

Systematically Evaluating Vulnerability Detectors
-
Discovering the Gaps within Design and Practice

Amit Seal Ami

Bangladesh

Master of Science, Software Engineering, University of Dhaka, 2014
Bachelor's Degree in Information Technology, University of Dhaka, 2012

A Dissertation presented to the Graduate Faculty of
The College of William and Mary in Virginia in Candidacy for the Degree of
Doctor of Philosophy

Department of Computer Science

The College of William and Mary in Virginia
May 2025

APPROVAL PAGE

This Dissertation is submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy



Amit Seal Ami

Approved by the Committee, May 2025



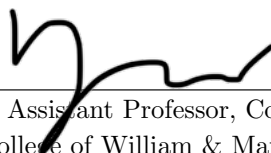
Committee Co-Chair

Adwait Nadkarni, Associate Professor, Computer Science
College of William & Mary

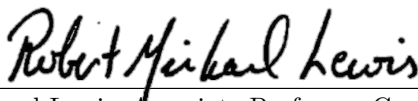


Committee Co-Chair

Denys Poshyvanyk, Chancellor Professor, Computer Science
College of William & Mary



Yixuan Zhang, Assistant Professor, Computer Science
College of William & Mary



Robert Michael Lewis, Associate Professor, Computer Science
College of William & Mary



Kevin Moran, Assistant Professor, Computer Science
University of Central Florida

COMPLIANCE PAGE

Research approved by

Protection of Human Subjects Committee

Protocol number(s): PHSC-2023-01-25-16069-apnadkarni

Date(s) of approval: 02/13/2023

ABSTRACT

The improvement and adoption of security-focused static analysis tools have significantly improved the detection of vulnerabilities, such as crypto-API misuse and data leaks. We are continuously becoming more dependent on these security analysis techniques because of their convenience automation, continuous integration and development support, and statically finding vulnerabilities efficiently.

However, there is a critical gap in these tools’ practical and effective application. Other than static benchmarks, we have yet to devise a mechanism to identify previously unknown flaws in these tools. Furthermore, how industry professionals perceive these tools and their limitations is unknown. As a result, the current progress towards designing and developing effective, practical static analysis-based security tools is hindered.

To address these gaps, we (1) contextualize mutation testing techniques by proposing and implementing a framework called μ SE. μ SE systematically evaluates static analysis-based data-leak detectors, identifying previously unknown soundness issues/flaws and exploring the propagation of 25 found flaws that may propagate or even resurface, across the lifecycle of three data leak detectors, due to implicit dependencies, assumptions, or similar design principles. Next, (2) we evaluate cryptographic API misuse detectors (crypto-detectors). To do this, we create a taxonomy of crypto-API misuse based on the existing state-of-the-art literature and documentation from industry sources spanning over the past 20 years. By analyzing the patterns of underlying crypto-APIs, we develop mutation operators and mutation scopes for creating mutations of crypto-API misuse. An implementation of this approach, namely MASC, is used to systematically evaluate 14 prominent crypto-detectors from industry and academia, finding 25 previously unknown flaws affecting these crypto-detectors. Based on our discussion with the developers of the crypto-detectors about the nature of the found flaws, we identify and highlight the need to shift from a technique-centric to a security-centric approach to address evolving software security challenges. Afterward, (3) we study the gap that exists in the design and adoption of static analysis-based security tools. Through interviews with 20 real-world practitioners, we analyze their perceptions, expectations, and challenges with SAST tools. By applying thematic analysis, we identify critical insights into developer needs and discuss areas for improvement in SAST design and development.

Finally, we qualitatively analyze a statistically significant sample of existing bug reports of open-source static analysis based security testing tools to identify the internal, implicit factors that influence the acknowledging, addressing, and prioritization of the reported issues as bugs and/or feature requests, and identify the conflicting perspectives of designers and developers stemming from the duality of the vulnerability detectors; as security-assurance tools and developer-friendly tools.

TABLE OF CONTENTS

Acknowledgments	vii
Dedication	viii
List of Tables	ix
List of Figures	xi
1 Introduction	1
1.1 Dissertation Organization	3
2 Background and Related Work	6
2.1 Soundness and Static Analysis Tools	6
2.1.1 Motivating Example	7
2.1.2 Background on Mutation Analysis	8
2.1.3 Related Work on Mutation Analysis and Data Leak Detectors .	9
2.2 Static Analysis for Crypto-API Misuse Detection	11
2.2.1 Motivating Example	12
2.2.2 Threat Model	12
2.2.3 Related Work on Crypto-API Misuse Detection	14
2.3 Static Analysis Tools for Security in Practice	16
2.4 Bugs in Vulnerability detectors	18
3 Evaluating Data Leak Detectors using Mutation	22

3.1	μ SE	27
3.2	Design	30
3.2.1	Security Operators	31
3.2.2	Mutation Schemes	33
3.2.2.1	Mutation Scheme Strategy 1: Leveraging Android Abstractions	34
3.2.2.2	Mutation Scheme Strategy 2: Evaluating Reach- ability	35
3.2.2.3	Mutation Scheme Strategy 3: Leveraging the Se- curity Goal	36
3.2.3	Analysis Feasibility & Methodology	38
3.2.4	Leveraging Security Operators and Mutation Schemes for other Security Goals	40
3.3	Implementation	41
3.4	Evaluation Overview	44
3.5	Executing μ SE to Create Mutants Representing Data Leaks	46
3.6	In-depth Evaluation of Data Leak Detection Tools with μ SE	48
3.6.1	Evaluating FlowDroid	49
3.6.2	Evaluating Argus	52
3.6.3	Evaluating HornDroid	53
3.6.4	Effectiveness of Individual Schemes in Finding Flaws	55
3.7	Flaw Propagation Study	56
3.7.1	Propagation of FlowDroid’s Flaws (F1-F13)	57
3.7.2	Propagation of Argus’s Flaws (F14-F22)	58
3.7.3	Propagation of HornDroid’s Flaws (F23-F25)	59
3.8	Discussion	60
3.9	Limitations	63

3.10 Chapter Summary	64
4 Evaluating Crypto-API Misuse Detectors	69
4.1 The MASC Framework	74
4.2 Taxonomy of Cryptographic Misuse	75
4.2.1 Identifying Information Sources	77
4.2.2 Search, Inclusion, and Exclusion Criteria	77
4.2.3 Misuse Extraction and Clustering	77
4.2.4 Extending the Taxonomy	78
4.3 Usage-based Mutation Operators	78
4.3.1 Operators based on Restrictive Crypto API Invocation	80
4.3.2 Operators based on Flexible Crypto API Invocations	83
4.3.2.1 Method Overriding	83
4.3.2.2 Class Extension	84
4.3.2.3 Object Instantiation Operators	84
4.4 Threat-based Mutation Scopes	85
4.5 Implementation	86
4.6 Evaluation Overview and Methodology	89
4.7 Results and Findings	94
4.8 Limitations	108
4.9 Discussion and Summary	110
4.9.1 Security-centric Evaluation vs. Technique-centric Design	110
4.9.2 Defining “Scope” for the Technique-centric Design	111
4.9.3 Utility of Seemingly-Uncommon or Evasive Tests	112
4.9.4 The Need to Strengthen Crypto-Detectors	112
4.9.5 Towards Crypto-Detectors Strengthened by a Security-Centric Evaluation	113

