=KeyGenerator.getInstance("AES");
8          $v_4$.init(256);
9          SecretKey $v_3$ = $v_4$.generateKey();
10         byte[] $v_2$= $v_3$.getEncoded();
11         return $v_2$;
12     } catch (Exception $v_5$) {
13         $v_5$.printStackTrace();
14         return null;
15     }
16 }
```

Figure 4.3: The pattern inferred from the code pair in Figure 4.2

ticular, given a program whose simplified version is shown in Listing 4.6, Seader first scans for any invocation of the critical API `SecretKeySpec(...)`. If no such invocation exists, Seader concludes that $\boldsymbol{P}$ does not have the above-mentioned vulnerability; otherwise, if the API is invoked (see line 8 in Listing 4.6), Seader then searches for any code matching the template in Figure 4.3. The template-matching process conducts inter-procedural analysis and checks for two conditions:

C1: Is the first parameter derived from a constant?

C2: Does the second parameter exactly match `"AES"`?

If any invocation of `SecretKeySpec(...)` satisfies both conditions, Seader reports the code

```
1  public class CEncryptor {
2    private char[] passPhrase;
3    private String alg = "AES";
4    public CEncryptor(String passPhrase) {
5      this.passPhrase = passPhrase.toCharArray();
6    }
7    public Result encrypt(byte[] plain) throws Exception {
8      SecretKey secret = new SecretKeySpec(new String(passPhrase).getBytes(), alg);
9    ...
10 }
11 public class Main {
12   public static void main(String[] args)
13   CEncryptor aes0 = new CEncryptor("password");
14   aes0.encrypt((byte[])args[0]);
15   ...
16 }
```

Listing 4.6: A simplified version of $\boldsymbol{P}$

to be vulnerable. Notice that if we only check line 8 of Listing 4.6, neither `newString(passPhrase).getBytes()` nor `alg` satisfies any condition. Thanks to the usage of inter-procedural analysis, SEADER can perform backward slicing to trace how both parameters are initialized. Because `alg` is a private field of `CEncryptor`, whose value is initialized on line 3 with `"AES"`, SEADER decides that C2 is satisfied. Similarly, `passPhrase` is another field whose value is initialized with a parameter of the constructor `CEncryptor(...)` (lines 4-6). When `CEncryptor(...)` is called with parameter `"password"` before the invocation of `SecretKeySpec(...)` (lines 7-14), C1 is satisfied. Therefore, SEADER concludes that line 8 matches the template; it matches concrete variable `secret` with the template variable `\$v\_0\$`.

For the found code match, SEADER customizes the abstract fix shown in Figure 4.3 by replacing the abstract variable `\$v\_0\$` with concrete variable `secret`. As shown in Figure 4.4, the customized fix first initializes a `KeyGenerator` instance with the algorithm "`AES`" and

Replace the matched statement with:
SecretKey secret = new SecretKeySpec(v_1, "AES");
Add these lines before the method encrypt(byte[] plain):

```
1  // store the key as a field for reuse purpose
2  byte[] $v_1$ = $m_0$();
3
4  // create a key based on an unpredictable random value
5
6  public byte[] $m_0$() {
7      try {
8          KeyGenerator $v_4$=KeyGenerator.getInstance("AES");
9          $v_4$.init(256);
10         SecretKey $v_3$ = $v_4$.generateKey();
11         byte[] $v_2$= $v_3$.getEncoded();
12         return $v_2$;
13     } catch (Exception $v_5$) {
14         $v_5$.printStackTrace();
15         return null;
16     }
17 }
```

Figure 4.4: A customized fix for $\boldsymbol{P}$ suggested by Seader

the key size "256", to generate an unpredictable AES key (lines 6–8). Next, the AES key is converted to a byte array (line 9), which value can be stored into a Java field so that the value is reusable by both encryption and decryption modules. Additionally, inside the method `encrypt(...)`, the original vulnerable statement is updated to create a secret key using the generated byte array.

4.2 Approach

There are two challenges to overcome in our research:

1. How can we infer generalized vulnerability-repair patterns from concrete ⟨insecure, secure⟩ code examples?

2. How can we ensure that the inferred patterns are applicable to code that is different

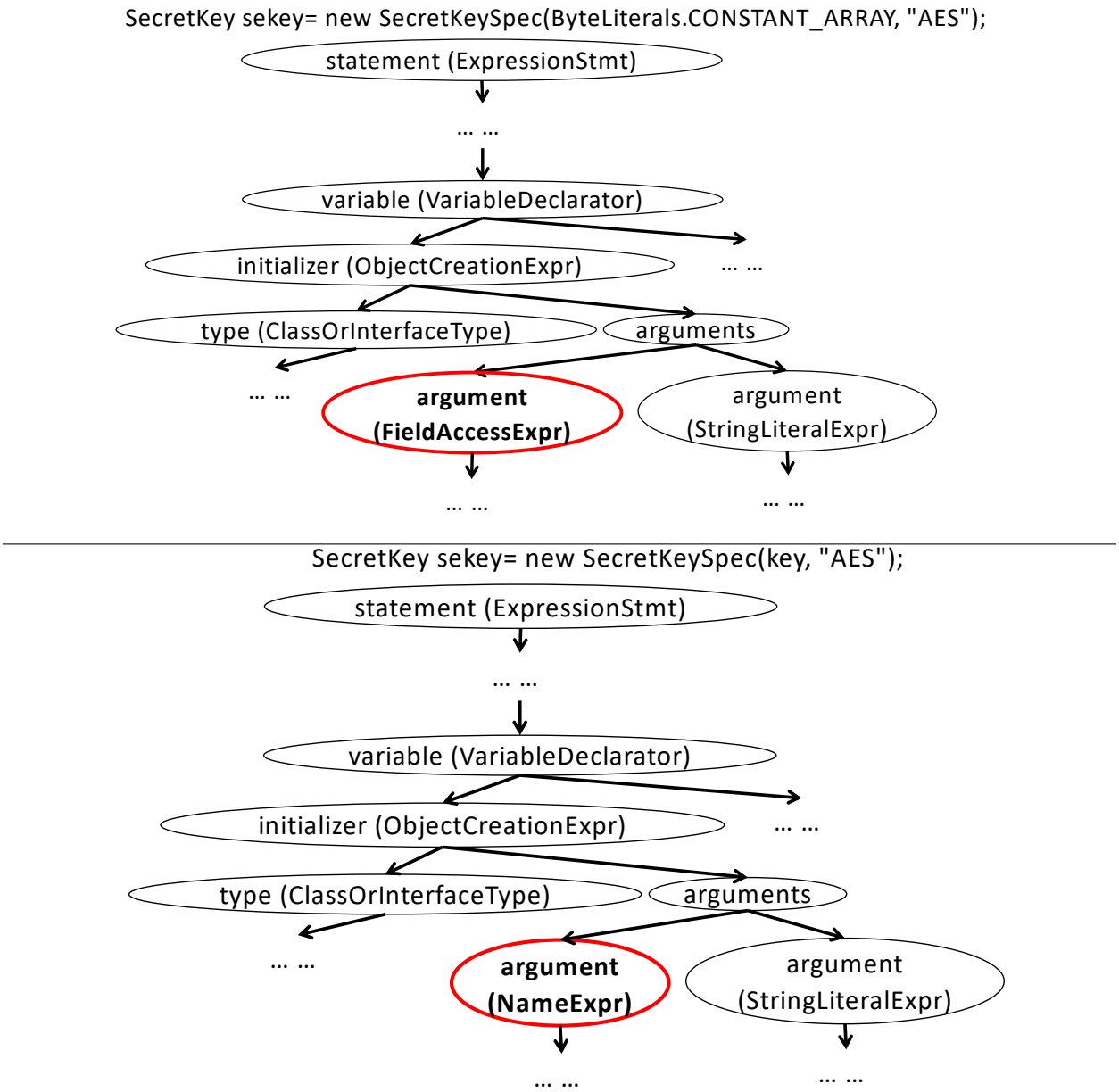

Figure 4.5: The simplified ASTs of the two statements related to a statement-level update operation

from the original examples?

To address these challenges, as shown in Figure 4.1, we designed two phases in Seader. The first phase takes two steps to infer vulnerability-repair patterns from ⟨insecure, secure⟩ code examples; the second phase contains another two steps to apply inferred patterns to given programs. In this section, we will first describe each of the four steps in detail (Section 4.2.1-Section 4.2.4). Next, we will explain the three specialized ways of example specification, which can facilitate users to demonstrate certain API misuses (Section 4.2.5).

4.2.1 Change Recognition

Given an $\langle I, S \rangle$ example pair, Seader compares code to locate (1) the root cause of any vulnerability demonstrated by I and (2) the security patch shown in S. Specifically, Seader applies syntactic program differencing to the code pair, to reveal any edit operation(s) that can transform I to S. This step consists of two parts: statement-level change recognition and expression-level change recognition.

Statement-level change recognition Seader first uses JavaParser [119] to generate ASTs for I and S, and then compares ASTs to create three types of edit operations:

- **delete (Node** a**)**: Delete node a.
- **insert (Node** a**, Node** b**, int** k**)**: Insert node a and position it as the $(k+1)^{th}$ child of node b.
- **update (Node** a**, Node** b**)**: Replace a with b. This operation changes a's content.

Specifically, when comparing any two statements $s_i \in I$ and $s_s \in S$, Seader checks whether the code string of s_i exactly matches that of s_j; if so, Seader considers s_i unchanged while

I is transformed to S. Otherwise, if the code strings of s_i and s_j are different, SEADER normalizes both statements by replacing concrete variables (e.g., `key`) with abstract ones (e.g., `\$v\_0`), and replacing constant values (e.g., `"AES"`) with abstract constants (e.g., `\$c\_0`). We denote the normalized representations as n_i and n_s. Next, SEADER computes the Levenshtein edit distance [130] between n_i and n_s, and computes the similarity score [109] with:

$$sim = 1 - \frac{edit\_distance}{max\_length(n_i, n_s)}$$

The similarity score sim is within [0, 1]. When $sim = 1$, n_i and n_j are identical. We set a threshold $th = 0.8$ such that if $sim >= th$, n_i and n_j are considered to match. In this way, SEADER can identify update operation(s). Compared with string-based match, the normalization-based match is more flexible, because it can match any two statements that have similar syntactic structures but distinct variables or constants. Finally, if a statement $s_i \in I$ does not find a match in S, SEADER infers a delete operation; if $s_s \in S$ is unmatched, SEADER infers an insert operation.

Expression-level change recognition For each statement-level update, SEADER tries to identify any finer-granularity edit (i.e., expression replacement) to better comprehend changes, and to prepare for later pattern generation (see Section 4.2.2). When s_i is updated to s_s, SEADER conducts top-down matching between ASTs to identify edits. Namely, while traversing both trees in a preorder manner, SEADER compares roots and inner nodes based on the AST node types, and compares leaf nodes based on the code content. Such node traversal and comparison continue until SEADER finds all unmatched subtrees or leaves.

For the example code shown in Figure 4.2, with statement-level change recognition, SEADER reveals one statement update and multiple statement additions. Figure 4.5 shows the sim-

plified ASTs of both before- and after- versions for the updated statement. By comparing the ASTs in a top-down manner, Seader finds the first arguments sent to the constructor to differ (e.g., `FieldAccessExpr` vs. `NameExpr`). Thus, Seader creates a finer-granularity operation to replace the statement-level update: **update (`ByteLiterals.CONSTANT\_ARRAY, key`)**.

Notice that we decided not to use existing tools, such as GumTree [105] and ChangeDistiller [109], to recognize changes for a variety of reasons. First, GumTree often mismatches nodes against developers' intent [136]. GumTree can generate four types of edit operations: add, delete, update, and move. However, in our research, we need only three edit types: add, delete, and update, so that Seader can infer API-misuse patterns from recognized changes. Second, ChangeDistiller only detects statement-level changes, without identifying expression-level changes. Additionally, it also generates four edit types. To avoid (1) fixing bugs in GumTree and (2) revising current tools to report three instead of four types of edit operations, we created our own program differencing algorithm.

4.2.2 Pattern Generation

When security experts present an $\langle I, S \rangle$ example pair to demonstrate any API misuse, we expect that they provide the code snippets to show only one vulnerability and its repair. Additionally, based on our experience with security-API misuses, each vulnerability is usually caused by the misuse of one security API. Therefore, to infer a general vulnerability-repair pattern from a given code pair, we need to overcome two technical challenges:

- How can we identify the security API whose misuse is responsible for the vulnerability (i.e., critical API)?
- How should we capture any relationship between the critical API and its surrounding

Insecure code (I)

```
1 void test(int iterations) {
2     byte[] salt = new byte[4];
3     AlgorithmParameterSpec paramSpec = new PBEParameterSpec(salt, iterations);
4 }
```

Secure code (S)

```
1 void test(int iterations) {
2   byte[] salt = new byte[8];
3   AlgorithmParameterSpec paramSpec = new PBEParameterSpec(salt, iterations);
4 }
```

Figure 4.6: An $\langle I, S \rangle$ where a critical API `PBEParameterSpec(...)` indirectly depends on a updated constant

code?

Task 1: Identifying the critical API Starting with the edit script E created in Section 4.2.1, SEADER looks for any update operation $update(e, e')$. If there is such an operation, SEADER searches for the security API whose invocation is data-dependent on e or e', and considers the API to be *critical*. For the example shown in Figure 4.5, the critical API is `SecretKeySpec(byte[],String)` because it is invoked with the updated expression as the first argument.

Similarly, Figure 4.6 presents another example where a numeric literal is updated from `4` to `8`. With data-dependency analysis, SEADER reveals that the constants are used to define variable `salt`, while `salt` is used as an argument when `PBEParameterSpec(...)` is invoked. Therefore, the method invocation depends on the updated expression, and the security API `PBEParameterSpec(byte[],int)` is considered *critical*.

If there is no update operation in E, SEADER searches for any overridden security API that encloses all edit operations, and considers the overridden API to be *critical*. Take the code pair shown in Figure 4.7 as an example. By comparing I with S, SEADER can identify one statement deletion and multiple statement insertions. As there is no update

Insecure code (*I*)

```
public class HostVerifier implements HostnameVerifier {
  @Override
  public boolean verify(String hostname, SSLSession sslSession){
    return true;
  }
}
```

Secure code (*S*)

```
public class HostVerifier implements HostnameVerifier {
  @Override
  public boolean verify(String hostname, SSLSession sslSession){
    //Please change "example.com" as needed
    if ("example.com".equals(hostname)) {
      return true;
    }
    HostnameVerifier hv = HttpsURLConnection.getDefaultHostnameVerifier();
    return hv.verify(hostname, sslSession);
  }
}
```

Figure 4.7: A pair of examples from which Seader infers the critical API to be an overridden method

operation and all edit operations are enclosed by an overridden method `verify(String, SSLSession)` (indicated by `@Override`), Seader further locates the interface or super class declaring the method (e.g., `HostnameVerifier`). If the overridden method together with the interface/super class matches any known security API, Seader concludes the overridden method to be *critical*.

Lastly, if no update operation or overridden security API is identified, Seader checks whether there is any deletion of security API call in E; if so, the API is *critical*. To facilitate later template matching (Section 4.2.3), for each identified critical API, Seader records the method binding information (e.g., `javax.crypto.spec.SecretKeySpec.SecretKeySpec(byte[], String)`).

Task 2: Extracting relationship between the critical API and its surrounding

code When a vulnerable code example has multiple statements (e.g., Figures 4.6 and 4.7), we were curious how the critical API invocation is related to other statements. On one extreme, if the invocation is irrelevant to all surrounding statements, we should not include any surrounding code into the generalized pattern. On the other extreme, if the invocation is related to all surrounding code, we should take all code into account when inferring a vulnerability-repair pattern. Thus, this task intends to decide (1) which statements of I to include into the vulnerable code template, (2) what additional security API call(s) to analyze for template matching (see Section 4.2.3), and (3) which statements of S to include into the abstract fix.

SEADER performs intra-procedural data-dependency analysis. If a statement defines a variable whose value is (in)directly used by the critical API invocation, the statement is extracted as *edit-relevant context*. SEADER uses such context to characterize the demonstrated vulnerability. For the insecure code I in Figure 4.6, since the API call (line 3) data-depends on variable `salt`, lines 2-3 are extracted as context. Additionally, when the critical API is an overridden method, its code implementation in I is considered edit-relevant context (see lines 3-4 in Figure 4.7). Based on the extracted edit-relevant context, SEADER abstracts all variables to derive a vulnerable code template T, and records mappings M between abstract and concrete variables. In addition to the critical API, SEADER also extracts binding information for any other security API invoked by the contextual code. Compared with edit-relevant context, these APIs provide more succinct hints. In our later template-matching process, these APIs can serve as "*anchors*" for SEADER to efficiently decide whether a program slice is worth further comparison with the template.

To locate the fix-relevant code in secure version S, SEADER identifies any unchanged code in the edit-relevant context, the inserted statements, and the new version of any updated statement. For the secure code S shown in Figure 4.6, lines 2-3 are fix-relevant, because

line 2 is the new version of an updated statement and line 3 is unchanged contextual code. Similarly, for the secure code S shown in Figure 4.7, lines 3-9 are fix-relevant, because lines 3 presents the critical API while lines 4-9 are inserted statements. Based on the above-mentioned variable mappings M and fix-related code, Seader further abstracts variables used in the fix-related code to derive an abstract fix F. Seader ensures that the same concrete variables used in I and S are mapped to the same abstract variables.

To sum up, given a $\langle I, S \rangle$ pair, Seader produces a pattern $Pat = \langle T, F \rangle$, which has a vulnerable code template T, an abstract fix F, and metadata to describe T (i.e., bindings of security APIs).

4.2.3 Template Matching

Given a program P, Seader uses a static analysis framework—WALA [16]—to analyze the program JAR file (i.e., bytecode). As shown by lines 1.2–1.4 in Algorithm 1, to find any code in P that matches the template T, Seader first searches for the critical API (i.e., invocation or method reimplementation). If the critical API does not exist, Seader concludes that there is no match for T. Next, if the critical API is invoked at least once, for each invocation, Seader conducts inter-procedural backward slicing to retrieve all code Sli on which the API call is data-dependent (i.e., `getBackwardSlice(x)`). When T invokes one or more security APIs in addition to the critical API, Seader further examines whether Sli contains matches for those extra APIs; if not, the matching trial fails (see lines 1.8–1.9). Next, Seader checks whether the matched code in Sli preserves the data dependencies manifested by T (i.e., `dataDependConsist(T,Sli)`). If those data dependencies also match, Seader reveals a vulnerability (see lines 1.10–1.11).

Alternatively, if the critical API is reimplemented, for each reimplementation, Seader com-

pares the code content against T, and reports a vulnerability if they match (see lines 1.13–1.14). At the end of this step, if any vulnerability is detected, SEADER presents the line number where the critical API is invoked or is declared as an overridden method, and shows related matching details. The matching details include both code matches and abstract-concrete variable mappings.

Actually, we designed our algorithm of template matching based on three considerations. First, as developers provide code examples in Java but WALA analyzes JAR files, template matching should leverage the minimum information (i.e., security APIs and variable data dependencies) to overcome any discrepancy between program representations (i.e., source code vs. bytecode). Second, although SEADER infers templates from simple code examples via intra-procedural analysis, we need to match code with templates step-by-step via inter-procedural analysis, so that SEADER can find matches even if the program context is more complicated. Third, many security-API misuses are relevant to parameter usage or method overriding, so our matching algorithm observes such unique characteristics to establish matches.

4.2.4 Fix Customization

This step involves two types of customization: variable customization and edit customization. To customize variables, based on the matching details mentioned in Section 4.2.3, SEADER replaces abstract variables in F with the corresponding concrete ones. We denote this customized version as F_c. For edit customization, SEADER suggests code replacements in two distinct ways depeding on the inferred edit operations mentioned in Section 4.2.1. Specifically, if there is only one update operation inferred, SEADER simply recommends an alternative expression to replace the original expression. Otherwise, SEADER presents F_c for developers to consider.

Algorithm 1: Matching Program P to template T

```
   Input: P, T, D /* program, template, and related metadata                    */
   Output: Matched /* a set of code matches from P to T                        */
1.1 Candi := ∅, Matched := ∅;
   /* 1. search for matches of the critical API                                 */
1.2 foreach code line x ∈ P do
1.3     if x invokes D(critical) // x declares D(critical) then
1.4         Candi := Candi ∪ x;
1.5 foreach x ∈Candi do
1.6     if x invokes D(critical) then
            /* 2(a). For API call, do program slicing and look for matches of other
               security APIs                                                    */
1.7         Sli = getBackwardSlice(x);
1.8         if (Sli has all matches for D(other)) == false then
1.9             continue;
            /* 3. check whether the data dependencies between security APIs in T match
               those in Sli                                                     */
1.10        if dataDependConsist(T, Sli) then
1.11            Matched:=Matched ∪ {Sli, mappings};
1.12    else
            /* 2(b). For API overriding, check the code                         */
1.13        if contentMatch(code(P, x), T) then
1.14            Matched := Matched ∪ {code(P, x), mappings};
```

Table 4.2: The stubs defined to ease example specification

Class	Members	Semantics
StringLiterals	StringLiterals (String... a)	This constructor creates a `StringLiterals` object with one or more string literals.
	getAString()	This method randomly returns one of the strings originally used to construct the `StringLiterals` object.
ByteLiterals	CONSTANT_ ARRAY	This field serves as a placeholder for a byte-array constant, whose value can be unspecified.
CharLiterals	CONSTANT_ ARRAY	This field serves as a placeholder for a char-array constant, whose value can be unspecified.

Notice that SEADER does not directly modify P to repair any vulnerability for two reasons. First, when template T contains multiple statements, it is possible that the corresponding code match involves statements from multiple method bodies. Automatically editing those statements can be risky and cause unpredictable impacts on program semantics. Second, some fixes require for developers' further customization based on their software environments (e.g., network configurations, file systems, and security infrastructures). As implied by Figure 4.7, the abstract fix derived from S will contain a comment `"//Pleasechange'example.com'asneeded"`, so will the customized fix by SEADER. This comment instructs developers to replace the standard hostname based on their circumstances.

4.2.5 Specialized Ways of Example Specification

We believe that by crafting $\langle I, S \rangle$ code pairs, security experts can demonstrate the misuse and correct usage of security APIs. However, we also noticed some scenarios where plain Java examples cannot effectively reflect the vulnerability-repair patterns. To solve this problem, we defined three **stub Java classes** (i.e., fake classes) for user adoption and invented **three specialized ways of example definition**. As shown in Table 4.2, the stub classes offer stub methods or fields to facilitate constant-related example specification. This section explains the scenarios where our special specification methods are needed.

Scenario 1. An API misuse involves an arbitrary constant value instead of any

particular constant. Plain examples only show the usage of particular constant values, but cannot generally represent the constant concept. Consider the vulnerability introduced in Section 4.1. Without using `ByteLiterals.CONSTANT\_ARRAY`, a domain expert has to define a plain example to show the API misuse, such as

```
SecretKeysekey=newSecretKeySpec("ABCDE".getBytes(),"AES");
```

Seader is designed to preserve all string literals from I when generalizing template T, and to look for those values when matching code with T. Consequently, given the above-mentioned example, Seader will inevitably embed `"ABCDE"` into the inferred template. To help users avoid such unwanted literal values in T, we defined `ByteLiterals.CONSTANT\_ARRAY` and `CharLiterals.CONSTANT\_ARRAY`. These static fields can be used as placeholders or wild-cards for constant arrays, to represent the general constant concept in examples. When Seader detects such fields in examples, it keeps them as they are in T and later matches them with constant values in P.

Scenario 2. An API misuse has multiple alternative insecure (or secure) options. Given a parameter of certain security API, suppose that there are (1) m distinct values to cause API misuse and (2) n alternatives to ensure correct API usage, where $m \geq 1$, $n \geq 1$. To express all possible combinations between the insecure and secure options via plain Java examples, users have to provide $m \times n$ pairs of examples, which practice is inefficient and undesirable. To solve this issue, we defined two stub methods in `StringLiterals`. As shown in Figure 4.8, one is a constructor of `StringLiterals`, which can take in any number of string literals as arguments (see line 1 in I) and store those values into an internal list structure. The other method is `getAString()`, which randomly picks and returns a value from that list (see line 2 in I). In this way, a domain expert can efficiently enumerate multiple secure/insecure options in just one code pair.

The examples in Figure 4.8 show that when security API `Cipher.getInstance(...)` is

Insecure code (I)

```
StringLiterals literals=new StringLiterals( "AES", "RC2", "RC4", "RC5", "DES",
    "blowfish", "DESede", "ARCFOUR");
Cipher.getInstance(literals.getAString());
```

Secure code (S)