5	Identifying the Gaps in the Practice of SASTs	117
5.1	Methodology	120
5.1.1	Summary of the Survey Protocol and Results	120
5.1.2	Interview Protocol	121
5.1.2.1	Interview Recruitment	122
5.1.2.2	Interview Protocol and Ethics	122
5.1.2.3	Interview Pilot & Refinement	123
5.1.2.4	Interviewing Procedure	123
5.1.3	Structure of the Interview Guide	124
5.1.3.1	Participants, Projects, and Organizations	124
5.1.3.2	Organization and Security	124
5.1.3.3	Organizational Context of SAST	126
5.1.3.4	Expectations from SAST	126
5.1.3.5	Impact of Unsound/Flawed SAST	126
5.1.3.6	Challenges and Improvements	127
5.1.4	Transcribing, Coding and Analysis	127
5.2	Interview Results	129
5.2.1	Participants, Projects, and Organizations	129
5.2.2	Organization and Security	130
5.2.3	Organizational Context of SAST	132
5.2.4	Expectations from SAST	135
5.2.5	Impact of Unsound/Flawed SAST	140
5.2.6	Challenges and Improvements	144
5.3	Threats to Validity	145
5.4	Discussion	146
5.4.1	Mind the Gap: The Dichotomy of Perceived Developer Needs and SAST Selection/Evaluation	146

5.4.2	The Power of Reputation and the Lack of Reliable Objective Criteria	147
5.4.3	Giving Developers What They Want	148
5.4.4	Industry is not prepared for the flaws of SASTs	148
5.4.5	Moving Forward: New Directions and Ideas	149
5.5	Chapter Summary	150
6	Identifying the Factors in the Lifecycle of Bugs in vulnerability detectors	151
6.1	Research Methodology	153
6.1.1	Identifying and Collecting Bug Reports	153
6.1.2	Analyzing Data	156
6.2	Analysis Results	158
6.2.1	The Balancing act of Vulnerability Alert and Alert Fatigue . .	159
6.2.2	Balancing Security-First Features and Developer Happiness . .	163
6.3	Threats to Validity	166
6.4	Chapter Summary	167
7	Future Work	168
8	Conclusion	169
	Bibliography	170
A	Appendix	215
A.1	Appendix of Chapter 3	215
A.2	Additional Evaluation Details	217
A.3	Crypto-API Misuse Taxonomy Data	218
A.4	Appendix	219
A.4.1	Code Snippets	219

A.4.2	Additional Implementation and Evaluation Details	222
A.4.2.1	Expanded rationale for choosing certain operators . . .	222
A.4.2.2	Do we need to optimize the number of mutants generated?	222
A.4.2.3	Further details regarding confirming killed mutants . .	224
A.4.2.4	Why GCS, LGTM, and QARK fail to detect base cases	224
A.4.3	Types of cases that our SLR approach may miss	225
A.4.4	Additional Evaluation Data	227
A.5	Survey Protocol	227
A.5.1	Survey Recruitment	228
A.5.2	Online Survey and Data Analysis	229
A.6	Survey Results	229
Vita		233

ACKNOWLEDGMENTS

This work is the culmination of efforts from many individuals who guided, mentored, tolerated, and assisted me throughout my Ph.D.

I learned about research, teaching, and mentoring from the best advisors possible, Dr. Adwait Nadkarni and Dr. Denys Poshyvanyk. Even when I struggled tremendously with health-related issues, both of them supported and encouraged me unwaveringly and pushed me as necessary. I am and will forever be grateful to them for that. As I begin my new journey in academia, their advice, work ethic, and values will continue to guide me. Additionally, the committee members, namely Dr. Yixuan Zhang, Dr. Robert Lewis, and Dr. Kevin Moran, provided me with valuable feedback to improve this work.

The labs directed by my advisors, Secure Platforms Lab (SPL) and Software Engineering Maintenance and Evolution Research Unit (SEMERU), were instrumental to my growth, offering both mentorship and opportunities to mentor others. Particularly, Kevin, Carlos, Michele, David, Nathan, Kaushal, Sunil, Prianka, Victor, and Akram made significant contributions to my development.

At the CS Department of William & Mary, the faculty members ensured that I felt welcomed and supported, in a country that is about eight thousand miles away from my home, Bangladesh. The staff members, especially Vanessa, Dale, and Jacquelyn, continue to support the entire CS department with the limited resources they have, and it only awes me even more. From them, I will carry on the value of resourcefulness.

I am thankful to my cohort in the CS department, namely Heather, Ken, Nathan, Yu, Ada, Wei, and Yang, who helped me navigate the unfamiliar territory of the USA and became my family away from home when I first arrived.

The Health Center of W&M, especially Dr. David Dafashy and Dr. Chris Massengill, supported me in surviving in the USA, literally. Whichever the health condition, be it mental or physical, they took care of me to ensure that I could get back to my normal, healthy state. From them, I will be carrying on the value of caring and living.

I am grateful to my parents, Sankar Nath Seal and Tinku Roy, for raising me and teaching me the values of honesty and relentlessness. The sacrifices they made and the lessons they taught shaped me into who I am today. My brother, Saumen Seal, taught me the value of silence and grit by forging his own path, which inspired me to forge my own.

I extend my thanks to the Bangladeshi community in Williamsburg for being there for me at my best and worst times. While the USA is vast and often lonely, my community was always there for me during my times of dire need. Similarly, a note of thanks to the research community for providing me with feedback, whether through acceptance or rejection, which has helped me grow and develop.

Agnika Prisha, my toddler daughter, brings the joy of living to my life. Her curiosity and insistence on learning, despite numerous failures, in mundane, daily tasks we take for granted, such as standing, walking, talking, and eating, inspire me to continue.

Finally, I want to thank Prianka Mandal, my wife. We have both learned and grown together throughout the years and she continues to choose, support, and love me in all possible ways, for which I am grateful to her. As we both continue our journey together, I hope we will feel at home wherever we are, together.

To the *Unknowable*, who continues to plan for the stars; like dust floating in the
currents of space, and us, like stars.

LIST OF TABLES

3.1	The number of leaks inserted by μ SE, and the final number marked as executable by μ SE’s EE. Note that the “-” indicates that the scheme is not applicable to a particular app, due to the app’s particular characteristics (<i>e.g.</i> , the absence of fragments)	46
3.2	Descriptions of flaws uncovered* in FlowDroid v2.0	65
3.3	Descriptions of flaws uncovered in Argus v3.1.2.	66
3.4	Descriptions of flaws uncovered in HornDroid.	67
3.5	Impact of Operator Placement Approaches in Finding Flaws.*	67
3.6	Analysis of the propagation of flaws in FlowDroid 2.0 to other data leak detectors.*	68
3.7	Analysis of the propagation of flaws in Argus 3.1.2 to other data leak detectors.* .	68
3.8	Analysis of the propagation of flaws in HornDroid to other data leak detectors.* .	68
4.1	Descriptions of Flaws discovered by Analyzing crypto-detectors	94
4.2	Descriptions of Flaws discovered by Analyzing crypto-detectors in Current Iteration	100
4.3	Flaws observed in crypto-detectors in previous iteration	115
4.4	Flaws observed in crypto-detectors in current iteration	116
5.1	Overview of interviewed participants, position(s), product area(s), security priority from the perspectives of participants and project meta-data. Fine-level details are binned to ensure the anonymity of the participants.	128