```
StringLiterals literals = new StringLiterals("AES/GCM/NoPadding",
    "RSA/ECB/OAEPWithSHA-1AndMGF1Padding");
Cipher.getInstance(literals.getAString());
```

Figure 4.8: A code pair where multiple alternative secure and insecure options are specified simultaneously

called, the parameter may have one of the insecure values (e.g., `"AES"`). Such vulnerability can be addressed when the value is replaced by one of the three secure options (e.g., `"AES/GCM/NoPadding"`). Given the example in Figure 4.8, SEADER extracts insecure and secure options from `StringLiterals`-related statements, detects vulnerabilities in P if the security API is invoked with any insecure option, and suggests all secure alternatives.

Scenario 3. An API misuse requires for a parameter value in a specific range. Given an integer parameter p of certain API, suppose that there is a threshold value th such that the API invocation is secure only when $p \geq th$. To enumerate all possible vulnerable cases and related repairs via plain examples, theoretically, a user has to provide $(th - Integer.MIN\_VALUE) \times (Integer.MAX\_VALUE - th + 1)$ code pairs, which practice is infeasible. Therefore, we invented a special way of example definition, which requires users to provide only (1) one insecure example by setting p to a concrete value less than th and (2) one secure example by setting $p = th$. As shown in Figure 4.6, if a security expert wants to describe the pattern that *the array size of the first parameter should be no less than 8*, then s/he can define I by creating an array with a smaller size (i.e., 4) and define S by setting the size to 8. SEADER can identify the integer literals used by I and S, and infer the secure value range $size \geq 8$.

4.3 Evaluation

This section first describes the evaluation datasets and metrics, and then presents Seader's effectiveness of pattern inference. Next, it explains the tool effectiveness of pattern application, including vulnerability detection and repair. We did all experiments on Linux Mint 20.3 Cinnamon, version 5.2.7; we used Intel Core i7-8700 processor and 32GB memory.

4.3.1 Datasets

We used one dataset to evaluate pattern inference, and two datasets to evaluate pattern application.

Prior research revealed a number of security-API misuses and related correct usage in Java [2, 100, 104, 106, 125, 134, 137, 140, 159, 163]. To evaluate Seader's effectiveness of pattern inference, we referred to those well-described API misuses and fixes while crafting code examples for Seader. Table 3.1 lists the 13 security class APIs we focused on, the insecure usage of certain method API(s) frequently mentioned by prior work, and the secure usage. With this domain knowledge, we handcrafted 28 $\langle I, S \rangle$ pairs. Among the pairs, 19 pairs are defined in the specialized ways introduced in Section 4.2.5, and 9 pairs are defined with plain Java examples. Within the 19 pairs, 8 pairs, 6 pairs, and 5 pairs separately belong to Scenarios 1–3.

4.3.1.1 Two datasets to evaluate pattern application

The first dataset is a third-party benchmark, consisting of 86 real vulnerabilities from 10 Apache open-source projects [21, 89]. We decided to use this dataset for two reasons. First, it was created by other researchers, so it can be used to objectively assess the effectiveness

of different vulnerability detectors. Second, most of the 86 vulnerabilities belong to the 13 security classes shown in Table 3.1, so they can properly measure SEADER's capability of pattern application. The second dataset contains 100 widely used Apache open-source projects. To create this dataset, we first ranked the Apache projects available on GitHub [11] in a descending order of their popularity (i.e., star counts). Next, we located the top 100 projects that satisfy the following constraints: (1) the project uses the security APIs that SEADER examines; (2) the project is compilable because SEADER analyzes the compiled JAR files. The resulting dataset is used to evaluate SEADER's effectiveness of repair suggestion.

4.3.2 Metrics

As with prior work [159], we leveraged the following three metrics to measure tools' capability of vulnerability detection:

Precision (P) measures among all reported vulnerabilities, how many of them are true vulnerabilities.

$$P = \frac{\text{\# of correct reports}}{\text{Total \# of reports}}$$

When a tool reports a set of vulnerabilities S_1 and the known set of vulnerabilities is S_2, we intersected S_1 with S_2 to automatically compute precision. Namely, $P = |S_1 \cap S_2|/|S_1|$.

Recall (R) measures among all known vulnerabilities, how many of them are detected by a tool.

$$R = \frac{\text{\# of correct reports}}{\text{Total \# of known vulnerabilities}}$$

When a tool reports a set of vulnerabilities S_1, we intersected S_1 with the set of known

vulnerabilities S_2 to automatically compute recall, i.e., $R = |S_1 \cap S_2|/|S_2|$.

F-score (F) is the harmonic mean of precision and recall; it can reflect the trade-off between precision and recall.

$$F = \frac{2 \times P \times R}{P + R}$$

4.3.3 Effectiveness of Pattern Inference

As mentioned in Section 4.3.1, we crafted 28 code pairs to evaluate Seader's effectiveness of pattern inference. We categorized the 28 pairs based on two criteria:

C1. Do I and S contain single or multiple statements?

C2. Does pattern inference abstract variables?

The two conditions actually reflect the difficulty levels or challenges of these pattern inference tasks. For instance, if I or S has multiple statements, Seader conducts data-dependency analysis to locate the edit-relevant context in I or to reveal the fix-relevant code in S. If I or S uses variables, Seader abstracts all variable names to ensure the general applicability of inferred patterns. As shown in Table 4.3, there are four simplest pairs; Seader can handle these pairs without conducting any data-dependency analysis or identifier generalization. Meanwhile, there are 18 most complicated cases that require Seader to analyze data dependencies and generalize identifiers. In our evaluation, Seader correctly inferred patterns from all pairs. When some pairs present secure/insecure options (e.g., distinct string literals) for the *same* critical API, Seader merged the inferred patterns. In this way, Seader derived 21 unique patterns.

Table 4.3: The 28 code pairs for pattern inference

	Single statement	Multiple statements
Identical	4	5
Abstract	1	18

Finding 1: *Our experiment shows* SEADER*'s great capability of pattern inference.* SEADER *shows impressive extensibility by inferring patterns from various examples.*

4.3.4 Effectiveness of Vulnerability Detection

To assess SEADER's capability of vulnerability detection, we used a third-party dataset (see Section 4.3.1.1). We applied SEADER and three state-of-the-art vulnerability detectors (i.e., CogniCrypt [127], CryptoGuard [159], and FindSecBugs [30]) to all subject programs. SEADER spent 262 seconds analyzing all programs. As shown in Table 4.4, SEADER outperformed the other tools by acquiring the highest average recall (72%) and F-score (82%). It obtained the same average precision rate—95%—as CryptoGuard and FindSecBugs, which rate is much higher than that of CogniCrypt (i.e., 48%).

SEADER and CogniCrypt reported API misuses in eight projects, while the other two tools reported issues in nine projects. As the dataset labels no vulnerability in `tika.jar` and three tools (except CogniCrypt) found zero vulnerability in that project, we could not measure P, R, or F for these tools. CogniCrypt falsely reported three vulnerabilities in `tika.jar`, so its measured precision is 0%. SEADER was unable to analyze `tomee.jar`, because WALA does not work well with the JAR files built by Maven [32]. We believe that once WALA developers overcome the limitation between WALA and Maven JARs in the future, SEADER can also analyze `tomee.jar`. CogniCrypt crashed on two projects: `spark.jar` and `artemis-commons.jar`. Among the four tools under comparison, CryptoGuard obtained a slightly lower F-score than SEADER (78% vs. 82%), followed by FindSecBugs and CogniCrypt.

Table 4.4: Evaluation results on the 86-vulnerability dataset [21]

Apache Project	**# of Labeled Vulnerabilities**	**CogniCrypt**			**CryptoGuard**			**FindSecBugs**			**Seader**		
		P(%)	**R(%)**	**F(%)**	**P(%)**	**R(%)**	**F(%)**	**P(%)**	**R(%)**	**F(%)**	**P(%)**	**R(%)**	**F(%)**
deltaspike.jar	2	40	100	57	100	100	100	100	100	100	100	100	100
directory-server.jar	19	51	95	67	100	26	42	100	58	73	94	84	89
incubator-taverna-workbench.jar	5	57	80	67	100	80	89	100	40	57	80	80	80
manifoldcf.jar	3	17	33	22	60	100	75	60	100	75	75	100	86
meecrowave.jar	3	100	100	100	100	67	80	100	67	80	100	100	100
spark.jar	27	-	-	-	100	100	100	100	85	92	100	93	96
tika.jar	0	0	-	-	-	-	-	-	-	-	-	-	-
tomee.jar (openejb-core)	7	50	43	46	83	71	77	75	43	55	-	-	-
wicket.jar	5	50	60	55	100	100	100	100	40	57	100	60	75
artemis-commons.jar	15	-	-	-	100	27	42	100	27	42	100	40	57
Overall	86	48	40	43	95	66	78	95	60	74	95	72	82

"-" means no value is computed, because there is no labeled API misuse in the ground truth dataset or there is no tool-reported misuse.

Two possible reasons can explain SEADER's higher F-score. First, thanks to its great extensibility, SEADER has a larger pattern set of API misuses. Second, its inter-procedural analysis can accurately detect API misuses in more complex scenarios.

Analysis of False Positives. We manually inspected the cases where SEADER did not report misuses correctly. We found one reason to explain why SEADER did not achieve 100% precision: the ground truth is incomplete, as it labels some instead of all invocations of `Random()`. We currently consider the extra calls of `Random()` found by SEADER to be false positives, although the actual precision rate is higher.

Analysis of False Negatives. SEADER missed some labeled API misuses, because the corresponding misuse patterns are not covered by our 21 inferred patterns. Some of these missing patterns can be added to SEADER if we feed the tool with more code examples. One missing pattern cannot get added even if we provide $\langle I, S \rangle$ pairs to SEADER. The pattern is to replace `newURL("http://...")` with `newURL("https://...")`. HTTPS is HTTP with encryption. Nowadays all websites are supposed to use HTTPS instead of HTTP for secure communication, so any URL string hardcoded in programs should always start with "`https`" instead of "`http`". In this pattern, the difference between insecure and secure code lies in the string literal, which is not handled by SEADER currently. To derive the pattern from given code examples, we need to extend SEADER and our specialized ways of example definition, to accurately locate and properly represent any difference within strings.

Finding 2: *On the third-party Apache dataset,* SEADER *outperformed existing tools by achieving the highest precision, recall, and F-score on average. Our experiment indicates* SEADER*'s great capability of vulnerability detection.*

Table 4.5: The sampled vulnerabilities and repairs

Security Class API	# of Reports	Vulnerability Detection			Repair Suggestion	
		Basic	Intra-	Inter-	Parameter or API replacement	Code replacement
Cipher	6	5		1	6	
HostnameVerifier	2	2				2
IvParameterSpec	4		1	3		4
KeyPairGenerator	3		1	2	3	
KeyStore	5	1		4		5
MessageDigest	7	5		2	7	
PBEParameter-Spec	7	1	4	2	4	3
SecretKeyFactory	4	2		2	4	
SecretKeySpec	11	6	1	4	9	8
SecureRandom	5	5			5	
SSLContext	5	5			5	
TrustManager	12	12				12
Total	71	44	7	20	43	34

4.3.5 Effectiveness of Repair Suggestion

By applying Seader to the second dataset mentioned in Section 4.3.1.1, we got hundreds of vulnerabilities reported together with repair suggestions. Due to the time limit, we did not check every vulnerability as well as their repair(s); instead, we manually sampled the vulnerability reports and suggested repairs for 71 misuse instances.

To ensure the representativeness of our manual inspection results, we took four steps to create the sample set. First, we clustered all bug reports based on API-misuse patterns. Second, we randomly picked 10 reports from each cluster for further checking. If any cluster contained nine or fewer reports, we picked all reports. Third, when bug reports referred to duplicated code snippets, we removed duplicates to simplify our manual task, getting 71 sampled vulnerabilities. Fourth, we mapped the 71 samples to the security classes that they correspond to, and created Table 6. Notice that because some security classes (e.g., SecretKeySpec and TrustManager) have multiple method APIs that are prone to misuses

(i.e., have multiple API-misuse patterns), the corresponding rows contain more than 10 samples (e.g., 11 for SecurityKeySpec and 12 for TrustManager).

After the first author created the sample set, both the first and fifth authors independently checked bug reports and fixing suggestions. The two authors compared their manual inspection results for cross-checking; they had extensive discussion for any opinion divergence and even involved the second author into discussion, until reaching a consensus.

Among the examined vulnerabilities, SEADER revealed 44 misuses without doing any backward slicing (see the **Basic** column); it successfully matched templates with single Java statements. SEADER revealed seven misuses via intra-procedural slicing because in each of these scenarios, SEADER located and analyzed multiple statements to match the related template. SEADER revealed 20 misuses via inter-procedural slicing, because multiple statements from different Java entities (i.e., methods or fields) demonstrate each of the misuses.

70 of the vulnerabilities are true positives; the remaining one was falsely reported. This is because during its analysis, SEADER checks whether the second parameter of `KeyStore.load(InputStreamstream,char[]password)` is derived from a hardcoded constant; if so, the API call is considered insecure. Such analysis logic can effectively identify any password derived from a hardcoded secret. However, in our experiment, it incorrectly reported a scenario where the password is loaded from a file, whose name is hardcoded as a string literal. In the future, we will overcome this limitation by implementing heuristics (e.g., regular expressions) in SEADER, to differentiate between constants serving for distinct purposes.

Additionally, among the 77 suggested repairs, 43 repairs are solely about parameter/API replacement; SEADER does not need to generate any code or customize any identifier to propose these fixes. Meanwhile, 34 repairs involve both multi-statement fixes and identifier customization. Notice that the total number of repair suggestions (i.e., 77) is larger than

the vulnerability count (i.e., 71). This is because Seader provides multiple suggestions for six vulnerabilities, as each of the code snippets matches two templates simultaneously and Seader suggests a repair for each match. Finally, we found all repairs by Seader to be correct. However, as Seader has a false positive when reporting API misuses, we count the repairs for the other 70 misuses as correct suggestions.

Finding 3: *We manually checked 77 fixes generated by* Seader*, and found 76 of them to be correct. It indicates that* Seader *has great capability of repair suggestion.*

4.4 Threats to validity

All inferred patterns and detected vulnerabilities are limited to our experiment datasets and two cryptographic libraries: JCA and JSSE. The observations may not generalize well to other subject programs (e.g., closed-source projects) or other security libraries. We actually also manually checked API misuses in Spring Security—a widely-used third-party security framework, and found more misuse patterns that can be handled by Seader (e.g., the parameter value of BCryptPasswordEncoder's constructor should not be less than 10). Seader can handle the API misuses that involve (1) calling certain method APIs with incorrect parameter values, (2) calling certain method APIs in incorrect sequential orders, and (3) incorrectly overriding certain method APIs. Therefore, Seader is generalizable in terms of (1) the API misuse patterns to handle, and (2) security libraries to cover.

In some repair suggestions provided by Seader, there are placeholders that we need developers to further customize (see "`//Pleasechange`example.com'asneeded`" in Figure 4.7). Such placeholders should be filled based on developers' software environments, or even require extra configurations outside the codebase (e.g., generating and loading SSL certificates). In the future, we will provide clearer suggestions on hands-on experience and create

interactive tools that guide developers to apply complete repairs step-by-step.

4.5 Summary

This Chapter introduces SEADER—a new approach to take in ⟨insecure, secure⟩ code examples, infer vulnerability-repair patterns from examples, and apply those patterns for vulnerability detection and repair suggestion. Compared with prior work, SEADER offers a more powerful means for security experts to extend the pattern set of API-misuse detectors, and concretizes security expertise as customized fixing edits for developers. Our evaluation shows SEADER's great capabilities of pattern inference and application; it detects API misuses and suggests fixes with high accuracy.

In the next chapter, we will widen SEADER's applicability by specifying more code pairs in Spring framework. We will also extend SEADER's capability by adding support for Java annotations and configuration files. Moreover, we will provide the exploration demo of the detected vulnerabilities with customized repair suggestions to motivate developers to improve the code quality.

Chapter 5

An Empirical Study on ChatGPT-4.0 in Generating Security Tests

5.1 Introduction

A supply chain attack, also called a value-chain or third-party attack, occurs when hackers infiltrate software systems through an outside partner or provider with access to those systems and data [37]. When such attacks target open-source software libraries, cybercriminals may compromise those libraries to distribute malicious code through the software supply chain, or leverage the known vulnerabilities in existing libraries to compromise any systems built on top of those libraries. In 2021, supply chain attacks grew 650%, and caused 12,000 incidents [38]. In 2022, there was a 742% year-over-year increase in open-source software supply chain attacks, targeting vulnerabilities in upstream ecosystems such as JavaScript, Java, .NET, and Python [82]. Open Web Application Security Project (OWASP) listed "vulnerable and outdated software components" as the sixth top vulnerability in 2022 [73].

To mitigate supply chain attacks, researchers and engineers created a variety of tools to reveal vulnerable library dependencies in software applications [5, 10, 51, 55, 70, 76, 79, 157, 160], and even suggest fixes for those vulnerabilities [55, 70, 78, 147]. For instance, snyk-test [79] and npm-audit [70] are CLI commands that scan JavaScript (JS) applications for their package dependencies, compare those packages against the package lists in predefined

vulnerability databases (e.g., CVE), and report a vulnerability for each found match. Snyk-fix [78] is another CLI command, which eliminates vulnerabilities by automatically applying recommended updates to the vulnerable package dependencies revealed by snyk-test.

Unfortunately, developers often do not trust the vulnerabilities reported by existing tools, because none of these tools demonstrate how the found vulnerabilities lead to serious consequences (e.g., denial of service) in their own projects [124]. As described in an article well received by developers *npm audit: Broken by Design* [22], "In many situations, (npm-audit) leads to a 99%+ false positive rate, creates an incredibly confusing first programming experience, ..., and at some point will lead to actually bad vulnerabilities slipping in unnoticed." In recent studies [124, 178], **developers suggested that for reported vulnerable dependencies or vulnerable API usage, it is necessary to demonstrate how such vulnerabilities incur security attacks to their own systems**.

To (1) help developers mitigate the threats of supply chain attacks, and (2) persuade them into seriously considering the outputs of vulnerability detectors as well as fixers, we did a novel study to explore generating security tests for software applications (Apps) that depend on vulnerable libraries (Libs). These test cases demonstrate **proof-of-concept exploits**, which execute Apps in certain ways to (i) propagate vulnerabilities from Libs to Apps via calls of the APIs (i.e., functions) defined by Libs, and (ii) trigger abnormal behaviors of Apps such as throwing errors or becoming unresponsive to customers' normal requests. When developers run these test cases, they can observe vulnerability propagation paths, foresee the serious consequences due to hackers' successful attacks, assess the severity levels, and better decide whether to address those reported vulnerabilities.

The biggest challenge in generating security tests is how to ensure that the generated tests (1) execute vulnerable API calls made by Apps, (2) trigger the problematic behaviors of Apps, and (3) fail when reported vulnerabilities are not fixed. Iannone et al. [121] recently

created a tool SIEGE to generate security tests using EvoSuite [110]—the widely used test generation tool. Kang et al. [126] built TRANSFER, to generate tests by combining EvoSuite with program analysis techniques (e.g., call graph analysis and program instrumentation). Unfortunately, both tools fail to generate security tests most of the time. They often spend a lot of time producing irrelevant tests but cannot synthesize the specialized test inputs, code, or oracle to demonstrate proof-of-concept exploits. Based on our initial experience with ChatGPT [43]—an artificial intelligence chatbot developed by OpenAI, we found that ChatGPT is able to create programs to satisfy our software requirements described in English. Thus, in this project, we explored ChatGPT's capability in generating security tests for Apps with vulnerable dependencies.

We explored the following research questions (RQs) and observed interesting phenomena:

RQ1: *How effectively does ChatGPT-4.0 generate security tests?* We created a dataset to include (1) 25 Libs and (2) 55 Apps, with each App depending on a vulnerable library version. For each App, we offered ChatGPT-4.0 a prompt to describe the vulnerability, program context in App, and a security test from Lib showing a proof-of-concept exploit. With the prompt, we asked ChatGPT-4.0 to generate a test for App by mimicking the given test. We found that ChatGPT-4.0 generated tests for all 55 Apps, 24 of which effectively demonstrated the proof-of-concept exploits.

RQ2: *How does ChatGPT-4.0's security performance differ given various types of prompts?*

By changing the default design of our prompt template, we fed ChatGPT-4.0 with different subsets of the descriptive information in the above-mentioned prompts (see RQ1). We observed that all information elements provided important guidance to ChatGPT-4.0, while the security test from Lib was more important than the others. Without security test from Lib provided, none of the generated tests by ChatGPT-4.0 could successfully exploit any

vulnerability.

RQ3: *How does ChatGPT-4.0 compare with existing tools of security test generation?* We also applied two state-of-the-art tools—SIEGE and TRANSFER—to our dataset. Surprisingly, we observed ChatGPT-4.0 to outperform both tools. Given the 55 apps, ChatGPT-4.0 exploited the vulnerabilities in 24 apps; TRANSFER exploited 4 vulnerabilities; SIEGE exploited none.

5.2 A Motivating Example

To facilitate discussion, here we introduce a concrete example to show how vulnerabilities in Libs incur security attacks. Bouncy Castle (BC) is a collection of Java APIs used in cryptography [52]. According to CVE-2020-28052 [54], its releases 1.65 and 1.66 have a vulnerability: `OpenBSDBCrypt.checkPassword(...)` compares incorrect data when checking the password, allowing wrong passwords to be accepted as valid ones.

Listing 5.7 shows the security test defined by a BC version later than 1.66, which demonstrates the vulnerability and the proof-of-concept exploit. As shown in Listing 5.7, ideally, if a BC version has no vulnerability, the first assertion (line 10) should succeed as the first parameter `tokenString` was derived from the password `test-token`; the second assertion (line 11) should succeed as the first parameter `tokenString` was not from `wrong-token`. However, BC 1.65 and BC 1.66 fail the second assertion, as the invalid password `wrong-token` is wrongly considered to match `tokenString`. Such a security test demonstrates the problematic behaviors of vulnerable library versions, and implies the potential of security exploits (e.g., sending in wrong passwords to pass identity authentication).

Although Listing 5.7 shows a proof-of-concept exploit of the library vulnerability, it does