6.1	Breakdown of Repositories, the total number of issues, and statistically significant sample size for 95% confidence as of February 2024	156
A.1	List of App names, URLs and IDs assigned by us for the purpose of the μ SE study	216
A.2	List of Applications mutated using MASC for the current study, CLOC = Count Lines of Code from Java Source files only [68], Source = Source Code collected from/Originated From	217
A.3	Relevance of crypto-detectors evaluated using MASC for the Extended Study . .	217
A.4	Selected Sources for Extracting Cryptography Misuse, from Academia from 2019-2022	218
A.5	Mutants analyzed vs detected by crypto-detectors	225
A.6	List of Applications mutated using MASC in previous study (S&P'22 [19]), CLOC = Count Lines of Code from Java Source files only [68], Source = Source Code collected from/Originated From	227
A.7	Abridged version of the Semi-structured Interview Guide	232

LIST OF FIGURES

3.1	μ SE tests a static analysis tool on a set of mutated Android apps and analyzes uncaught mutants to discover and/or fix flaws.	28
3.2	The components and process of the μ SE.	30
3.3	A generic “network export” security operator, and its more fine-grained instanti- ations in the context of FlowDroid [32] and MalloDroid [100].	30
3.4	The number of mutants (<i>e.g.</i> , data leaks) to analyze drastically reduces at every stage in the process.	38
4.1	A conceptual overview of the MASC framework.	75
4.2	The derived taxonomy of cryptographic misuses. (n) indicates misuse was present across n artifacts. A \checkmark indicates that the specific misuse case was instantiated with MASC’s mutation operators for our evaluation (Sec. 4.6). New misuse cases found as part of this extension are highlighted in Blue.	76
4.3	Architecture Overview of the Main Scope of MASC	88
5.1	Overview of Semi-structured Interview Guide. We used Laddering technique to delve deeper in each topic, while the semi-structured approach helped us to freely deviate as necessary based on participant response.	125

Systematically Evaluating Vulnerability Detectors

-

Discovering the Gaps within Design and Practice

Chapter 1

Introduction

After over a decade of research and development in both academia and industry, security-focused program-analysis techniques are now being used at nearly every stage of software development and maintenance lifecycle, from requirements engineering to fault localization and fixing, *e.g.*, through GitHub CodeScan Initiative [119] for finding vulnerabilities, such as crypto-API misuse and sensitive data-leaks. Furthermore, such techniques have gained worldwide attention because of the recent high-profile attacks and exploit across the public sector *e.g.*, SolarWind [222], triggering responses from both corporate and government entities, such as emphasizing security through the improvement of existing approaches, *e.g.*, Static Application Security Testing (SAST) tools. In essence, the use and dependence on program-analysis techniques will only increase to ensure software security.

The underlying cause of optimistically using and depending on the security-focused analysis techniques, such as crypto-API misuse detectors (in short, crypto-detectors) and sensitive data-leak detectors, is the convenience these techniques offer through automation, support for continuous integration and development (CI/CD), and the promise of detecting vulnerabilities as long as these are within scope without flaws. In addition, the potential

IEEE Copyright Note: In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of the College of William and Mary's products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.

of identifying vulnerabilities statically, *i.e.*, without depending on runtime-analysis in a time-consuming manner, has made the SAST-based tools an attractive choice among the security-focused program-analysis techniques.

However, such optimism is unwarranted, as we have generally relied on manually curated, static benchmarks to gauge the effectiveness of SASTs. This is also because of the lack of a systematic framework that can handle the various patterns of vulnerabilities, *i.e.*, variants. Considering these, the thesis statement of this work is the following:

Thesis Statement: For the design, evaluation, and application of SASTs such that they are effective at detecting vulnerabilities, it is imperative to systematically identify the design and implementation flaws in them, as well as the factors influencing the flaws.

In proving this thesis, we seek answers to the following research questions in this dissertation.

RQ1: How to systematically find flaws in SASTs while considering the diverse practices adopted in the wild? This doctoral research proposes the systematic evaluation of SASTs, susceptible data-leak detectors, and crypto-API misuse detectors while leveraging the well-founded approach of mutation analysis.

While traditional mutation analysis is used to gauge the effectiveness of existing test cases, this thesis advocates that vulnerabilities can be mutated to represent both the diverse variations of vulnerabilities that are implemented and introduced by developers, either intentionally or unintentionally and the complex usage patterns of relevant security-specific APIs, such as crypto-primitives enabling APIs from language-specific frameworks. We propose that we can systematically evaluate, analyze, and identify flaws in SASTs by introducing mutated vulnerabilities in open source program source code, which, then, is analyzed by a target SAST. This research analyzes both sensitive-data leak detectors and crypto-detectors from industry and academia. We identify previously unknown, unique flaws while gaining insights, such as possible causes and remedies.

RQ2: What perceptions and beliefs of practitioners about SASTs and their

flaws impact the effective application of SASTs? In addition to this, this work identifies a key gap in the adoption and use of SASTs in the software industry: the research community does not possess an in-depth understanding of how SASTs are perceived in the industry, *e.g.*, whether practitioners are aware of the flaws, or limitations, these SASTs may have, and whether such perceptions and beliefs impact the adoption and use of SASTs. Without addressing this critical gap, it is impossible to create SASTs that are effective, *i.e.*, possess fundamental properties that help ensure software security, and aligned *i.e.*, practitioners have an accurate understanding of what SASTs provide, instead of practitioners having an inaccurate expectation of, and from SASTs.

Therefore, to explore this gap through a qualitative, interview-based study in this research and report how practitioners with different security and business needs choose SASTs and depend on those. Furthermore, we study the beliefs and expectations of practitioners regarding the limitations and flaws of SASTs, and how they address those flaws of SASTs.

1.1 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 provides background and discusses related work of this dissertation. In Chapter 3, we contextualize mutation testing techniques to introduce mutation operators and mutation schemes by proposing and implementing a framework, namely μ SE. μ SE is then used for systematically evaluating prominent, well-known static analysis based data-leak detectors. In addition to identifying soundness issues in these tools, we explore how the found flaws may propagate across tools based on various factors, such as dependency and similar design principles/ethos.

In Chapter 4, we look at another application of static analysis for security, specifically for cryptographic API misuse detection, or in short, crypto-detectors. By taking an in-depth look at prominent crypto-detectors from industry, academia, and open source community, we contextualize mutation testing techniques for systematically evaluating

crypto-detectors. As part of this, we start by creating a taxonomy of crypto-API misuse by analyzing sources from industry and academia. Furthermore, we introduce novel concepts of mutation scope and crypto-API specific mutation operators, using which we evaluate prominent crypto-detectors from industry, academia, and open source community, with a discussion about the current technique-centric approach adopted by crypto-detectors, and the security-centric approach the community needs to adapt to address the evolving needs of software engineers.

In Chapter 5, we argue that there is a gap between the SAST developer community and the user community that needs to be addressed. To develop SASTs that are effectively leveraged by developers for finding vulnerabilities, researchers and tool designers must understand how developers perceive, select, and use SASTs, what they expect from the tools, whether they know of the limitations of the tools, and how they address those limitations. In this chapter, we describe a qualitative study that explores the assumptions, expectations, beliefs, and challenges experienced by developers who use SASTs. We perform in-depth, semi-structured interviews with 20 practitioners who possess a diverse range of software development expertise, as well as a variety of unique security, product, and organizational backgrounds. We identify key findings that shed light on developer perceptions and desires related to SASTs and also expose gaps in the status quo — challenging long-held beliefs in SAST design priorities. Finally, we provide concrete future directions for researchers and practitioners rooted in an analysis of our findings.

Finally, in chapter 6, we argue that the lifecycle of bugs in SASTs, that compromise the offered security guarantees, are influenced by external/internal factors, such as implicit assumptions made by tool designers and availability of data for bug reproduction. Furthermore, such cases must be aligned with the existing design goals and threat model defined by the SAST developers, which are often unknown or not communicated with the SAST users. As a result, a soundness compromising bug that is still within the technical scope (e.g., statically analyzable) of a SAST may still be considered “out-of-scope” by SAST developers. Therefore, we perform a qualitative study by sampling representative

bugs from open source SASTs. Furthermore, we provide details on qualitatively analyzing them to identify those implicit factors to further reduce the gap between the developers and users of the SASTs.

Chapter 2

Background and Related Work

2.1 Soundness and Static Analysis Tools

This work is motivated by the pressing need to help researchers and practitioners identify instances of unsound assumptions or design decisions in their static analysis tools, thereby *extending the sound core* of their soundy techniques. That is, security tools may already have a core set of sound design decisions (*i.e.*, the sound core) and may claim soundness based on those decisions. The soundness manifesto [190] defines the *sound core* in terms of specific language features, whereas we use the term in a more abstract manner to refer to the design goals of the tool. Systematically identifying unsound decisions may allow researchers to resolve flaws and help extend the sound core of their tools.

Moreover, research papers and tool documentations indeed do not articulate many of the unsound assumptions and design choices that lie outside their sound core, aside from some well-known cases (*e.g.*, choosing not to handle reflection, race conditions), as confirmed by these results (Section 3.4). In addition, there is a chance that developers of these techniques may be unaware of some implicit assumptions/flaws due to a host of reasons: *e.g.*, because the assumption was inherited from prior research or a certain aspect of Android was not modeled correctly. Therefore, the objective is to discover instances of such hidden assumptions and design flaws that affect the security claims made by tools,

document them explicitly, and possibly, help developers mend existing artifacts.

2.1.1 Motivating Example

Consider the following motivating example of a prominent static analysis tool, FlowDroid [32]:

FlowDroid is a highly popular static analysis framework for detecting private data leaks in Android apps by performing a data flow analysis. Some of the limitations of FlowDroid are well-known and stated in the paper [32]; *e.g.*, FlowDroid does not support reflection, like most static analyses for Java. However, through a manual but systematic, preliminary, analysis of FlowDroid, we discovered a security limitation that is not well-known or accounted for in the paper, and hence affects guarantees provided by the tool’s analysis. We discovered that FlowDroid (*i.e.*, v1.5, which was latest at the time of the preliminary analysis in October 2017) does not support “Android fragments” [26], which are app modules that are widely used in most Android apps (*i.e.*, in more than 90% of the top 240 Android apps per category on Google Play, as demonstrated in the original USENIX’18 paper [50]). This flaw renders any security analysis of general Android apps using FlowDroid unsound, due to the high likelihood of fragment use, even when the app developers may be cooperative and non-malicious. Further, FlowDroid v2.0, which was released on 10/10/2017 [279], claims to address fragments, *but failed to detect the exploit*.

On investigating further, we found that FlowDroid v1.5 was extended or used by at least 13 research tools [180, 172, 305, 37, 218, 252, 188, 259, 6, 244, 179, 182, 210], none of which acknowledge or address this limitation in modeling fragments. This leads us to conclude that this significant flaw not only persists in FlowDroid, but may have additionally propagated to the tools that inherit it directly (*i.e.*, by inheriting its code), or indirectly (*i.e.*, by adhering to similar design principles/goals). The flaw propagation study in Section 3.7 confirms this conjecture for inheritors of FlowDroid, as well as the two other data leak detectors that we evaluate in-depth, *i.e.*, Argus [105] and Horndroid [59].

We reported the flaws to the authors of FlowDroid, and created two patches to fix

it. The patches were confirmed to work on FlowDroid v2.0 built from source, and were accepted into FlowDroid’s repository [281] in December 2017. Thus, we were able to discover and fix an undocumented flaw that significantly affected FlowDroid’s soundness claims, thereby expanding its sound core. However, we have confirmed that FlowDroid v2.5 [279] still fails to detect leaks in fragments, and are working with developers to resolve this issue.

Through this example, we demonstrate that unsound assumptions in security-focused static analysis tools for Android are not only detrimental to the validity of their own analysis, but may inadvertently propagate to future research. Thus, identifying these unsound assumptions is not only beneficial for making the user of the analysis aware of its true limits, but in addition for the research community in general. As of today, aside from a handful of manually curated testing tool-kits (*e.g.*, DroidBench [32]) with hard-coded (but useful) checks there has been no prior effort at methodologically discovering problems related to soundness in Android static analysis tools and frameworks. *This work is motivated by the need to systematically identify and resolve the unsound assumptions in security-related static analysis tools.*

2.1.2 Background on Mutation Analysis

Mutation analysis has a strong foundation in the field of SE, and is typically used as a test adequacy criterion, measuring the effectiveness of a set of test cases [224]. Faulty programs are created by applying transformation rules, called *mutation operators* to a given program. The larger the number of faulty programs or *mutants* detected by a test suite, the higher the effectiveness of that particular suite. Since its inception [142, 78], mutation testing has seen striking advancements related to the design and development of advanced operators. Research related to development of mutation operators has traditionally attempted to adapt operators for a particular target domain, such as the web [239], data-heavy applications [27, 308, 85], or GUI-centric applications [225]. Recently, mutation analysis has been applied to measure the effectiveness of test suites for both functional

and non-functional requirements of Android apps [79, 162, 184].

This chapter builds upon SE concepts of mutation analysis and adapts them to a security context. The methodology does not simply use the traditional mutation analysis, but rather *redefines* it to effectively improve security-focused static analysis tools, as described in Sections 3.2 and 3.3.

2.1.3 Related Work on Mutation Analysis and Data Leak Detectors

μ SE builds upon the theoretical underpinnings of mutation analysis from SE, and to our knowledge, is the first work to adapt mutation analysis to evaluate the soundness claimed by security tools. Moreover, μ SE adapts mutation analysis to security, and makes fundamental and novel modifications (described previously in Section 3.2). We now describe prior work in three other related areas:

Formally Verifying Soundness: While an ideal approach, formal verification is one of the most difficult problems in computer security. For instance, prior work on formally verifying apps often requires the monitor to be rewritten in a new language or use verification-specific programming constructs (*e.g.*, verifying reference monitors [110, 289], information flows in apps [204, 205, 304]), which poses practical concerns for tools based on numerous legacy codebases (*e.g.*, FlowDroid [32], CHEX [191]). Further, verification techniques generally require correctness to be specified, *i.e.*, the policies or invariants that the program is checked against. Concretely defining what is “correct” is hard even for high-level program behavior (*e.g.*, making a “correct” SSL connection) and may be infeasible for complex static analysis tools (*e.g.*, detecting “all incorrect SSL connections”). μ SE does not aim to substitute formal verification of static analysis tools; instead, it aims to uncover existing limitations of such tools.