```
1  public void performTest() throws Exception {
2    ... ...
3    int costFactor = 4;
4    SecureRandom random = new SecureRandom();
5    salt = new byte[16];
6    for (int i = 0; i < 1000; i++) {
7      random.nextBytes(salt);
8      final String tokenString = OpenBSDBCrypt.
9          generate("test-token".toCharArray(), salt, costFactor);
10     isTrue(OpenBSDBCrypt.checkPassword(tokenString, "test-token".toCharArray()));
11     isTrue(!OpenBSDBCrypt.checkPassword(tokenString, "wrong-token".toCharArray()));
12 }
```

Listing 5.7: A security test to demonstrate the proof-of-concept exploit of the vulnerability in BC 1.65 & 1.66

not show how Apps built on top of vulnerable BC versions can behave abnormally or get attacked due to that vulnerability. In the scenarios when an App does not call `OpenBSDBCrypt.checkPassword(...)`, the App is not influenced by the vulnerability at all; consequently, even if existing dependency checkers (e.g., OWASP Dependency-Check [10]) report a vulnerable dependency for App, the report is not useful but can mislead or irritate developers.

Our research intends to generate security test cases for Apps, in order to demonstrate (1) how vulnerabilities get propagated from Libs to Apps, and (2) how the resulting vulnerabilities in Apps can be leveraged by hackers.

5.3 Methodology

We did an empirical study on ChatGPT's capability of generating security test cases, for Apps which have dependencies on vulnerable Libs and call vulnerable APIs. Our study has three phases: dataset construction (Section 5.3.1), prompt design (Section 5.3.2), and

result validation (Section 5.3.3). Phase I collects known vulnerabilities in Libs, exemplar tests that exploit vulnerabilities, and Apps that can be affected by their usage of vulnerable APIs. Phase II adopts the information collected by Phase I, formulates a variety of prompts for individual $\langle Lib, App \rangle$ pairs, and sends prompts to ChatGPT-4.0. These prompts ask ChatGPT-4.0 to leverage all information provided, and to generate security tests that demonstrate proof-of-concept exploits. Phase III gathers all outputs by ChatGPT-4.0 if there is any, assesses the quality of generated tests, and evaluates ChatGPT-4.0's capability accordingly.

5.3.1 Phase I: Dataset Construction

We took two steps to create a dataset: (1) finding exploitable vulnerabilities, (2) getting vulnerable Libs and dependent client Apps.

5.3.1.1 Locating Exploitable Vulnerabilities

As vulnerabilities sparsely exist in software libraries, it can be very inefficient to blindly mine GitHub repositories for vulnerabilities. Thus, we started with the datasets mentioned by prior work [126, 156] and initiated our exploration with 628 entries. Each entry is linked to a CVE entry or JIRA issue to describe a vulnerability in a Java library *Lib*, and a GitHub repository that shows both the vulnerable and patched versions of *Lib*.

We manually inspected all available data for each entry, to decide what security test was added in the patched version, what APIs were revised by the patched version, and how the security test(s) as well as revised APIs are relevant to the described vulnerability. Because vulnerability description is not always precise or comprehensive in pinpointing the vulnerable APIs in Libs, we spent lots of time understanding the program context to locate vulnerable

APIs. We consider an API in *Lib* to be vulnerable if it (1) is mentioned or implied by the vulnerability description of CVE or JIRA entry and is revised, (2) directly or indirectly calls the described vulnerable API, (3) is invoked by the described API and is the root cause for the described vulnerability, or (4) shares the same root cause with the described API (i.e., they both call the same root-cause vulnerable method). Afterwards, we chose vulnerability entries based on the following criteria:

(a) The Java library *Lib* has at least one JUnit test from the patched version, to demonstrate behavioral differences between the vulnerable and patched versions.

(b) *Lib* should compile successfully.

(c) The test execution does not require for complex setups of client/server machines.

Criterion (a) ensures the exploitability of confirmed vulnerabilities. Namely, if no security test is included for a given vulnerability, it is hard for us to manually craft or justify the ground truth of proof-of-concept exploit. Criteria (b) and (c) ensure that we can run the security test defined for *Lib*, to observe the behavioral differences between vulnerable and patched versions. In our study, we treated the three criteria mentioned above as filters, and applied them one-by-one to refine initial vulnerability datasets. In particular, Criterion (a) removed most entries (i.e., 427) from the original datasets: 304 entries were removed because Java libraries have no test, and 123 entries were removed because the tests are irrelevant to vulnerabilities. Criterion (b) eliminated 49 entries, and (c) removed additional 107 entries. At the end of this step, we found 45 unique entries to match all criteria mentioned above.

5.3.1.2 Collecting Libs and Apps for Vulnerabilities

For each *Lib* mentioned in the 45 refined vulnerability entries (see Section 5.3.1.1), we searched on GitHub for client Apps depending on *Lib*, by using the library or package name as keyword(s). As GitHub typically retrieved lots of projects for each of our keyword-based search requests, we could not afford the time or effort of examining all projects. Thus, for each request, we limited ourselves to inspect the first 10 pages of results returned by GitHub, aiming at finding up to 4 client applications to satisfy the criteria below:

(e) At least one non-private Java method (not test) in *App* (in)directly calls vulnerable API(s) in *Lib*, with non-constant parameters.

(f) *App* compiles and runs successfully.

Criterion (e) ensures the feasibility of security exploits. Basically, if users of *App* craft malicious input values to feed certain public or protected method(s) in *App* (i.e., the callers of vulnerable APIs), they can run vulnerable APIs in malicious ways and thus realize attacks. Criterion (f) ensures that we can check the correctness of tool-generated tests via compilation and program execution.

Table 5.1: The library vulnerabilities and client applications included in our dataset

Category	**Vulnerability Entry ID**	**Library**	**Affected Library Versions**	**Vulnerable API(s) & Potential Exploit**	**# of Apps**
	CVE-2017-7957	XStream [85]	[, 1.4.9]	`XStream.fromXML(...)` mishandles attempts to create an instance of the primitive type "`void`" during unmarshalling, leading to a remote application crash, i.e., denial of service (**DoS**).	2
	CVE-2018-1000873	Jackson-Modules-Java8 [64]	[, 2.9.8)	`ObjectMapper.readValue(...)` triggers DoS when it deserializes a very large decimal value to time.	4

Denial of Service (13)	CVE-2018-11761	Apache Tika [83]	[0.1, 1.18]	`SAXParser.parse(...)` was not configured to limit entity expansion, and thus could lead to DoS.	1
	CVE-2018-12418	Junrar [68]	[,1.0.1)	The `Archive` constructor gets into an infinite loop when handling corrupt RAR files.	1
	CVE-2018-1274	Spring Data Commons [80]	[1.13, 1.13.10], [2.0, 2.0.5]	`PropertyPath.from(...)` allocates resource without limits, and thus can cause DoS due to its consumption of CPU and memory.	1
	CVE-2019-10093	Apache Tika	[1.19, 1.21]	`Parser.parse(...)` enables a carefully crafted 2003ml or 2006ml file to consume all available SAXParsers in the pool.	3 (2*)
	CVE-2019-12402	Apache Commons Compress [48]	[1.15, 1.18]	Malicious inputs to `ZipArchiveOutputStream.putArchiveEntry(...)` or `ZipEncoding.encode(...)` can cause infinite loops.	1
	CVE-2020-28491	Jackson Dataformat: CBOR [63]	[, 2.11.4), (2.12.0-rc1, 2.12.1)	`ObjectMapper.createParser(...)` allocates resources without limits; it can cause `java.lang.OutOfMemoryError`.	1
	CVE-2021-27568	Json-smart [66, 67]	v1:[, 1.3.2), v2: [, 2.3.1), [2.4, 2.4.1)	`JSONParser.parse(...)` throws an uncaught exception, which can cause an application crash or expose sensitive information.	2
	CVE-2021-30468	Apache CXF [50]	[, 3.3.11), [3.4.0, 3.4.4)	Malicious inputs to `JsonMapObjectReaderWriter.fromJson(...)` or `JsonMapObjectReaderWriter.fromJsonToJsonObject(...)` can result in an infinite loop.	1
	CVE-2022-45688	JSON-java(i.e., hutool-json) [60]	[, 20230227)	Malicious inputs to `XML.toJSONObject(...)` or `JSONML.toJSONObject(...)` can trigger DoS.	3
	TwelveMonkeys-595	TwelveMonkeys [84]	[0, 3.6.4)	A corrupt JPEG file to `ImageReader.read(...)` can cause DoS.	2 (1*)
	Zip4j-263	Zip4j [86]	[0, 2.7.0)	The `ZipFile(...)` constructor can take in a null File reference, which later produces a null pointer exception.	2
	CVE-2018-1002200	Plexus Archiver [74]	[,3.6.0)	`UnArchiver.extract(...)`, `ZipUnArchiver.extract(...)`, and `TarGZipUnArchiver.extract(...)` allow attackers to write to arbitrary files via "../" in an archive entry (**Zip Slip**).	3

Directory Traversal (6)	CVE-2018-1002201	ZT Zip [87]	[, 1.13)	`ZipUtil.unpack(...)` allows attackers to write to arbitrary files via archive extraction (Zip Slip).	1
	CVE-2018-19859	OpenRefine [44]	[, 3.2-beta)	`ImportingUtilities.allocateFile(...)` allows arbitrary file write via archive extraction (Zip Slip).	1(1*)
	CVE-2018-20227	RDF4J [75]	2.4.2	`ZipUtil.extract(...)` enables arbitrary file write via archive extraction (Zip Slip).	1
	CVE-2021-29425	Apache Commons IO [49]	[, 2.7)	`FileNameUtils.normalize(...)` enables **directory traversal**, which provides access to files beyond the target file location.	3
	HTTPCLIENT-1803	Apache HttpClient	[,4.5.3)	The `URIBuider` constructor, `URIBuilder.setHost(...)`, `URIBuilder.build(...)`, and `URIBuilder.toString(...)` can result in directory traversal.	1
Remote Code Execution (4)	CVE-2017-7525	Jackson Databind [62]	[, 2.6.7.1) [2.7.0, 2.7.9.1) [2.8.0, 2.8.9)	A deserialization flaw in the library allows maliciously crafted inputs to `ObjectMapper.readValue(...)` to trigger remote code execution.	2
	CVE-2020-26217	XStream	[, 1.4.14)	Malicious inputs to `XStream.fromXML(...)` allow attackers to run arbitrary shell commands.	3
	CVE-2021-23899	OWASP JSON Sanitizer	[,1.2.2)	`JsonSanitizer.sanitize(...)` may allow hackers to inject arbitrary code into embedding documents.	1
	CVE-2022-25845	Fastjson [57]	[, 1.2.83)	`JSON.parseObject(...)` may deserialize untrusted data, allowing hackers to attack remote servers.	2
	CODEC-134	Apache Commons Codec [47]	[, 1.13)	Malicious inputs to `Base64.decodeBase64(...)` or `Base64.decode(...)` can realize **covert channel** [177], which creates a capability of transferring data between processes that should not communicate	3
	CVE-2018-1000632	Dom4j [56]	[, 2.1.1)	Malicious inputs to `DocumentHelper.createElement(...)` or `Branch.addElement(...)` can result in **XML injection**, which tampers with XML documents.	3
	CVE-2020-13956	Apache HttpClient [59]	[, 4.5.13,), [5.0.0, 5.0.3)	Malicious inputs to `CloseableHttpClient.execute(...)` or `URIUtils.extractHost(...)` trigger **Blind Server-Side Request Forgery (SSRF)**, which attack induces an application to issue a back-end HTTP request to a supplied URL, but the response from the back-end request is not returned to the application's front-end response.	1

Others (7)	CVE-2020-13973	OWASP JSON Sanitizer [65]	[,1.2.1)	`JsonSanitizer.sanitize(...)` does not properly escape disallowed characters, and thus facilitates **cross-site scripting (XSS)**, which enables the browser to unknowingly execute malicious script on the client side and perform actions that are otherwise blocked by the browser's Same Origin Policy.	1
	CVE-2020-28052	Bouncy Castle	1.65, 1.66	`OpenBSDBCrypt.checkPassword(...)` improperly verifies passwords, allowing wrong ones to be accepted as valid ones.	2 (1*)
	CVE-2020-5408	Spring Security [81]	[4.2.0, 4.2.16), [5.0.0, 5.0.16), [5.1.0, 5.1.10), [5.2.0, 5.2.4), [5.3.0, 5.3.2)	`BCryptPasswordEncoder.encode(...)` presents cryptographic weakness, which may allow hackers to decrypt encrypted messages via a dictionary attack.	2 (2*)
	CVE-2023-34454	snappy-java [77]	[, 1.1.10.1)	`Snappy.compress(...)` improperly validates array length, and may cause Access Violation errors.	1

* indicates the number of clients with injected vulnerable dependencies.

Such a manual crawling process removed 15 from the 45 entries mentioned above, because we found no client project to satisfy both criteria for those entries. As shown in Table 5.1, our dataset includes 30 vulnerability entries, corresponding to 25 unique libraries. These libraries cover various domains, such as data processing (e.g., Apache Commons Codec [47]), web development (e.g., Apache CXF [50]), and security (e.g., Spring Security [81]). Most libraries have single vulnerabilities (e.g., Dom4j [56], while a few libraries have multiple (e.g., XStream [85]). In Table 5.1, We identified 4 major categories among the 30 vulnerabilities: denial of service, directory traversal, remote code execution, and others. **Affected Library Versions** shows the vulnerable library versions described by each CVE entry or JIRA issue. **Vulnerable API(s) & Potential Exploit** shows the vulnerable APIs and security consequence we summarized by inspecting all relevant data.

According to our experience, it is very challenging to crawl for sufficient Apps satisfying (e)–(f) for any vulnerable library. To conduct a representative empirical study with sufficient data points, we had to fully leverage the retrieved Apps even though they do not depend on vulnerable library versions. Specifically for each found project *App* satisfying criteria (e)–(f), we examined the version history to reveal any version of *App*—e.g., App_i—that depends on a vulnerable library version. If App_i exists, we checked out App_i to prepare for ChatGPT-4.0 usage. Otherwise, we manually revised dependency to inject the vulnerability. For instance, OpenRefine [44] is a library whose versions before 3.2-beta suffer from CVE-2018-19859. However, we only found one client project for it, which depends on a safe version of OpenRefine (i.e., 3.3). To make sure that this client is still usable in our study, we downgraded the library dependency to 3.1 and tried to compile the revised project.

Our manual revision of dependencies does not compromise the validity of our research, as we fairly compared all approaches of security test generation on the same set of client Apps, no matter whether their vulnerable dependencies are real or injected. The last column in Table 5.1 shows the number of clients we included for each Lib. Our dataset includes in total seven projects with injected vulnerable dependencies, all of which are marked with asterisks (*) in the table.

Notice that we do not expect ChatGPT-4.0 to automatically detect vulnerable dependency or the usage of vulnerable APIs. Instead, we assume some tools or domain experts to detect vulnerabilities, and provide all relevant data to ChatGPT-4.0 for exploit generation. Our experiment simulates such a usage scenario of ChatGPT-4.0, and our dataset has the data below for each exploit generation task:

(i) A GitHub project or software application *App*.

(ii) A vulnerable version of library *Lib* on which *App* depends.

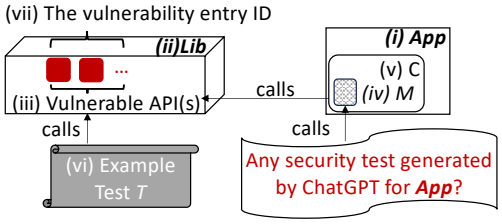

Figure 5.1: Our default usage of ChatGPT to generate security tests

Prompt Design

Write a JUnit test for function: *[function-name]* in "CLIENT CODE". The test should verify whether the function *[function-name]* is affected by the vulnerability: [*vulnerability-ID*], or affected by vulnerable APIs: [*vulnerable-API-list*]. Mimic the test function "TEST FUNCTION:" *[test-function-name]* I provide to generate the input in your test. For the provided code "CLIENT CODE", mock the classes and functions that are not Java SE APIs and have no definition provided.
TEST FUNCTION:
``` *Java*
*[exemplar-test-impl]*
```
CLIENT CODE:
``` *Java*
*[client-code-impl]*
```

Figure 5.2: Our prompt template

(iii) The vulnerable library API(s) called by *App*.

(iv) The non-private method M inside *App* that (in)directly calls vulnerable APIs.

(v) The Java class C defining method M.

(vi) The exemplar security test case T (i.e., a Java method) from *Lib*. If it indirectly calls vulnerable APIs, the definition of all methods standing between T and APIs is also included.

(vii) The vulnerability entry ID (i.e., CVE ID or bug issue ID).

Fig. 5.1 illustrates the relations between information items (i)–(vii) mentioned above. Basically, a vulnerable version of the library *Lib* can include one or more vulnerable APIs; such a vulnerability is explained in detail by a CVE report or bug issue, which can be uniquely referenced or identified via a vulnerability entry ID. A later version of *Lib* may fix the vulnerability, and define an exemplar security test T to provide maliciously crafted inputs as it calls vulnerable API(s). In this way, T can demonstrate the behavioral differences between the vulnerable and fixed versions, or show a proof-of-concept exploit of the known vulnerability. Meanwhile, the application *App* depends the vulnerable library version and defines a method $C.M(...)$ to call the vulnerable API(s).

At the end of Phase I, our dataset consists of all relevant information for 55 $\langle App, Lib \rangle$ pairs. We leveraged that information to define 55 exploit generation tasks, which were used in later phases.

5.3.2 Phase II: Prompt Design

Based on the information collected in Phase I, we formulated prompts, and sent those prompts to ChatGPT-4.0 to generate exploits for Apps with vulnerable dependencies. This section introduces both our default template design for prompts (Section 5.3.2.1) and alternative designs (Section 5.3.2.2).

5.3.2.1 The Design of Our Default Prompt Template

As shown in Fig. 5.1, we typically used ChatGPT-4.0 to generate a security test for any given $\langle App, Lib \rangle$ pair, by providing as much relevant information as possible in the prompts. We intended to check whether ChatGPT-4.0 can (1) mimic the exemplar test T, and (2) similarly craft malicious inputs to M so that M also calls library API(s) with malicious inputs. To do that, we defined a prompt template that can be concretized with the information (iii)–(vii) mentioned in Section 5.3.1.2. As shown in Fig. 5.2, the template variable *[function-name]* is the name of M (see **(iv)**); *[vulnerability-ID]* refers to **(vii)**, *[vulnerable-API-List]* refers to **(iii)**; *[test-function-name]* is the name of example test (see **(vi)**); *[exemplar-test-impl]* refers to **(vi)**; and *[client-code-impl]* refers to **(v)**. Using this template, we generated 55 prompts for our dataset, and sent all prompts to ChatGPT-4.0.

For the motivating example described in Section 5.2, we actually found a GitHub project depending on Bouncy Castle. As shown in Listing 5.8, the project defines a method `BcryptPasswordHashFunction.check(...)` to directly call vulnerable API `OpenBSDBCrypt.`

```java
public class BcryptPasswordHashFunction implements PasswordHashFunction {
  ...
  @Override
  public boolean check(String passwordHash, String password) {
    // the vulnerable API call
    return OpenBSDBCrypt.checkPassword(passwordHash, password.toCharArray());
  }
  ...
}
```

Listing 5.8: A GitHub project defines a Java file to call OpenBSDBCrypt.checkPassword()

`checkPassword(...)`. Although both the API and its caller take in two parameters, the caller method `check(...)` has to convert its second parameter `password` before calling that API. To generate a proof-of-concept exploit for the client method `BcryptPasswordHashFunction.check(...)`, we formulated the prompt shown in Fig. 5.3 by customizing our template. This figure only shows partial code of example test and client class to simplify our presentation, while the actual prompt we sent to ChatGPT-4.0 includes the complete code.

Write a JUnit test for function: check in "CLIENT CODE". The test should verify whether the function check is affected by the vulnerability: CVE-2020-28052, or affected by vulnerable APIs: [OpenBSDBZCrypt.checkPassword]. Mimic the function "TEST FUNCTION:" performTest I provide to generate the input in your test. For the provided code "CLIENT CODE", mock the classes and functions that are not Java SE APIs and have no definition provided.
TEST FUNCTION:
``` Java
@Test
public void performTest() throws Exception {
  ... // Here we omit details of BC security test (Listing 1) for brevity.
}
```
CLIENT CODE:
``` java
public class BcryptPasswordHashFunction implements PasswordHashFunction {
  // Here we omit details of BcryptPasswordHashFunction.java, which defines the method `check(…)` in Listing 2
  ...
}
```

Figure 5.3: A prompt derived from our template

Given the prompt, ChatGPT-4.0 successfully generated an executable security test as requested. Listing 5.9 shows a brief version of the generated code: a class named

`BcryptPasswordHashFunctionTest` defines a test function `testCheckFunction()`, to call the vulnerable API with appropriate formats of the seed inputs of exemplar test. The generated test is very similar to the exemplar test, but it shows the vulnerability exploit for client code.

```
... // We omit less important details for brevity
public class BcryptPasswordHashFunctionTest {
  ...
  @Test
  public void testCheckFunction() {
    int costFactor = 4;

    for (int i = 0; i < 1000; i++) {
      random.nextBytes(salt);
      final String tokenString = OpenBSDBCrypt.generate("test-token".toCharArray(), salt, costFactor);
      assertTrue(bcryptPasswordHashFunction.check(tokenString, "test-token"));
      assertFalse(bcryptPasswordHashFunction.check(tokenString, "wrong-token"));
    }
  }
}
```

Listing 5.9: A brief and commented version of the security test successfully generated by ChatGPT

Table 5.2: The six prompt templates we explored

Id	Prompt Template	Id	Prompt Template
P	Default (including all elements iii-vii)	P_3	Without C (v)
P_1	Without vulnerable APIs (iii)	P_4	Without the exemplar test (vi)
P_2	Without M (iv)	P_5	Without the vulnerability ID (vii)

5.3.2.2 The Design of Alternative Prompt Templates

In addition to the default prompt template, we also defined five variant templates by removing a single element from (iii)–(vii) each time. In this way, we can explore how different information elements contribute to ChatGPT-4.0's effectiveness. As shown in Table 5.2, to facilitate presentation, we use P to simply refer to our default template design, which

leverages the information elements (iii)–(vii). We use P_1–P_5 to separately refer to the five template variants, each of which contains one fewer item than P. For instance, P_1 leaves out (iii), but defines four variables to take in (iv)–(viii). P_5 leaves out (vii), but takes in (iii)–(vi). We then generated 55 prompts using each variant template, and sent all prompts to ChatGPT-4.0.

5.3.3 Phase III: Result Validation

After sending all prompts to ChatGPT-4.0, we gathered and recorded ChatGPT-4.0's outputs if there is any. For each test class T' generated by ChatGPT-4.0, we copied the code to its corresponding software project *App*. If T' has a unique name and defines one or more test functions for method-to-test M, we put the class into the test folder of *App*. Otherwise, if T' has the same name as an existing test class, we copied the relevant non-conflicting content and appended it to that class; if T' defines test functions for a non-public method M, we copied the code and pasted it to any existing (test) class where M is accessible. To sum up, we integrated the generated tests into client projects, ensuring that the tests were put at the right places.

For each generated test, we compiled the whole project *App* to examine for compilation errors; if some compilation errors were obvious and easy to fix (e.g., missing/wrong package names or missing library dependencies), we fixed those errors manually to explore whether ChatGPT-4.0 was able to synthesize the most important logic of security tests successfully. After one or multiple iterations of the compilation-and-fixing procedure, if a test compiled successfully, we further executed the whole project *App* with that test to observe runtime behaviors. If any exception or runtime error was thrown, we studied the exception/error message, inspected the intermediate program status via step-by-step debugging, and discussed the relevance

with vulnerability exploitation among authors until reaching a consensus.

5.4 Experiments and Results

We did experiments for the following research questions (RQs):

RQ1: *How effectively does ChatGPT-4.0 generate security tests?* We investigated the strengths and weaknesses of ChatGPT-4.0 when it generates proof-of-concept exploits for known vulnerabilities.

RQ2: *How does ChatGPT's security performance differ given various types of prompts?* Among the various information we provided to ChatGPT-4.0 (see (iii)–(vii) mentioned in Section 5.3.1.2), we wanted to learn which type of information is more crucial, and how each information type contributes to ChatGPT-4.0's capability of security test generation.

RQ3: *How does ChatGPT-4.0 compare with existing tools of security test generation?* This RQ explores whether it is worth the effort of trying to generate security tests using large language models (LLMs), instead of using program analysis techniques.

This section first introduces the metrics we defined to assess tools of security test generation (Section 5.4.1). It then explains our experiments and results for the RQs (Sections 5.4.2–5.4.4).

5.4.1 Metrics

There are three metrics used in our experiments:

Table 5.3: Security test generation by ChatGPT-4.0 (Total: 55 A, 40 C, 24 V)

Idx	Vulnerability Entry ID	# of Clients	A	C	V	Idx	Vulnerability Entry ID	# of Clients	A	C	V
1	CODEC-134	3	3	2	2	16	CVE-2020-13973	1	1	0	0
2	CVE-2017-7525	2	2	2	2	17	CVE-2020-26217	3	3	3	2
3	CVE-2017-7957	2	2	1	1	18	CVE-2020-28052	2	2	2	1
4	CVE-2018-1000632	3	3	1	1	19	CVE-2020-28491	1	1	1	0
5	CVE-2018-1000873	4	4	2	1	20	CVE-2020-5408	2	2	2	2
6	CVE-2018-1002200	3	3	1	0	21	CVE-2021-23899	1	1	1	0
7	CVE-2018-1002201	1	1	1	1	22	CVE-2021-27568	2	2	2	2
8	CVE-2018-11761	1	1	0	0	23	CVE-2021-29425	3	3	2	2
9	CVE-2018-12418	1	1	0	0	24	CVE-2021-30468	1	1	1	0
10	CVE-2018-1274	1	1	0	0	25	CVE-2022-25845	2	2	2	2
11	CVE-2018-19859	1	1	1	0	26	CVE-2022-45688	3	3	3	1
12	CVE-2018-20227	1	1	1	0	27	CVE-2023-34454	1	1	1	1
13	CVE-2019-10093	3	3	2	0	28	HTTPCLIENT-1803	1	1	1	0
14	CVE-2019-12402	1	1	0	0	29	TwelveMonkeys-595	2	2	2	0
15	CVE-2020-13956	1	1	1	1	30	Zip4j-263	2	2	2	2

Tool Applicability (A) counts for how many $\langle App, Lib \rangle$ pairs, a tool generates a security test.

Test Compilability (C) counts the number of generated tests that are compilable.

Vulnerability Exploitation (V) counts the number of compilable tests that exploit the known vulnerabilities as expected.

5.4.2 ChatGPT-4.0's Effectiveness in Security Test Generation

We created 55 prompts using the default prompt template P (see Section 5.3.2), and sent them to ChatGPT-4.0 for results. This experiment examines "*When we provide as much relevant information as possible in the prompts sent to ChatGPT-4.0, how effectively can it generate security tests?*" As shown in Table 5.3, ChatGPT-4.0 generated tests for all prompts: 26 of the tests are compilable and runnable; 14 tests are compilable and runnable after we manually applied minor fixes (e.g., customizing hardcoded file paths); 24 of these 40 tests (26 + 14) effectively mimic the behaviors of given library tests, and successfully exploit vulnerabilities by throwing relevant errors or runtime exceptions.

5.4.2.1 Uncompilable Tests.

Among the 55 generated tests, 15 tests do not compile and cannot get easily fixed via minor changes. Six tests do not compile as they violate Java access rules. For instance, four of the tests access private members from outside of the classes (i.e., fields and methods); one test directly references a method defined inside an enum from outside; one test has an ambiguous method reference, which can be interpreted as a call to either of two same-named methods. Another 5 of the 15 tests use undefined program entities such as methods and classes. Finally, 4 of the 15 tests call methods with inappropriate parameter lists, by missing some parameters or using wrongly typed parameters. Our observations imply that ChatGPT-4.0 does not guarantee code compilation, even though the majority of tests it generated (40/55) are easy to compile.

5.4.2.2 Ineffective or Less Effective Tests.

Sixteen generated tests do not trigger vulnerabilities as expected. They either throw exceptions/errors other than the expected ones, or throw no exception/error at all. Four reasons can explain such ineffectiveness.

1. Mockito [69]—a mocking framework—was frequently used by ChatGPT-4.0 to mock unknown variables, methods, or classes when generating tests. Unfortunately, the framework could not mock everything (e.g., final classes). As a result, it led to MockitoExceptions thrown by some of the tests.

2. In generated tests, the parameters passed to the method-to-test M are not always well prepared. They may be malformed, include `null`-values in critical fields, or fail to contain the essential values to trigger vulnerabilities.

3. Some client projects already applied patches in response to the known library vulnerabilities. Thus, even though the generated tests try to trigger those vulnerabilities, the client projects screen out those trials and prevent library vulnerabilities from being exploited.

4. A different bug in the project anet [46] was revealed by a generated test. This bug causes a runtime exception and prevents the test from further execution to trigger the known vulnerability CVE-2021-23899. We actually filed an issue for the newly revealed bug, and developers of that project confirmed our bug report.

5.4.2.3 Effective Tests.

Twenty-four generated tests can trigger vulnerabilities as expected. For all of these tests, ChatGPT-4.0 successfully extracted vulnerability-triggering inputs from the exemplar tests, reused those inputs in test generation, prepared meaningful values for parameters to call method M, and threw exactly the expected exceptions/errors or presented relevant abnormal program behaviors (e.g., infinite loop and timeout). Based on our experience, when M calls vulnerable API(s) directly, requires for very few and simple parameters, and has simple program logic without many conditional or loop statements, ChatGPT-4.0 was more likely to generate security tests successfully. Meanwhile, when M calls vulnerable API(s) indirectly, requires for many or complex parameters, or has complex program logic with a lot of code, ChatGPT-4.0 was unlikely to generate good tests.

Finding 1: *ChatGPT-4.0 is promising in generating security tests for known library vulnerabilities. Given 55 test generation tasks, it produced 55 tests, 40 of which are easy to compile and 24 of the tests successfully exploited vulnerabilities.*

5.4.3 Information Elements That May Impact ChatGPT-4.0's Effectiveness

We used the 5 template variants P_1–P_5 mentioned in Section 5.3.2, generated prompts for the 55 $\langle App, Lib \rangle$ pairs in our dataset, and sent them to ChatGPT-4.0 for results.

Table 5.4: The comparison of applicability and compilability between tests generated in different ways

Id	Prompt Template	Tool Applicability (A)	Test Compilable?			
			Yes (C)	No		
				Access Rule Violation	Incorrect Method Calls	Unknown Entity Usage
P	Default (all elements)	55	40	6	4	5
P_1	Without vulnerable APIs (iii)	55	39	3	2	11
P_2	Without M (iv)	55	42	5	1	7
P_3	Without C (v)	55	16	3	1	35
P_4	Without the exemplar test (vi)	55	32	7	2	14
P_5	Without the vulnerability ID (vii)	55	40	8	0	7

5.4.3.1 ChatGPT-4.0's Applicability Given Divergent Types of Prompts

As shown in Table 5.4, no matter what information item in the default template was removed, the resulting prompts always guided ChatGPT-4.0 to produce tests. It means that ChatGPT-4.0 has great applicability: it is always applicable no matter how the prompts were formulated.

5.4.3.2 ChatGPT-4.0's Test Compilability Given Divergent Types of Prompts

The number of compilable tests for each prompt template varies a lot. As shown in Table 5.4, when P_2 was used and M was not specified, ChatGPT-4.0 generated more compilable tests than what it did for the default template P (42 vs. 40). When P_5 was used and no vulnerability ID was mentioned, ChatGPT-4.0 generated the same number of compilable tests as what it did for P—40. When P_1 was used and the vulnerable APIs were not specified,

ChatGPT-4.0 generated slightly fewer compilable tests—39. However, when P_3 and P_4 were used, a lot fewer generated tests compile, i.e., 16 and 32. One possible reason is that both P_3 and P_4 significantly removed the code context relevant to test generation, while the other templates removed almost no code context. As a generative AI tool, ChatGPT-4.0 predicts the next word(s) given a data sequence, by using (1) an encoder to process the input sequence and (2) a decoder to generate the output [61]. It tended to generate more compilable tests when more relevant program context was provided.

Among the six prompt templates, P_3 caused ChatGPT-4.0 to create the most uncompilabe tests—39; 35 of these tests fail compilation due to their usage of unknown entities. This may be because P_3 does not specify the Java class C holding the function-to-test; ChatGPT-4.0 could not identify many valid or usable entities available in the software projects, so it usually refers to some nonexistent entities in the produced tests. In contrast, P_2 caused ChatGPT-4.0 to create the fewest uncompilable tests—13, only 7 of which fail compilation due to their usage of unknown entities. This comparison implies that ChatGPT-4.0 could produce more compilable tests when (1) more program context is provided in prompts, and (2) there is no constraint on what method to test.

No matter what template we tried, ChatGPT-4.0 always produced uncompilable tests for some prompts. This indicates that ChatGPT-4.0 does not strictly follow the rules of Java program syntax and semantics. As a general AI model, it is trained on a massive dataset of text from the Internet; it leverages its training to predict what words and phrases are likely to come next in a given context. Thus, the generated code may violate access rules, call methods with inappropriate parameter lists, or use unknown program entities. This observation implies the necessity of applying sanity checks to ChatGPT-4.0-generated code and fixing any revealed bugs, to ensure the program quality.

Finding 2: *Among the five template variants we explored, P_3 and P_4 caused ChatGPT-4.0 to work considerably worse in producing compilable tests. ChatGPT-4.0 tended to produce more compilable tests, given more contextual code and fewer constraints relevant to the test-generation tasks.*

Table 5.5: The comparison of vulnerability exploitation between tests generated in different ways

Idx	Prompt Template	Vulnerability Exploited?			
		Yes (V)	No		
			No error/exception	Mockito exception	Other exceptions/errors
P	Default (all elements)	24	4	2	10
P_1	Without vulnerable APIs (iii)	15	6	8	10
P_2	Without M (iv)	14	4	8	16
P_3	Without C (v)	1	7	5	3
P_4	Without the exemplar test (vi)	0	11	11	10
P_5	Without the vulnerability ID (vii)	14	5	5	16

5.4.3.3 ChatGPT-4.0's Vulnerability Exploitation Given Divergent Types of Prompts

As shown in Table 5.5, removing any element from P made ChatGPT-4.0 exploit vulnerabilities less effectively. Specifically, among the five variants, P_4 caused ChatGPT-4.0 to work worst, producing zero successful vulnerability exploit. This implies that exemplar tests offer (1) crucially important demonstration for potential exploit methods, and (2) essential hints for ChatGPT-4.0 to formulate vulnerability-triggering input values. Without such information, ChatGPT-4.0 generated 11 tests throwing no error/exception, 11 tests wrongly mocking program entities, and 10 tests triggering irrelevant errors or exceptions. Although slightly better than P_4, P_3 also worsened ChatGPT-4.0 significantly and led it to generate only one exploit successfully. This may be because the removed Java class C holds lots of program context information, whose absence caused ChatGPT-4.0 to produce tests in a context-insensitive way, making the produced tests irrelevant or invalid.

P_1, P_2, and P_5 had very similar effects on ChatGPT-4.0, as the tool produced 15, 14, and

14 successful exploits given the prompts derived from each of them. All these numbers are much lower than the number reported for the default template P: 24. This implies that the elements removed by individual templates (iii, iv, vii) provide valuable signals to ChatGPT-4.0, to help it identify and focus on the vulnerable APIs, function-to-test, and specialized vulnerability. While (v) and (vi) provide as much relevant code as possible for ChatGPT-4.0 to refer to, the other elements (iii, iv, vii) guide ChatGPT-4.0 to pay special attention to the most important content in the relevant code.

Finding 3: *Among the five information elements covered by the default prompt template P, all elements played an important role to help ChatGPT-4.0 effectively generate vulnerability exploits. In particular, (v) and (vi) were more important than (iii), (iv), and (vii).*

5.4.4 Tool Comparison

Two tools were recently proposed to automatically generate security tests: SIEGE [121] and TRANSFER [126]. SIEGE adopts a genetic algorithm (GA). For any $\langle App, Lib \rangle$ pair, it requires users to describe the search target (i.e., the coverage goal for tests-to-generate), including (1) fully qualified name of the target vulnerable class in *Lib*, (2) vulnerable API method name, and (3) vulnerable line number. SIEGE reuses EvoSuite [110]—the popularly used test generation tool—to randomly generate tests, select tests based on their closeness to the specified target, and evolve those tests with some randomness to generate better tests that are closer to the target. SIEGE stops when the time budget is used up or some tests perfectly match the target.

Similar to SIEGE, TRANSFER also tries to generate security tests using GA, adopts the program analysis feature of EvoSuite to create both call graphs and control-flow graphs for *App*, and leverages the dynamic instrumentation feature of EvoSuite to assess test coverage.

Table 5.6: The comparison between ChatGPT-4.0 and state-of-the-art security test generators

	Tool Applicability (A)	**Test Compilability (C)**	**Vulnerability Exploitation (V)**
ChatGPT-4.0	55	40	24
TRANSFER	16	13	4
SIEGE	1	1	0

However, for any $\langle App, Lib \rangle$ pair, TRANFER requires users to specify (1) the vulnerable API and (2) a library test to show the vulnerability exploit. TRANSFER then dynamically instruments *Lib*, to identify the program states relevant to the vulnerability, and to extract the conditions that must be satisfied by any security test to show the proof-of-concept exploit. Finally, TRANSFER adopts the extracted information to guide EvoSuite and to generate security tests. Due to the adoption of exemplar security test from *Lib* and more advanced program analysis techniques, TRANSFER manifested better effectiveness than SIEGE in prior work [126].

5.4.4.1 Experiment

We prepared the inputs required by SIEGE and TRANSFER for 55 $\langle App, Lib \rangle$ pairs, and executed both tools accordingly. For each test output by either tool, we copied the code and pasted it to appropriate places. We leveraged the build process to reveal compilation errors and applied minor fixes to obvious and simple compilation errors, ensuring to fairly compare the outputs by different tools. If compilation succeeded, we further executed the project with the generated test to observe runtime behaviors. If any exception or runtime error was thrown, we studied the exception/error message, inspected the intermediate program status via step-by-step debugging, and discussed the relevance among authors until reaching a consensus.

5.4.4.2 Results.

As shown in Table 5.6, surprisingly, ChatGPT-4.0 outperformed both state-of-the-art tools considerably by having much better tool applicability, test compilability, and vulnerability exploitation. Specifically, ChatGPT-4.0 generated tests for all 55 $\langle App, Lib \rangle$ pairs, while TRANSFER only generated test functions or code snippets for 16 pairs. SIEGE worked much worse, generating tests for only one pair.

In terms of compilability, 13 out of the 16 tests generated by TRANSFER are compilable, while 3 tests fail to compile due to their usage of unknown entities. The only test output by SIEGE compiles successfully. Based on the small number of tests generated by TRANSFER and SIEGE, it is hard to conclude whether they were more likely to generate compilable tests than ChatGPT-4.0.

In terms of vulnerability exploitation, only four of the tests output by TRANSFER successfully trigger vulnerabilities; the remaining nine compilable tests fail to do so. Among these ineffective or less effective ones, six tests execute smoothly, without triggering any error or exception; three tests trigger irrelevant exceptions. The only test by SIEGE fails to trigger any vulnerability, because it throws an irrelevant exception. For the four tasks handled well by TRANSFER, ChatGPT-4.0 also successfully triggered vulnerabilities. Our observations imply that ChatGPT-4.0 worked overwhelmingly better than TRANSFER and SIEGE.

Several reasons can help explain why TRANSFER and SIEGE worked worse than ChatGPT-4.0 in security test generation. First, both tools adopt EvoSuite, to generate tests and execute the method-to-test M. However, EvoSuite is not always effective; some or even most of the tests generated by EvoSuite do not execute M at all. Second, both tools have difficulty synthesizing or mocking complex input parameters for M. For instance, neither tool synthesized a parameter of type `net.sourceforge.pmd.RuleSet`, but ChatGPT-4.0

mocked such an object via the Mockito framework.

Third, SIEGE cannot take in any exemplar test to infer the exploit method or vulnerability-triggering inputs, and it has no domain knowledge about security exploits. Meanwhile, although TRANSFER can take in exemplar tests to infer the exploit knowledge, it has difficulty incorporating inferred information into test generation. For instance, CODEC-134 is related to vulnerable APIs `Base64.decode(...)` and `Base64.decodeBase64(...)`. As shown in Fig. 5.4, we provided both TRANSFER and ChatGPT-4.0 inputs relevant to that vulnerability, including the exemplar library test to demonstrate the security exploit of `Base64.decode(...)` (see Fig. 5.4 (a)), and a client Java class that calls the vulnerable API `Base64.decodeBase64(...)` (see Fig. 5.4 (b)). We tried both tools to generate a security test for the client code, to trigger the vulnerability similarly as the exemplar test. In the figure, (c) and (d) separately show the tools' outputs. Unfortunately, TRANSFER could not reuse any domain knowledge demonstrated by the exemplar test. However, ChatGPT-4.0 successfully transferred the domain knowledge and effectively produced a security test.

Finding 4: *ChatGPT-4.0 outperformed both TRANSFER and SIEGE in terms of all metrics we evaluated: tool applicability, test compilability, and vulnerability exploitation. This observation implies the great potential of ChatGPT-4.0 in security test generation.*

5.5 Threats to validity

This study is based on self-craft prompts to instruct ChatGPT-4.0 to generate the vulnerability test. Moreover, the training date and model involvement for ChatGPT-4.0 is a black box, So the observation may be subject to human bias and the randomness behind ChatGPT-4.0.

Threats to Internal Validity: Our approach relies on the use of self-crafted prompts to instruct

ChatGPT-4.0 to generate vulnerability tests. Despite our intent to create these prompts as objectively and uniformly as possible, we are cognizant that inadvertent biases or assumptions could seep into them. For instance, our instructions are primarily tailored towards generating JUnit tests for Java classes, which may inadvertently limit the diversity of the test cases generated. This focus on JUnit tests could, in theory, prevent the generation of other potentially useful test structures or approaches. Therefore, while this study presents a comprehensive investigation within this specific scope, the results may not fully represent ChatGPT-4.0's capabilities to generate other types of tests or work with other testing frameworks. Moreover, Our study's design primarily focuses on single-prompt interactions with ChatGPT-4.0. However, it is known that ChatGPT-4.0 is designed to learn and improve its responses from a conversational context. This feature can be particularly beneficial in a scenario where an initial test generated by ChatGPT-4.0 contains compilation issues. In a multi-turn interaction, ChatGPT-4.0 could be informed about these bugs and fix them in a follow-up response. However, our single-prompt approach doesn't exploit this potential for iterative refinement. Thus, our findings may underestimate ChatGPT-4.0's full capabilities when used in a conversational, iterative manner.

Threats to External Validity: ChatGPT-4.0's exact training data and the intricacies of its model evolution remain undisclosed, rendering them a 'black box' to us. This obscurity presents a couple of challenges. Firstly, it introduces an element of unpredictability in ChatGPT-4.0's responses to our prompts. ChatGPT-4.0's understanding and generation of tests are influenced by its training data, and without complete knowledge of this data, it is challenging to fully comprehend or predict its responses. Secondly, the ongoing evolution of the model by its developers can lead to changes in its behavior over time. While we can control the input (prompts) and observe the output (generated tests), we have little insight into the underlying processes and changes that might affect these outputs. This lack of

control and visibility might introduce some degree of randomness in our results, potentially impacting the replicability of our study.

5.6 Discussion

From our study, we obtained the following insights about the strengths and weaknesses of ChatGPT-4.0.

ChatGPT-4.0 is always able to generate security tests, even though the test quality varies a lot. According to ChatGPT itself, "OpenAI did not use reinforcement learning with human feedback to train me. Instead, I was pre-trained using a combination of unsupervised and supervised learning techniques, such as language modeling, auto-encoding, and sequence prediction. My training involved processing massive amounts of text data from the internet, which allowed me to learn patterns and relationships between words and phrases." [53]. The pre-trained knowledge enables ChatGPT-4.0 to always produce tests in our study, but does not guarantee the quality. To provide quality assurance, future work can explore two directions. First, fine-tune or further train ChatGPT-4.0 specially for test generation using the dialogue data between humans [53], so that ChatGPT-4.0 learns to output better tests. Second, integrate ChatGPT-4.0 with automatic compilation and testing, so that compilation/testing errors get used as feedback to help ChatGPT-4.0 iteratively refine test generation.

ChatGPT-4.0 requires for exemplar Lib tests to generate security exploits, although such test examples are not always available or helpful. As shown in Section 5.4.3, without any test example, ChatGPT-4.0 could not generate any exploit; with the test examples provided, however, ChatGPT-4.0 generated exploits for 44% of the cases (24/55). The observations imply two things. First, ChatGPT-4.0 can be effectively applied

to generate security tests that mimic hand-crafted tests and exploit known vulnerabilities. Second, future work can put more effort into (1) providing more initial test examples, and (2) better training ChatGPT-4.0 so that it can fully leverage the domain knowledge embedded in test examples to produce more successful vulnerability exploits.

Compared with the design of program analysis-based tools, ChatGPT-4.0's design seems weak in inter-procedural analysis but stronger in code generation. Based on our experience, when a method-to-test M indirectly calls vulnerable API(s), ChatGPT-4.0 barely succeeded in vulnerability exploitation. One possible reason is that ChatGPT-4.0 cannot relate the program context in different methods via caller-callee relations. Our observation implies the future direction of combining program analysis techniques with ChatGPT-4.0 in novel ways, so that (1) ChatGPT-4.0 fulfills more exploits when the program context is complex or there is a long call chain between M and vulnerable APIs, or (2) program analysis-based tools (especially dynamic analysis tools) obtain better results when ChatGPT-4.0-generated tests trigger the execution of more paths.

5.7 Summary

We conduct an empirical study to evaluate the effectiveness of ChatGPT-4.0 in vulnerability test generation for known vulnerabilities in client projects. First, the findings demonstrate that ChatGPT-4.0 can generate unit tests that reveal known vulnerabilities with few-shot prompts; the produced vulnerability test function provides direction for investigation, such as bug fixes and security knowledge with ChatGPT-4.0. Second, it surpasses existing tools in capability and effectiveness, so integrating the ChatGPT-4.0 with other developed environments may help with vulnerability generation in the development process. Third, the impact of prompt mutation highlights the need for specific prompts with detailed instructions

and relevant background knowledge to enhance ChatGPT-4.0's effectiveness in generating high-quality security unit tests.