Evaluating Static Analyses: Recently, there has been significant work in the area of experimentally evaluating the features and effectiveness of static analysis techniques. For instance, Qiu *et al.* [241] performed a comparative evaluation of precision and runtime per-

formance, among FlowDroid+IccTA, AmanDroid and DroidSafe, by using a common configuration setup, using a benchmark that extends DroidBench [91] and ICC-Bench [154]. Pauck *et al.* [236] propose the ReproDroid framework that automatically evaluates the effectiveness of Android taint analysis tools using user-labeled ground truth in Android apps. However, there are fundamental differences in our work, and these related approaches, in terms of the primary goal, scope, and the actual techniques leveraged.

To elaborate, μ SE focuses on exhaustively generating security test cases for evaluating Android security tools, and systematically performing in-depth evaluations of such tools to discover gaps in the soundness that directly affect their ability to detect security vulnerabilities such as data leaks. μ SE’s security focus is evident in our additional efforts towards designing security-focused mutation, and attributing undetected mutants to actual flaws and design choices in tools that affect security. On the contrary, both prior approaches focus on evaluating either the presence of promised static analysis *features*, or the general precision, *i.e.*, with a lack of specific focus on security. Furthermore, μ SE is a holistic, automated, mutation framework that generates thousands of expressive, *security-goal-focused test cases* for evaluating security tools, while related work generally relies on handcrafted benchmarks [241] or user-specified ground-truth [236], which may be sufficient for evaluating features, but not for a thorough security evaluation of tools. That is, μ SE *empowers* security researchers to discover flaws without delving into the intricate details of program analysis techniques. However, we do note that certain aspects of prior work (*e.g.*, the automated bootstrapping of tools in ReproDroid [236]) are complementary to μ SE, and may be incorporated into its pipeline in the future. Finally, while μ SE evaluates soundness, our mutation-based approach may be used to evaluate precision as well, which is a separate research direction that we aim to explore in the future. The popularity and open-source nature of Android has spurred an immense amount of research related to examining and improving the security of the underlying OS, SDK, and apps. Recently, Acar *et al.* have systematized Android security research [7], and we discuss work that introduces static analysis-based countermeasures for Android security issues according to

Acar *et al.*'s categorization.

Perhaps the most prevalent area of research in Android security has concerned the permissions system that mediates access to privileged hardware and software resources. Several approaches have motivated changes to Android's permission model, or have proposed enhancements to it, with goals ranging from detecting or fixing unauthorized information disclosure or leaks in third party applications [96, 32, 115, 209, 208, 303, 166] to detecting over privilege in applications [103, 35, 291]. Similarly, prior work has also focused on benign but vulnerable Android applications, and proposed techniques to detect or fix vulnerabilities such as cryptographic API misuse API [100, 93, 266, 101] or unprotected application interfaces [104, 66, 177]. Moreover, these techniques have often been deployed as modifications to Android's permission enforcement [97, 96, 212, 109, 86, 104, 57, 227, 72, 311, 260, 58, 144, 237, 254], SDK tools [103, 35, 291], or inline reference monitors [302, 165, 43, 42]. While this work demonstrates the evaluation of only a small subset of these tools with μ SE, our experiments demonstrate that μ SE has the potential to impact nearly all of them. For instance, we can apply μ SE to vet SSL analysis tools by purposely introducing complex SSL errors in applications, or privilege or permission misuse analysis tools, by developing security operators that attempt to misuse permissions.

2.2 Static Analysis for Crypto-API Misuse Detection

Insecure use of cryptographic APIs is the second most common cause of software vulnerabilities after data leaks [290]. To preempt vulnerabilities before software release, non-experts such as software developers or quality assurance teams are likely to use crypto-API misuse detectors (or *crypto-detectors*) as a part of the Continuous Integration/Continuous Delivery (CI/CD) pipeline (*e.g.*, Xanitizer [300] and ShiftLeft [257] used in GitHub Code Scan [120]), quality assurance suites (*e.g.*, SWAMP [282]) or IDEs (*e.g.*, CogniCrypt [176]). Thus, the inability of a crypto-detector to flag an instance of a misuse that it claims to detect directly impacts the security of end-user software. We illustrate this problem with

a motivating example, followed by a threat model that describes the potential adversarial conditions a crypto-detector may face in the wild.

2.2.1 Motivating Example

Consider Alice, a Java developer who uses CryptoGuard [243], a state-of-the-art crypto-detector, for identifying cryptographic vulnerabilities in her software before release. In one of her apps, Alice decides to use the DES cipher, as follows:

```
1 Cipher cipher = Cipher.getInstance("des");
```

Listing 2.1: Instantiating DES as a cipher instance in lower case.

This is another instance of the misuse previously shown in Listing 4.1, *i.e.*, using the vulnerable DES cipher. *CryptoGuard* is unable to detect this vulnerability as Alice uses “des” instead of “DES” as the parameter (see Section 4.7). However, this is a problem, because the lowercase parameter makes no functional difference as Java officially supports both parameter choices. As *CryptoGuard* does not detect this vulnerability, Alice will assume that her app is secure and release it to end-users. Thus, we need to systematically identify such flaws, which would allow the maintainers of crypto-detectors such as *CryptoGuard* to promptly fix them, enabling holistic security improvements.

2.2.2 Threat Model

To evaluate crypto-detectors, we first define the *scope* of our evaluation, for which we leverage the documentation of popular crypto-detectors to understand how they position their tools, *i.e.*, what use cases they target (see [15] for all quotes). For example, Toolx’s documentation states that it may be used to “ensure compliance with security and coding standards” . Similarly, SpotBugs’s *Find Security Bugs* plugin is expected to be used for “security audits” [268]. Further, CogniCrypt states that its analyses “ensure that all usages of cryptographic APIs remain secure” [175], which may suggest the ability to detect vulnerabilities in code not produced by the developer, but originating in a third-party source (*e.g.*, external library, or a contractor), whose developer may not be

entirely “virtuous”. In fact, 8/9 crypto-detectors evaluated in the S&P’22 [19] paper, in addition to all the crypto-detectors evaluated in this extended study claim similar cases that demand strong guarantees, *i.e.*, for tasks such as compliance auditing or security assurance that are generally expected to be performed by an *independent* third party that assumes the worst, including bad coding practices or malpractice [24]. As aptly stated by Anderson [24], “*When you really want a protection property to hold, it’s vital that the design and implementation be subjected to hostile review*”.

Thus, given that crypto-detectors claim to be useful for tasks such as compliance audits, it is likely for them to be deployed in adversarial circumstances, *i.e.*, where there is tension between the party that uses a crypto-detector for evaluating software for secure crypto-use (*e.g.*, markets such as Google Play, compliance certifiers such as Underwriters Laboratories (UL) [160]), and the party implementing the software (*e.g.*, a third-party developer). With this intuition, we define a threat model consisting of three types of adversaries (**T1** – **T3**), which guides/scopes our evaluation according to the conditions crypto-detectors are likely to face in the wild:

T1 *Benign developer, accidental misuse* – This scenario assumes a benign developer, such as Alice, who accidentally misuses crypto-API, but attempts to detect and address such vulnerabilities using a crypto-detector before releasing the software.