```
private static final String[]
BASE64_IMPOSSIBLE_CASES = {
      "ZE==",
      "ZmC=",
      "Zm9vYE==",
      "Zm9vYmC=",
   };
// Base64.decode(…) should reject the malicious
// inputs, in order to disable the potential data theft
@Test
   public void testBase64ImpossibleSamples() {
      Base64 codec = new Base64();
      for (String s : BASE64_IMPOSSIBLE_CASES) {
         try {
            codec.decode(s);
            fail();
         } catch (IllegalArgumentException ex) {
            // expected
         }
      }
   }
```

(a) An exemplar test from the library Apache Commons Codec

```
package com.egzosn.pay.common.util.sign.encrypt;
public class Base64 {
   private Base64() {}
// When this App depends on a vulnerable library version, it does not
// reject the malicious inputs that enable data theft or covert channel
   public static byte[] decode(String str) {
      return
org.apache.commons.codec.binary.Base64.decodeBase64(str);
   }
   public static String encode(byte[] bytes) {
      return
org.apache.commons.codec.binary.Base64.encodeBase64String(bytes);
   }
}
```

(b) A Java class that calls the vulnerable API Base64.decode() in Apache Commons Codec

```
Test 0:
String string0 = "Qg==\r\n";
byte[] byteArray0 = Base64.decode(string0);
Test 1:
byte[] byteArray0 = null;
String string0 = Base64.encode(byteArray0);
```

(c)Two tests generated by TRANSFER that fail to exploit the Apache Commons Codec vulnerability CODEC-134

```
import com.egzosn.pay.common.util.sign.encrypt.Base64;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.junit.MockitoJUnitRunner;
import static org.junit.Assert.*;

@RunWith(MockitoJUnitRunner.class)
public class Base64Test {

    private static final String[] BASE64_IMPOSSIBLE_CASES = {
        "ZE==", "ZmC=", "Zm9vYE==", "Zm9vYmC=",
    };