T2 *Benign developer, harmful fix* – This scenario also assumes a benign developer such as Alice who is trying to address a vulnerability identified by a crypto-detector in good faith, but ends up introducing a new vulnerability instead. For instance, a developer may not fully understand the problem identified by a crypto-detector, such as missing certificate verification (*e.g.*, an empty `checkServerTrusted` method in a custom `TrustManager`), and address it with an inadequate Stack Overflow fix [272].

T3 *Evasive developer, harmful fix* – This scenario assumes a developer whose goal is to finish a task as quickly or with low effort (*e.g.*, a third-party contractor), and is hence attempting to purposefully evade a crypto-detector. Upon receiving a vulnera-

bility alert from a crypto-detector, such a developer may try quick-fixes that do not address the problem, but simply hide it (*e.g.*, hiding the vulnerable code in a class that the crypto-detector does not analyze).

For example, Google Play evaluates apps by third-party developers to ensure compliance with its crypto-use policies, but there is ample evidence of developers seeking to actively violate these policies [277, 276]. In fact, as Oltrogge et al. [226] recently discovered that developers have been using Android’s Network Security Configurations (NSCs) to *circumvent safe defaults* (*e.g.*, to permit cleartext traffic that is disabled by default).

This threat model, which guides MASC’s design (Section 4.1), represents that adversarial conditions under which crypto-detectors may have to operate in practice, and hence, motivates an evaluation based on what crypto-detectors *should be* detecting. However, we note that there may be a gap between *what should be* and *what is*, *i.e.*, while crypto-detectors may want to be relevant in strong deployment scenarios such as compliance checking, their actual design may not account for adversarial use cases (*i.e.*, **T3**). Therefore, we balance our evaluation that uses this threat model with a discussion that acknowledges all views related to this argument, and especially the tool designer’s perspective (Sec. 4.9).

2.2.3 Related Work on Crypto-API Misuse Detection

Security researchers have recently shown significant interest in the external validation of static analysis tools [241, 91, 154, 236, 51]. Particularly, there is a growing realization that static analysis security tools are sound in theory, but *soundy in practice*, *i.e.*, consisting of a core set of sound decisions, as well as certain strategic unsound choices made for practical reasons such as performance or precision [190]. Soundy tools are desirable for security analysis as their sound core ensures sufficient detection of targeted behavior, while also being *practical*, *i.e.*, without incurring too many false alarms. However, given the lack

of oversight and evaluation they have faced so far, *crypto-detectors* may violate this basic assumption behind soundness and may in fact be *unsound*, *i.e.*, have fundamental flaws that prevent them from detecting even straightforward instances of crypto-API misuse observed in apps. This intuition drives our approach for systematically evaluating crypto-detectors, leading to novel contributions that deviate from related work.

To the best of our knowledge, MASC is the first framework to use mutation testing, combined with a large-scale data-driven taxonomy of crypto-API misuse, for comprehensively evaluating the detection ability of crypto-detectors to find design/implementation flaws. However, in a more general sense, Bonett et al. [51] were the first to leverage the intuition behind mutation testing for evaluating Java/Android security tools, and developed the μ SE framework for evaluating data leaks detectors (*e.g.*, FlowDroid [32] and Argus [105]). MASC significantly deviates from μ SE in terms of its design focus, in order to address the unique challenges imposed by the problem domain of crypto-misuse detection (*i.e.*, $\mathbf{RC}_1 - \mathbf{RC}_3$ in Sec. 4). Particularly, μ SE assumes that for finding flaws, it is sufficient to *manually* define “a” security operator and strategically place it at hard-to-reach locations in source code. This assumption does not hold when evaluating crypto-detectors as it is improbable to cast cryptographic misuse as a single mutation, given that cryptographic misuse cases are diverse (\mathbf{RC}_1), and developers may express the same type of misuse in different ways (\mathbf{RC}_2). For example, consider three well-known types of misuse that would require unique mutation operators: (1) using DES for encryption (operator inserts prohibited parameter names, *e.g.*, DES), (2) trusting all SSL/TLS certificates (operator creates a malformed `TrustManager`), and (3) using a predictable initialization vector (IV) (operator derives predictable values for the IV). In fact, developers may even express the same misuse in different ways, necessitating unique operators to express such distinct *instances*, *e.g.*, the DES misuse expressed differently in Listing 4.1 and Listing 2.1. Thus, instead of adopting μ SE’s single-operator approach, MASC designs general usage-based mutation operators that can expressively instantiate misuses from our taxonomy of 109 misuses. In a similar manner, MASC’s contextualized mutation abstractions (*i.e.*, for evaluating crypto-

detectors) distinguish it from other systems that perform vulnerability injection for C programs [89], discover API misuse using mutation [295, 114], or evaluate static analysis tools for precision using handcrafted benchmarks or user-defined policies [241, 236].

Finally, the goal behind MASC is to assist the designers of crypto-detectors [100, 93, 71, 175, 176] in identifying design and implementation gaps in their tools, and hence, MASC is complementary to the large body of work in this area. Particularly, prior work provides rule-sets or benchmarks [54, 55] consisting of a limited set of cryptographic “bad practices” [53], or taxonomies of smaller subsets (*e.g.*, SSL/TLS misuse taxonomy by Vasan et al. [211]), or examines the precision of crypto-detectors [64]. However, we believe that ours is the first systematically-driven and comprehensive taxonomy of *crypto-API* misuse, which captures 109 cases that are further expanded upon into numerous unique misuse instances through MASC’s operators. Thus, relative to prior handcrafted benchmarks, MASC can thoroughly test detectors with a far more comprehensive set of crypto-misuse instances.

2.3 Static Analysis Tools for Security in Practice

SASTs have been adapted for finding security vulnerabilities [65], resource leaks [31, 293, 219, 59, 30, 173, 41], enforcing policies [33], and crypto-API misuse [243, 71, 175, 100, 93, 53, 213, 307]. Our work studies the perspectives and beliefs of practitioners regarding SASTs through 20 in-depth interviews, and is closely related to work in three areas, namely prior studies on the usability of static analysis tools, research on evaluating SASTs, and the study of general security practices in industry.

Usability of Static Analysis Tools: Researchers have studied how practitioners use static analysis tools and their perspectives on improving the output of static analysis tools [40, 39, 38, 95, 47, 87]. In particular, Johnson *et al.* [167] found that poorly presented output, including false positives, is one of the main problems from developers’ perspective. This has been confirmed by later studies [67], where they suggested that the false positive

rate should be around or below 20%. In a similar vein, Distefano *et al.* [87] recognized that while false negatives matter, it is difficult to quantify false negative rate compared to false positive rate, and thus, it is prudent to focus on optimizing the latter. Our work complements existing literature by detailing how practitioners across the industry choose and perceive SASTs (\mathcal{F}_4 , \mathcal{F}_5 , \mathcal{F}_7). However, our qualitative findings deviate from prior work (in the context of security), *i.e.*, we find that developers prefer low false negatives, and are willing to tolerate high false positive rates (\mathcal{F}_{10} , \mathcal{F}_{11}) provided the tool detects vulnerabilities.

Further, recent work [39, 38, 249] proposes allowing subjective interpretations of defect warnings for productivity, *i.e.*, the notion of “effective false positives”, which has been adapted in the context of security by Wickert *et al.* [296]. We study whether this concept works in the context of security, *i.e.*, whether practitioners see merit in letting developers decide what constitutes a vulnerability, and find that it does not, drawing attention to the risks of letting developers become the arbiters of false positives (\mathcal{F}_{12}).