    @Test
    public void testDecode() {
        for (String s : BASE64_IMPOSSIBLE_CASES) {
            try {
                Base64.decode(s);
                fail("Should have thrown IllegalArgumentException");
            } catch (IllegalArgumentException ex) {
                // expected
            }
        }
    }
}
```

(d) The test generated by ChatGPT that successfully exploits the Commons Codec vulnerability CODEC-134

Figure 5.4: An example to show the output difference between TRANSFER and ChatGPT

Chapter 6

Conclusion

The Java platform, along with third-party open-source libraries, offers various APIs to support secure coding. However, using these APIs securely can be challenging for developers without cybersecurity training. Despite the availability of various tools designed to identify API insecure usage, it is still unclear how well they help developers with secure coding practices. In this dissertation, we intend to assess different aspects of current tools and reveal the gaps between existing work and developers' needs. To better help developers with secure code practice, we proposed a novel framework for detecting, repairing, and demonstrating security vulnerabilities related to API(s) insecure usage.

In the following sections, we will first summarize our works for mitigating the API insecure usage. Then, we present future work on how to help developers secure coding practice from various perspectives.

6.1 Summary

In this dissertation, we first conducted quantitative and qualitative analyses of current automatic API-related vulnerability detection tools across different aspects to reveal the gaps between existing solutions and developers' needs. This study shows that while some tools provide general guidance on misuse detections and repairs, they are insufficient to help developers fully eliminate API insecure usages. Developers still need better approaches to 1)

handle a wider range of misuse cases beyond just variable misuses as well as provide customized repair suggestions, and 2) provide a demonstration of the detected vulnerability caused by insecure use of API(s). To address the first limitation, we proposed Seader —a new approach to take in ⟨insecure, secure⟩ code examples, infer vulnerability-repair patterns from examples, and apply those patterns for vulnerability detection and repair suggestion. Compared with prior work, Seader offers a more powerful means for security experts to extend the pattern set of API-misuse detectors (extensibility), and concertizes security expertise as customized fixing edits for developers (capability). To address the second limitation, we explored using ChatGPT-4.0 in generating security exploits to demonstrate how vulnerabilities are propagated from software libraries to the client applications built on top of libraries. Our exploration indicates that ChatGPT-4.0 works effectively in generating security tests, given prompts that cover the relevant domain knowledge. ChatGPT-4.0 even outperformed state-of-the-art that leverage complex program analysis and genetic programming to generate diverse tests.

6.2 Future work

6.2.1 Vulnerability Detection and Testing for Java Code

In Chapter 3 and Chapter 4, Existing work has focused on analyzing insecure API usage, primarily in JCA and JSSE cryptographic libraries. These findings may not generalize to other Java codebases, like closed-source projects or third-party security libraries. Many Java frameworks and libraries expose powerful APIs that require proper and secure configuration by developers, developers can misconfigure these APIs to unsuitable or insecure defaults. For example, Mengsu and Meng et al. [99, 140] observed that users tend to disable CSRF

protection for convenience in StackOverflow posts. Additionally, our manual examination in Chapter 5 revealed vulnerabilities can arise from both code flaws(e.g., boundary checks), and non-code issues such as YAML and annotations. The existing tool SEADER currently focuses on API misuse patterns that involve (1) calling certain method APIs with incorrect parameter values, (2) calling certain method APIs in incorrect sequential orders, and (3) incorrectly overriding certain method APIs.

To help developers secure their code practice, we will generalize detection and testing approaches across a wider range of insecure Java API usage beyond just cryptographic libraries. Firstly, we plan to expand the supported patterns further to detect more insecure Java API usage types. Specifically, 1) Patterns that catch configuration errors across XML, YAML, and properties where API security is disabled or weakened. 2) Patterns that identify business logic flaws and violation of semantic constraints between APIs. Secondly, we plan to expand the tools to handle vulnerability detection in more applications, including identifying misuse of APIs related to web applications, databases, networking, containers, and other common Java libraries. The approach should be able to uncover vulnerabilities from code-level flaws, configuration issues, and common weakness categories. By expanding the vulnerability patterns modeled beyond the current cryptographic focus, we can enable the tool to flag a more diverse set of insecure API usage scenarios, support comprehensive detection of insecure Java API usage spanning configurations, business logic, and more.

6.2.2 Automatic Security Tests Generation using Large Language Models (LLMs)

In Chapter 5, we observed ChatGPT-4.0 to work effectively in generating security tests, given prompts that cover the relevant domain knowledge. ChatGPT-4.0 even outperformed

state-of-the-art tools to generate diverse tests. This implies that in the future, we can think about creating tools on top of ChatGPT-4.0, to synthesize prompts and generate security tests automatically.

Synthesize prompts. We designed the prompt template with the test domain knowledge and vulnerable API-relevant descriptive information. We fed ChatGPT-4.0 with full or different subsets of the descriptive information, and the result shows considerably different results of ChatGPT-4.0. The results provide two insights. First, ChatGPT-4.0 is very sensitive to prompt quality; there is a huge gap among its outcomes when it is given good or bad prompts. Namely, to fully leverage the strength of ChatGPT-4.0, we need to carefully design prompt templates. Second, the more domain knowledge included in the prompts, the better ChatGPT-4.0 works to generate tests as we need. For example, through our experiment, we found ChatGPT-4.0 exhibited proficiency in resolving some problems when provided with specific error information. To improve the quality of the prompts, we need to 1) incorporate compile-time feedback and user input into prompts, driving iterative refinement, and 2 leverage techniques like prompt engineering and lifelong learning to automatically fine-tune prompts based on project source code.

High-quality security tests generation. Among the test cases we evaluated in Chapter 5, 15 were non-compilable and couldn't be repaired with straightforward changes. We observed primary issues such as using undefined program entities (e.g., methods and classes) and unauthorized access to private class members. These observations highlight ChatGPT-4.0's limitation in assuring code compilation, particularly when complicated program contexts extending beyond a mere class are pivotal. To mitigate this, we need to further combine the LLMs with existing program analysis tools capable of offering granular program context. For instance, integrate program analysis techniques like static analysis and symbolic execution to extract structural information to inform test generation. Additionally, we plan

to fine-tune the model with program context information, train the model to understand the coding environment in which it operates and equip LLMs with the capabilities to handle vulnerability generation in a more specialized manner.

6.2.3 Secure Coding Practice Crossing Language

This dissertation focuses on security code practices in Java. Similar issues around insecure API usage likely exist across other programming languages (e.g., JavaScript, Python, Golang, PHP, etc.). As we have seen, developers often unintentionally introduce vulnerabilities into production code. There is a need to expand research on improving security code practices for developers in other languages beyond just Java. In the future, we will integrate more validation and testing methods for vulnerability detection across languages, such as fuzzing, AI-assisted analysis, and dynamic symbolic execution. By broadening the scope of techniques and languages studied, my work will offer more robust solutions that help developers write more secure code in any language.

Bibliography

[1] Nist samate: static analysis tool exposition (sate) iv. https://samate.nist.gov/SATE.html, 2012.

[2] SLOTH: TLS 1.2 vulnerability (CVE-2015-7575). https://access.redhat.com/articles/2112261, 2016.

[3] Developers lack skills needed for secure DevOps, survey shows. https://www.computerweekly.com/news/450424614/Developers-lack-skills-needed-for-secure-DevOps-survey-shows, 2017.

[4] MD5Function.java. https://github.com/apache/phoenix/blob/7987a74e6cea1103a028e128f98e2fb3c2252b82/phoenix-core/src/main/java/org/apache/phoenix/expression/function/MD5Function.java, 2017.

[5] GitHub - nearform / gammaray: Node.js vulnerability scanner. https://github.com/nearform/gammaray, 2019.

[6] Add 201 parametric crypto (JCA) misuses. https://github.com/stg-tud/MUBench/pull/427, 2019.

[7] Too few cybersecurity professionals is a gigantic problem for 2019. https://techcrunch.com/2019/01/27/too-few-cybersecurity-professionals-is-a-gigantic-problem-for-2019/, 2019.

[8] Cryptoguardoss/cryptoapi-bench. https://github.com/CryptoGuardOSS/cryptoapi-bench, 2020.

[9] Cryptoguardoss/cryptoguard. https://github.com/CryptoGuardOSS/cryptoguard, 2020.

[10] OWASP Dependency-Check. https://owasp.org/www-project-dependency-check/, 2020.

[11] Github. https://github.com, 2020.

[12] Java Secure Socket Extension (JSSE) Reference Guide. https://docs.oracle.com/javase/9/security/java-secure-socket-extension-jsse-reference-guide.htm, 2020.

[13] Gossiper.java. https://github.com/apache/cassandra/blob/79e693e16e2152097c5b27d2d7aaa1763e34f594/src/java/org/apache/cassandra/gms/Gossiper.java, 2020.

[14] Snyk vulnerability database. http://snyk.io/vuln, 2020.

[15] Soot. https://github.com/soot-oss/soot, 2020.

[16] Wala ir. https://github.com/wala/WALA/wiki/Intermediate-Representation-(IR), 2020.

[17] Cwe-327 : Avoid weak encryption providing not sufficient key size (jee) | cast appmarq. https://www.appmarq.com/public/tqi,1039028,CWE-327-Avoid-weak-encryption-providing-not-sufficient-key -size-JEE, 2021. (Accessed on 03/22/2021).

[18] Mubench. https://github.com/stg-tud/MUBench, 2021.

[19] AndroBugs. https://www.androbugs.com, 2021.

[20] AndroGuard. https://androguard.readthedocs.io/en/latest/#, 2021.

[21] ApacheCryptoAPI-Bench. https://github.com/CryptoAPI-Bench/ApacheCryptoAPI-Bench/tree/main/apache_codes, 2021.

[22] npm audit: Broken by Design. https://overreacted.io/npm-audit-broken-by-design/, 2021.

[23] CWE - Common Weakness Enumeration. https://cwe.mitre.org/index.html, 2021.

[24] CWE-295: Improper Certificate Validation. https://cwe.mitre.org/data/definitions/295.html, 2021. URL https://cwe.mitre.org/data/definitions/295.html.

[25] CWE-326: Inadequate Encryption Strength. https://cwe.mitre.org/data/definitions/326.html, 2021. URL https://cwe.mitre.org/data/definitions/326.html.

[26] CWE-327: Use of a Broken or Risky Cryptographic Algorithm. https://cwe.mitre.org/data/definitions/327.html, 2021.

[27] CWE-330: Use of Insufficiently Random Values. https://cwe.mitre.org/data/definitions/330.html, 2021. URL https://cwe.mitre.org/data/definitions/330.html.

[28] CWE-757: Selection of Less-Secure Algorithm During Negotiation ('Algorithm Downgrade'). https://cwe.mitre.org/data/definitions/757.html, 2021.

[29] CWE-798: Use of Hard-coded Credentials. https://cwe.mitre.org/data/definitions/798.html, 2021. URL https://cwe.mitre.org/data/definitions/798.html.

[30] Find security bugs, 2021. URL https://find-sec-bugs.github.io/.

[31] Jlint. http://artho.com/jlint/, 2021.

[32] Maven. https://maven.apache.org, 2021.

[33] MobSF/Mobile-Security-Framework-MobSF. https://github.com/MobSF/Mobile-Security-Framework-MobSF, 2021.

[34] OWASP Benchmark. https://owasp.org/www-project-benchmark/, 2021.

[35] OWASP/Benchmark. https://github.com/OWASP/Benchmark, 2021.

[36] SonarQube. https://github.com/SonarSource/sonarqube, 2021.

[37] Supply chain attacks show why you should be wary of third-party providers. https://www.csoonline.com/article/3191947/supply-chain-attacks-show-why-you-should-be-wary-of-third-party-providers.html, 2021.

[38] Supply chain attacks on open source software grew 650% in 2021. https://techmonitor.ai/technology/cybersecurity/supply-chain-attacks-open-source-software-grew-650-percent-2021, 2021.

[39] Weak SSL/TLS protocols should not be used. https://rules.sonarsource.com/java/tag/cwe/RSPEC-4423, 2021.

[40] Argus saf. http://pag.arguslab.org/argus-saf, 2021.

[41] Xanitizer by rigs it - because security matters, 2021. URL https://www.rigs-it.com/xanitizer/.

[42] False Positives in Vulnerability Scanning: Why We Think We Can Do Better. https://www.lunasec.io/docs/blog/the-issue-with-vuln-scanners/, 2022.

[43] Introducing ChatGPT. https://openai.com/blog/chatgpt, 2023.

[44] OpenRefine. https://github.com/OpenRefine/OpenRefine, 2023.

[45] american fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2023.

[46] NCI-Agency/anet: Advisor Network, 2023.

[47] Codec. https://commons.apache.org/proper/commons-codec/, 2023.

[48] Commons Compress - Overview. https://commons.apache.org/proper/commons-compress/, 2023.

[49] apache / commons-io. https://github.com/apache/commons-io, 2023.

[50] apache / cxf. https://github.com/apache/cxf, 2023.

[51] sonatype-nexus-community / auditjs: Audits an NPM package.json file to identify known vulnerabilities. https://github.com/sonatype-nexus-community/auditjs, 2023.

[52] The Legion of the Bouncy Castle. https://www.bouncycastle.org, 2023.

[53] How does ChatGPT actually work? https://www.zdnet.com/article/how-does-chatgpt-work/, 2023.

[54] CVE-2020-28052 Detail. https://nvd.nist.gov/vuln/detail/cve-2020-28052, 2023.

[55] About Dependabot alerts. https://docs.github.com/en/code-security/dependabot/dependabot-alerts/about-dependabot-alerts, 2023.

[56] Dom4j. https://dom4j.github.io, 2023.

[57] alibaba / fastjson. https://github.com/alibaba/fastjson, 2023.

[58] What Is Fuzz Testing and How Does It Work? | Synopsys. https://www.synopsys.com/glossary/what-is-fuzz-testing.html, 2023.

[59] apache / httpcomponents-client. https://github.com/apache/httpcomponents-client, 2023.

[60] stleary / JSON-java. https://github.com/stleary/JSON-java, 2023.

[61] Inside ChatGPT's Brain: Large Language Models. https://serokell.io/blog/language-models-behind-chatgpt, 2023.

[62] FasterXML / jackson-databind. https://github.com/FasterXML/jackson-databind, 2023.

[63] FasterXML / jackson-dataformats-binary. https://github.com/FasterXML/jackson-dataformats-binary, 2023.

[64] FasterXML / jackson-modules-java8. https://github.com/FasterXML/jackson-modules-java8, 2023.

[65] OWASP / json-sanitizer. https://github.com/OWASP/json-sanitizer, 2023.

[66] netplex / json-smart-v1. https://github.com/netplex/json-smart-v1, 2023.

[67] netplex / json-smart-v2. https://github.com/netplex/json-smart-v2, 2023.

[68] junrar / junrar. https://github.com/junrar/junrar, 2023.

[69] Mockito. https://site.mockito.org/, 2023. Accessed on June 12, 2023.

[70] npm-audit. https://docs.npmjs.com/cli/v9/commands/npm-audit, 2023.

[71] NVD. https://nvd.nist.gov, 2023.

[72] OSS-Fuzz. https://google.github.io/oss-fuzz/, 2023.

[73] OWASP Top Ten. https://owasp.org/www-project-top-ten/, 2023.

[74] Plexus Archiver Component. https://codehaus-plexus.github.io/plexus-archiver/index.html, 2023.

[75] eclipse / rdf4j. https://github.com/eclipse/rdf4j, 2023.

[76] Retire.js. https://retirejs.github.io/retire.js/, 2023.

[77] xerial / snappy-java. https://github.com/xerial/snappy-java, 2023.

[78] Automatic fixing with snyk fix - Snyk User Docs. https://docs.snyk.io/snyk-cli/test-for-vulnerabilities/automatic-remediation-with-snyk-fix, 2023.

[79] Test - Snyk User Docs. https://docs.snyk.io/snyk-cli/commands/test, 2023.

[80] spring-projects / spring-data-commons. https://github.com/spring-projects/spring-data-commons, 2023.

[81] spring-projects / spring-security. https://github.com/spring-projects/spring-security, 2023.

[82] The Next Supply Chain Attack Vector: Open-Source Software. https://www.supplychainbrain.com/blogs/1-think-tank/post/36830-the-next-supply-attack-vector-open-source-software, 2023.

[83] Apache Tika. https://tika.apache.org, 2023.

[84] haraldk / TwelveMonkeys. https://github.com/haraldk/TwelveMonkeys, 2023.

[85] XStream. https://x-stream.github.io, 2023.

[86] srikanth-lingala / zip4j. https://github.com/srikanth-lingala/zip4j, 2023.

[87] ZT Zip. https://github.com/zeroturnaround/zt-zip, 2023.

[88] Sharmin Afrose, Sazzadur Rahaman, and Danfeng Yao. Cryptoapi-bench: A comprehensive benchmark on java cryptographic api misuses. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 49–61. IEEE, 2019.

[89] Sharmin Afrose, Ya Xiao, Sazzadur Rahaman, Barton P. Miller, Danfeng, and Yao. Evaluation of static vulnerability detection tools with java cryptographic api benchmarks, 2021.

[90] Sharmin Afrose, Ya Xiao, Sazzadur Rahaman, Barton P. Miller, and Danfeng Daphne Yao. Evaluation of static vulnerability detection tools with java cryptographic api benchmarks. *ArXiv*, abs/2112.04037, 2021.

[91] Kaled M. Alshmrany, Mohannad Aldughaim, Ahmed Bhayat, and Lucas C. Cordeiro. Fusebmc: An energy-efficient test generator for finding security vulnerabilities in c programs. In Frédéric Loulergue and Franz Wotawa, editors, *Tests and Proofs*, pages 85–105, Cham, 2021. Springer International Publishing. ISBN 978-3-030-79379-1.

[92] S. Amann, H. A. Nguyen, S. Nadi, T. N. Nguyen, and M. Mezini. A systematic evaluation of static api-misuse detectors. *IEEE Transactions on Software Engineering*, 45(12):1170–1188, 2019. doi: 10.1109/TSE.2018.2827384.