Finally, prior work has focused on particular usability aspects of static analysis tools, such as the use cases and their contexts [214, 288, 206], and filtering warning messages [156]. Our analysis of the pain points experienced by developers echoes some of these concerns (\mathcal{F}_{17}), highlights unique challenges practitioners face (\mathcal{F}_4 , \mathcal{F}_5 , \mathcal{F}_{15}), and culminates in our discussion of a path forward for both researchers and practitioners towards better and more useful SASTs.

Evaluating SASTs: Historically, security researchers have focused on creating benchmarks for evaluating SASTs with a focus on precision, recall, and efficiency, with the help of benchmarks, such as ICC-Bench [293], DroidBench [111], CryptoAPI-Bench [12], ApacheCryptoAPIBench [13], OWASP Benchmark [4], Parametric Crypto Misuse Benchmark [297], Ghera [198], and CamBench [253]. Moreover, prior work has proposed automated evaluation using benchmarks [236, 192]. Our study reveals that practitioners, in general, do not trust third-party benchmarks, due to them being basic (*i.e.*, not repre-

sentative of real vulnerabilities), or worse, biased in favor of a specific SAST tool. Since SAST tools often accompany custom benchmarks claimed to be general (as is the case with many of the benchmarks above), we cannot deny this perception.

A recent body of research considers the limitations of existing benchmarks and leverages mutation testing to uncover flaws in the detection capabilities of SASTs [51, 21, 19, 20, 17]. As we discuss in Section 5.4, enabling better benchmarks or automated evaluation of SASTs using such evolving approaches may be a path forward towards providing developers with what they care most, *i.e.*, SASTs that can detect real, valuable, vulnerabilities.

Study of Security Practices in Industry: Finally, prior work has studied the relationship between developers’ security expertise, and the actual implementation of secure software. For example, researchers have identified that practitioners need security-specific support in the form of developer-friendly APIs and supporting tools [10, 137, 136, 207, 261] to make better security choices, recognizing that practitioners may not know enough about security [139, 299, 67]. Furthermore, researchers have explored how developers address security-specific tasks and the challenges they face [8, 140, 143, 280, 164]. Our work complements previous studies by exploring how developers choose SASTs ($\mathcal{F}_4, \mathcal{F}_5$). Moreover, we explore how developers depend on SASTs to cover their knowledge gaps (\mathcal{F}_6) and the challenges, beliefs, and perceptions associated with implementing security in software with the help of SASTs ($\mathcal{F}_8\text{--}\mathcal{F}_{17}$).

2.4 Bugs in Vulnerability detectors

Traditional Bug detection tools are repurposed for security centric evaluation/vulnerability detection, even though they originated to function as developer friendly tools. Because these vulnerability detectors tend to serve multiple roles, *i.e.*, finding vulnerabilities *e.g.*, crypto-API misuse detection [53, 93, 100, 175, 176, 213, 243] and data leak detection [30, 32, 41, 59, 152, 173, 221, 294] and reporting non-security issues, such as code quality, this results in overlapping or conflicting interests, expectations, and assumptions made by both

tool developers and tool users. This mismatch becomes apparent when users submit an issue, such as experiencing unexpected behavior or feature requests; developers recognize these as in-scope issues or reject the issues, prioritize accepted issues, and address and deliver solutions of the issues.

Since Vulnerability detectors are just like any software when it comes to addressing their bugs, we need to grasp an understanding of the lifecycle of bugs in software systems in addition to how bugs in Vulnerability detectors are being addressed in the state of the art.

Bugs and Their Lifecycle: User reported issues are important to software practitioners as they are considered as helpful feedback for improving software systems. However, the quality of bug reports can differ greatly, and this can cause delays in fixing issues. The life cycle of a bug usually include a number of steps: initially, the issue is reported and found by a user or developer, preferably with a lot of information such as reproduction steps, anticipated and actual behavior, and stack traces [312]. After bugs have been reported, they are triaged. During the bug traiging, they are reviewed, prioritized, and assigned to developers to be fixed [250]. This process can be complex and time-consuming, especially when reports are unclear or incomplete [312, 250]. Following triaging, developers try to reproduce the bug to understand the possible causes, and figure out fixes [250, 62]. Not all reported bugs are resolved. A majority are “won’t fix” because they are out of scope, not reproducible, or do not have sufficient information. Others remain open for months pending when developers will mark them as “won’t fix.” That means not all the issues are similarly prioritized in software development. [235]. Moreover, bugs can be reopened, which causes loops in their life cycle. It reflects the issues in bug-fixing and tracking practices [99]. Several characteristics, such as bug type, developer experience, and quality of communication among the developers, have a significant effect on the time to resolve or close bug reports [44, 52]. Automated tools have been proposed to improve bug tracking, classification, prioritization, and even predicting whether issues will be resolved

or abandoned. The developers of these tools hope to make the process more effective and save developers time dealing with bad reports [235, 247, 306, 28, 49, 234, 61, 313]. Others cite that bug fixing is a socio-technical practice that consists of technical, social, and organizational dimensions. For this reason, complete automation is difficult but desired for improving software quality [28, 250, 52].

Security Bug Management: Effectively managing the security bug is critical to maintaining the security of the software. For this reason, the process of vulnerability discovery and vulnerability disclosure are both important.

Recent studies show that organizations use a mix of specialized internal teams, third-party contractors, and bug bounty programs to find vulnerabilities. But they still face problems like communication gaps, resource constraints, and trust issues [14]. The workforce itself lacks diversity, where the marginalized groups face unique and complex barriers to participation [9]. There are many ways to find vulnerabilities. For example, using diagnostic tools, reviewing code, creating malicious inputs etc. Although testers and hackers may use similar methods, their results often differ due to their skills and experience [292, 48]. Automation and machine learning have made discovery of vulnerabilities more flexible as well as scalable, primarily at function as well as patch levels. But misleading patterns as well as unexpected side effects still make such methods unreliable [113, 155, 183].

After vulnerability discovery, the next step is to disclose the vulnerability properly. Vulnerability disclosure is done in multiple ways. Among all, Coordinated Vulnerability Disclosure (CVD) is most commonly thought to be standard because it facilitates coordination between researchers, vendors, as well as stakeholders [202, 148]. However, CVD is difficult to implement for reasons such as difficulties in contacting appropriate stakeholders [63]. Moreover, CVD is also stressful and time-consuming for researchers, as they might face ethical dilemmas, communication complexities, etc. [202]. To solve these problems, automated solutions like Automated Responsible Disclosure (ARD) have been introduced

[187]. But still some problems exists like early discussion of vulnerabilities even before their disclosure [189].

Chapter 3

Evaluating Data Leak Detectors using Mutation

Mobile devices such as smartphones and tablets have become the fabric of our consumer computing ecosystems; by the year 2020, more than 80% of the world’s adult population is projected to own a smartphone [92]. This popularity of mobile devices is driven by the millions of diverse, feature-rich, third-party applications or “apps” they support. However, in fulfilling their functionality, apps often require access to security and privacy-sensitive resources on the device (*e.g.*, GPS location, security settings). Applications can neither be trusted to be well-written or benign, and to prevent misuse of such access through malicious or vulnerable apps [185, 138, 309, 245, 100, 266, 93], it is imperative to understand the challenges in securing mobile apps.