[93] Sven Amann, Sarah Nadi, Hoan Anh 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 2016, 2016. doi: 10.1145/2901739.2903506. URL http://dx.doi.org/10.1145/2901739.2903506.

[94] Kijin An, Na Meng, and Eli Tilevich. Automatic inference of java-to-swift translation rules for porting mobile applications. In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, MOBILESoft ?18, page 180?190, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357128. doi: 10.1145/3197231.3197240. URL https://doi.org/10.1145/3197231.3197240.

[95] Thanassis Avgerinos, Sang Kil Cha, Alexandre Rebert, Edward J Schwartz, Maverick Woo, and David Brumley. Automatic exploit generation. *Communications of the ACM*, 57(2):74–84, 2014.

[96] David Brumley, Pongsin Poosankam, Dawn Song, and Jiang Zheng. Automatic patch-based exploit generation is possible: Techniques and implications. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 143–157, 2008. doi: 10.1109/SP.2008.17.

[97] Sang Kil Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. Unleashing mayhem on binary code. In *2012 IEEE Symposium on Security and Privacy*, pages 380–394, 2012. doi: 10.1109/SP.2012.31.

[98] Eason Chen, Ray Huang, Han-Shin Chen, Yuen-Hsien Tseng, and Liang-Yi Li. Gptutor: a chatgpt-powered programming tool for code explanation, 2023.

[99] M. Chen, F. Fischer, N. Meng, X. Wang, and J. Grossklags. How reliable is the crowdsourced knowledge of security implementation? In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 536–547, May 2019. doi: 10.1109/ICSE.2019.00065.

[100] Mengsu Chen, Felix Fischer, Na Meng, Xiaoyin Wang, and Jens Grossklags. How reliable is the crowdsourced knowledge of security implementation? *arXiv preprint arXiv:1901.01327*, 2019.

[101] Bodin Chinthanet, Serena Elisa Ponta, Henrik Plate, Antonino Sabetta, Raula Gaikovina Kula, Takashi Ishio, and Kenichi Matsumoto. Code-based vulnerability detection in node.js applications: How far are we? In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ASE '20, pages 1199–1203, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450367684. doi: 10.1145/3324884.3421838. URL https://doi.org/10.1145/3324884.3421838.

[102] Jared Demott, Dr Richard, R.J. Enbody, Dr William, and William Punch. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing. In *Black Hat and DEFCON*, 07 2007.

[103] 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*, pages 73–84, 2013.

[104] Sascha Fahl, Marian Harbach, Thomas Muders, Lars Baumgärtner, Bernd Freisleben, and Matthew Smith. 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*, pages 50–61, 2012.

[105] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. Association for Computing Machinery.

ISBN 9781450330138. doi: 10.1145/2642937.2642982. URL https://doi.org/10.1145/2642937.2642982.

[106] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy amp;paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136, 2017. doi: 10.1109/SP.2017.31.

[107] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 121–136. IEEE, 2017.

[108] Felix Fischer, Huang Xiao, Ching-Yu Kao, Yannick Stachelscheid, Benjamin Johnson, Danial Razar, Paul Fawkesley, Nat Buckley, Konstantin Böttinger, Paul Muntean, and Jens Grossklags. Stack overflow considered helpful! deep learning security nudges towards stronger cryptography. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 339–356, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL https://www.usenix.org/conference/usenixsecurity19/presentation/fischer.

[109] Beat Fluri, Michael Wuersch, Martin PInzger, and Harald Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on software engineering*, 33(11):725–743, 2007.

[110] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[111] Jyoti Gajrani, Meenakshi Tripathi, Vijay Laxmi, Gaurav Somani, Akka Zemmari, and Manoj Singh Gaur. Vulvet: Vetting of vulnerabilities in android apps to thwart exploitation. *Digital Threats: Research and Practice*, 1(2):1–25, 2020.

[112] Vinod Ganapathy, Sanjit A. Seshia, Somesh Jha, Thomas W. Reps, and Randal E. Bryant. Automatic discovery of api-level exploits. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 312–321, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 1581139632. doi: 10.1145/1062455.1062518. URL https://doi.org/10.1145/1062455.1062518.

[113] Jun Gao, Pingfan Kong, Li Li, Tegawende F. Bissyande, 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)*, pages 388–398, 2019. doi: 10.1109/MSR.2019.00065.

[114] Jun Gao, Li Li, Pingfan Kong, Tegawende F. Bissyande, and Jacques Klein. Understanding the evolution of android app vulnerabilities. *IEEE Transactions on Reliability*, 70(1):212–230, 2021. doi: 10.1109/TR.2019.2956690.

[115] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. The most dangerous code in the world: validating ssl certificates in non-browser software. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 38–49. ACM, 2012.

[116] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing: Sage has had a remarkable impact at microsoft. *Queue*, 10(1):20–27, jan 2012. ISSN 1542-7730. doi: 10.1145/2090147.2094081. URL https://doi.org/10.1145/2090147.2094081.

[117] Matthew Green and Matthew Smith. Developers are not the enemy!: The need for usable security apis. *IEEE Security & Privacy*, 14(5):40–46, 2016.

[118] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, VN Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting ssl usage in applications with sslint. In *2015 IEEE Symposium on Security and Privacy*, pages 519–534. IEEE, 2015.

[119] Roya Hosseini and Peter Brusilovsky. Javaparser: A fine-grain concept indexing tool for java problems. In *CEUR Workshop Proceedings*, volume 1009, pages 60–63. University of Pittsburgh, 2013.

[120] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12): 92–106, December 2004. ISSN 0362-1340. doi: 10.1145/1052883.1052895. URL https://doi.org/10.1145/1052883.1052895.

[121] Emanuele Iannone, Dario Di Nucci, Antonino Sabetta, and Andrea De Lucia. Toward automated exploit generation for known vulnerabilities in open-source libraries. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 396–400. IEEE, 2021.

[122] S. Jalil, S. Rafi, T. D. LaToza, K. Moran, and W. Lam. Chatgpt and software testing education: Promises & perils. In *2023 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 4130–4137, Los Alamitos, CA, USA, apr 2023. IEEE Computer Society. doi: 10.1109/ICSTW58534.2023.00078. URL https://doi.ieeecomputersociety.org/10.1109/ICSTW58534.2023.00078.

[123] Java Cryptography Architecture. Java cryptography architecture. https://docs.oracle.com/javase/9/security/java-cryptography-architecture-jca-reference-guide.htm.

[124] M. Kabir, Y. Wang, D. Yao, and N. Meng. How do developers follow security-relevant best practices when using npm packages? In *2022 IEEE Secure Development Conference (SecDev)*, pages 77–83, Los Alamitos, CA, USA, oct 2022. IEEE Computer Society. doi: 10.1109/SecDev53368.2022.00027. URL https://doi.ieeecomputersociety.org/10.1109/SecDev53368.2022.00027.

[125] B. Kaliski. PKCS #5: Password-Based Cryptography Specification Version 2.0. RFC 2898 (Informational), September 2000. URL http://www.ietf.org/rfc/rfc2898.txt.

[126] Hong Jin Kang, Truong Giang Nguyen, Bach Le, Corina S Păsăreanu, and David Lo. Test mimicry to assess the exploitability of library vulnerabilities. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 276–288, 2022.

[127] Stefan Krüger, Sarah Nadi, Michael Reif, Karim Ali, Mira Mezini, Eric Bodden, Florian Göpfert, Felix Günther, Christian Weinert, Daniel Demmler, et al. Cognicrypt: supporting developers in using cryptography. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 931–936. IEEE, 2017.

[128] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. Crysl: An extensible approach to validating the correct usage of cryptographic apis. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[129] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration. *Empirical Software Engineering*, 23:384–417, 2018.

[130] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[131] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. *CoRR*, abs/1801.01681, 2018. URL http://arxiv.org/abs/1801.01681.

[132] Siqi Ma, David Lo, Teng Li, and Robert H Deng. Cdrep: Automatic repair of cryptographic misuses in android applications. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, pages 711–722, 2016.

[133] Siqi Ma, Ferdian Thung, David Lo, Cong Sun, and Robert H Deng. Vurle: Automatic vulnerability detection and repair by learning from examples. In *European Symposium on Research in Computer Security*, pages 229–246. Springer, 2017.

[134] James Manger. A chosen ciphertext attack on rsa optimal asymmetric encryption padding (oaep) as standardized in pkcs #1 v2.0. In Joe Kilian, editor, *Advances in Cryptology — CRYPTO 2001*, pages 230–238, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. ISBN 978-3-540-44647-7.

[135] Aaron Marback, Hyunsook Do, Ke He, Samuel Kondamarri, and Dianxiang Xu. Security test generation using threat trees. In *2009 ICSE Workshop on Automation of Software Test*, pages 62–69, 2009. doi: 10.1109/IWAST.2009.5069042.

[136] Junnosuke Matsumoto, Yoshiki Higo, and Shinji Kusumoto. Beyond gumtree: A hybrid approach to generate edit scripts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 550–554, 2019. doi: 10.1109/MSR.2019.00082.

[137] Florian Mendel, Tomislav Nad, and Martin Schläffer. Improving local collisions: New

attacks on reduced sha-256. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology – EUROCRYPT 2013*, pages 262–278, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-38348-9.

[138] Na Meng, Miryung Kim, and Kathryn S. McKinley. Systematic editing: Generating program transformations from an example. In *PLDI*, pages 329–342, 2011.

[139] Na Meng, Miryung Kim, and Kathryn McKinley. LASE: Locating and applying systematic edits. In *ICSE*, page 10, 2013.

[140] Na Meng, Stefan Nagy, Danfeng Yao, Wenjie Zhuang, and Gustavo Arango-Argoty. Secure coding practices in java: Challenges and vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 372–383. IEEE, 2018.

[141] Ravindra Metta, Raveendra Kumar Medicherla, and Samarjit Chakraborty. Bmc+fuzz: Efficient and effective test generation. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1419–1424, 2022. doi: 10.23919/DATE54114.2022.9774672.

[142] Robert C. Miller and Brad A. Myers. Interactive simultaneous editing of multiple text regions. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 161–174, Berkeley, CA, USA, 2001. USENIX Association. ISBN 1-880446-09-X.

[143] 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*, pages 133–146, 2018.

[144] 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, pages 935–946, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3900-1. doi: 10.1145/2884781.2884790. URL http://doi.acm.org/10.1145/2884781.2884790.

[145] Nathalia Nascimento, Paulo Alencar, and Donald Cowan. Comparing software developers with chatgpt: An empirical investigation, 2023.

[146] Kristen L Newbury. Automated Hotfixes for Misuses of Crypto APIs. Master's thesis, University of Alberta, 2020.

[147] Duc Cuong Nguyen, Erik Derr, Michael Backes, and Sven Bugiel. Up2dep: Android tool support to fix insecure code dependencies. In *Annual Computer Security Applications Conference*, ACSAC '20, pages 263–276, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450388580. doi: 10.1145/3427228.3427658. URL https://doi.org/10.1145/3427228.3427658.

[148] Nikolaos Nikolaidis, Karolos Flamos, Daniel Feitosa, Alexander Chatzigeorgiou, and Apostolos Ampatzoglou. The End of an Era: Can Ai Subsume Software Developers? Evaluating Chatgpt and Copilot Capabilities Using Leetcode Problems. http://dx.doi.org/10.2139/ssrn.4422122.

[149] Rolf Oppliger. *SSL and TLS: Theory and Practice*. Artech House, Inc., USA, 2009. ISBN 1596934476.

[150] 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 Juan Garbajosa, Xiaofeng Wang, and Ademar Aguiar, editors, *Agile Processes in Software Engineering and Extreme Programming*, pages 86–103, Cham, 2018. Springer International Publishing. ISBN 978-3-319-91602-6.

[151] Rumen Paletov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Inferring crypto api rules from code changes. *ACM SIGPLAN Notices*, 53(4):450–464, 2018.

[152] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358231. doi: 10.1145/3239235.3268920. URL https://doi.org/10.1145/3239235.3268920.

[153] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*, 48(5):1592–1609, 2020.

[154] Pablo Martín Pérez, Joanna Filipiak, and José María Sierra. Lapse+ static analysis security software: Vulnerabilities detection in java ee applications. In James J. Park, Laurence T. Yang, and Changhoon Lee, editors, *Future Information Technology*, pages 148–156, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-22333-4.

[155] Nam H. Pham, Tung Thanh Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Detection of recurring software vulnerabilities. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 447–456, New York, NY, USA, 2010. Association for Computing Machinery. ISBN 9781450301169. doi: 10.1145/1858996.1859089. URL https://doi.org/10.1145/1858996.1859089.

[156] Serena E. Ponta, Henrik Plate, Antonino Sabetta, Michele Bezzi, and Cédric Dangremont. A manually-curated dataset of fixes to vulnerabilities of open-source software. In *Proceedings of the 16th International Conference on Mining Software Repositories*,

MSR '19, pages 383–387. IEEE Press, 2019. doi: 10.1109/MSR.2019.00064. URL https://doi.org/10.1109/MSR.2019.00064.

[157] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empirical Software Engineering*, 25(5):3175–3215, 2020.

[158] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng Yao. 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*, CCS '19, pages 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3345659. URL https://doi.org/10.1145/3319535.3345659.

[159] Sazzadur Rahaman, Ya Xiao, Sharmin Afrose, Fahad Shaon, Ke Tian, Miles Frantz, Murat Kantarcioglu, and Danfeng (Daphne) Yao. 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*, CCS '19, pages 2455–2472, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367479. doi: 10.1145/3319535.3345659. URL https://doi.org/10.1145/3319535.3345659.

[160] Kristiina Rahkema and Dietmar Pfahl. Swiftdependencychecker: Detecting vulnerable dependencies declared through cocoapods, carthage and swift pm. In *Proceedings of the 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems*, MOBILESoft '22, pages 107–111, New York, NY, USA, 2022. Association for

Computing Machinery. ISBN 9781450393010. doi: 10.1145/3524613.3527806. URL https://doi.org/10.1145/3524613.3527806.

[161] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. Learning syntactic program transformations from examples. In *Proceedings of the 39th International Conference on Software Engineering*, pages 404–415. IEEE Press, 2017.

[162] Shuai Shao, Guowei Dong, Tao Guo, Tianchang Yang, and Chenjie Shi. Modelling analysis and auto-detection of cryptographic misuse in android applications. In *2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing*, pages 75–80. IEEE, 2014.

[163] Yaron Sheffer, Ralph Holz, and Peter Saint-Andre. Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS). RFC 7525, May 2015. URL https://rfc-editor.org/rfc/rfc7525.txt.

[164] Gurpreet Singh and Supriya. A study of encryption algorithms (rsa, des, 3des and aes) for information security. *International Journal of Computer Applications*, 67:33–38, 2013.

[165] Larry Singleton, Rui Zhao, Myoungkyu Song, and Harvey Siy. Cryptotutor: Teaching secure coding practices through misuse pattern detection. In *Proceedings of the 21st Annual Conference on Information Technology Education*, pages 403–408, 2020.

[166] Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt, 2023.

[167] Spotbugs. spotbugs/spotbugs. URL https://github.com/spotbugs/spotbugs.

[168] Ari Takanen, Jared Demott, Charles Miller, and Atte Kettunen. *Fuzzing for Software Security Testing and Quality Assurance, Second Edition*. Artech House, 2017.

[169] Haoye Tian, Weiqi Lu, Tsz On Li, Xunzhu Tang, Shing-Chi Cheung, Jacques Klein, and Tegawendé F. Bissyandé. Is chatgpt the ultimate programming assistant – how far is it?, 2023.

[170] Harshal Tupsamudre, Monika Sahu, Kumar Vidhani, and Sachin Lodha. Fixing the fixes: Assessing the solutions of sast tools for securing password storage. In Matthew Bernhard, Andrea Bracciali, L. Jean Camp, Shin'ichiro Matsuo, Alana Maurushat, Peter B. Rønne, and Massimiliano Sala, editors, *Financial Cryptography and Data Security*, pages 192–206, Cham, 2020. Springer International Publishing.

[171] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13. IBM Press, 1999.

[172] 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 *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1329–1341, 2014.

[173] Jeff Williams and Arshan Dabirsiaghi. The unfortunate reality of insecure libraries. *Asp. Secur. Inc*, pages 1–26, 2012.

[174] Dianxiang Xu, Manghui Tu, Michael Sanford, Lijo Thomas, Daniel Woodraska, and Weifeng Xu. Automated security test generation with formal threat models. *IEEE Transactions on Dependable and Secure Computing*, 9(4):526–540, 2012. doi: 10.1109/TDSC.2012.24.

[175] Shengzhe Xu, Ziqi Dong, and Na Meng. Meditor: Inference and application of api migration edits. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC ?19, page 335?346. IEEE Press, 2019. doi: 10.1109/ICPC.2019.00052. URL https://doi.org/10.1109/ICPC.2019.00052.

[176] Zhiwu Xu, Xiongya Hu, Yida Tao, and Shengchao Qin. Analyzing cryptographic api usages for android applications using hmm and n-gram. In *2020 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, pages 153–160. IEEE, 2020.

[177] Sebastian Zander, Grenville Armitage, and Philip Branch. A survey of covert channels and countermeasures in computer network protocols. *IEEE Communications Surveys & Tutorials*, 9(3):44–57, 2007. doi: 10.1109/COMST.2007.4317620.

[178] Ying Zhang, Md Mahir Asef Kabir, Ya Xiao, Danfeng Daphne Yao, and Na Meng. Automatic detection of java cryptographic api misuses: Are we there yet. *IEEE Transactions on Software Engineering*, 49(1):288–303, 2022.