Security analysis of third-party apps has been one of the dominant areas of smartphone security research in the last decade, resulting in tools and frameworks with diverse security goals. For instance, prior work has designed tools to identify malicious behavior in apps [97, 310, 29], discover private data leaks [96, 32, 115, 37], detect vulnerable application interfaces [104, 66, 191, 177], identify flaws in the use of cryptographic primitives [100, 93, 266], and define sandbox policies for third-party apps [145, 163]. To protect users from malicious or vulnerable apps, it is imperative to assess the challenges and pit-

falls of existing tools and techniques. However, *it is unclear whether existing security tools are sufficiently robust to expose particularly well-hidden unwanted behaviors.*

Our work is motivated by the pressing need to discover the limitations of application analysis techniques for Android. Existing application analysis techniques, specifically those that employ static analysis, must in practice trade soundness for precision, as there is an inherent conflict between the two properties. A sound analysis requires the technique to over-approximate (*i.e.*, consider instances of unwanted behavior that may not execute in reality), which in turn deteriorates precision. This trade-off has practical implications on the security provided by static analysis tools. That is, *in theory*, static analysis is expected to be sound, yet, in practice, these tools must purposefully make unsound choices to achieve a feasible analysis that has sufficient precision and performance to scale. For instance, techniques that analyze Java generally do not over-approximate analysis of certain programming language features, such as reflection, for practical reasons (*e.g.*, Soot [287], FlowDroid [32]). Although this particular case is well-known and documented, many such unsound design choices are neither well-documented, nor known to researchers outside a small community of experts.

Security experts have described such tools as *soundy*, *i.e.*, having a core set of sound design choices, in addition to certain practical assumptions that sacrifice soundness for precision [190]. Although soundness is an elusive ideal, *soundy* tools certainly seem to be a practical choice: *but only if the unsound choices are known, necessary, and properly documented.* However, the present state of *soundy* static analysis techniques is dire, as unsound choices (1) may not be documented, and unknown to non-experts, (2) may not even be known to tool designers (*i.e.*, implicit assumptions), and (3) may propagate into future research. The soundness manifesto describes the misplaced confidence generated by the insufficient study and documentation of *soundy* tools, in the specific context of language features [190]. Motivated by the manifesto, we leverage soundness at the general, conceptual level of design choices, and attempt to resolve the status quo of *soundy* tools by making them more secure as well as transparent.

We describe the *Mutation-based Soundness Evaluation* (μ SE, read as “muse”) framework that enables systematic security evaluation of Android static analysis tools to discover unsound design assumptions, leading to their documentation, as well as improvements in the tools themselves. In particular, this chapter describes the extension of the original μ SE paper published in USENIX’18 in August [50]. μ SE leverages the practice of mutation analysis from the software engineering (SE) domain [224, 142, 78, 193, 80, 239, 27, 308, 225, 85], and specifically, more recent advancements in mutating Android apps [184]. In doing so, μ SE adapts a well-founded practice from SE to security, by making useful changes to contextualize it to evaluate security tools.

μ SE creates *security operators*, which reflect the security goals of the tools being analyzed (*e.g.*, data leak or SSL vulnerability detection). These security operators are seeded, *i.e.*, inserted into one or more Android apps, as guided by a *mutation scheme*. This seeding results in the creation of multiple mutants (*i.e.*, code that represents the target unwanted behavior) within the app. Finally, the mutated application is analyzed using the security tool being evaluated, and the undetected mutants are then subjected to a deeper analysis. We propose a semi-automated methodology to analyze the uncaught mutants, resolve them to flaws in the tool, and confirm the flaws experimentally.

We demonstrate the effectiveness of this approach by evaluating static analysis research tools that detect data leaks in Android apps. Based on the analysis of the discovered flaws, we provide immediate patches that address one flaw, and identify classes of design-level flaws that may be hard to address without significant research effort. Further, we perform a flaw propagation study that checks for the presence of these flaws among *seven* data leak detectors.

The general contributions of this chapter can be summarized as follows:

- We introduce the novel paradigm of *Mutation-based Soundness Evaluation* (*i.e.*, μ SE), which provides a systematic methodology for discovering flaws in static analysis tools for Android, leveraging the well-understood practice of mutation analysis. We adapt mu-

tation analysis for security evaluation and design the abstractions of *security operators* and *mutation schemes*.

- *We design and implement the μ SE framework* for evaluating Android static analysis tools. μ SE adapts to the security goals of a tool being evaluated and allows the detection of unknown or undocumented flaws.
- *We demonstrate the effectiveness of μ SE by evaluating several widely used Android security tools* that detect private data leaks in Android apps. μ SE detects undocumented flaws, and demonstrates their propagation. This analysis leads to the documentation of unsound assumptions at the design-level, and immediate security patches for an easily fixable but evasive flaw.

We published an earlier version of this work in USENIX'18 [50] where we analyzed FlowDroid [32] using μ SE and discovered 13 previously undocumented flaws. The current version of this study, described in this chapter, substantially extends upon the previous work, in the following manner:

- **Design and Implementation of μ SE:** We designed a new scope-based mutation placement approach, in addition to the mutation schemes originally explored in the USENIX'18 paper [50]. Further, we refined the implementation of the reachability-based scheme by integrating class declaration-level placement. Moreover, this extension includes a discussion on leveraging the abstractions invented in μ SE for evaluating tools with security goals other than data leak detection, such as the detection of cryptographic-API misuse vulnerabilities. Such a discussion on the general applicability of μ SE's abstractions was not present in the USENIX'18 paper. Finally, we enhance μ SE's execution engine as a part of this extension, thereby improving its accuracy in terms of associating execution traces with mutants.
- **Multiplicative improvement in mutants generated:** In the USENIX'18 paper [50], 7 Android apps were used as base applications for mutation, to produce 7,584 compilable mutants. In this extension, we added 8 new apps, resulting in a total of 15 real-world,

open-source Android apps used for mutation. We were able to seed 30,117 compilable mutants into these 8 new apps, of which 4,385 were executable, and were used for evaluating additional data leak detectors (see next). More importantly, to gauge the impact of the design and implementation improvements on μ SE’s ability to generate compilable mutants, we also mutated the 7 original apps, which resulted in 24,819 mutants, *i.e.*, 17,235 more mutants (a 2.27x *increase in compilable mutants*) over the USENIX’18 study.

- **Evaluation of the Effectiveness of Mutation Schemes:** In this extension, we perform a data-driven evaluation of the implemented mutation schemes, across two primary directions: (1) their ability to create executable mutants, and (2) their ability to create mutants that may lead to the discovery of flaws. This analysis, and the resultant insights regarding schemes is novel, as no such evaluation was performed in the USENIX’18 paper [50].
- **Significant additional extrinsic evaluation with an in-depth analysis of Argus and HornDroid:** We studied FlowDroid in-depth in the USENIX’18 paper, which formed *the core of the extrinsic evaluation*. In this study, we effectively *tripled the extrinsic evaluation* by performing an in-depth soundness evaluation of two additional state-of-the-art data leak detecting static analysis approaches for Android, namely Argus (previously known as AmanDroid) [105] and HornDroid [59]. We used the newly generated set of 4,385 executable mutants generated from the 8 new base apps to evaluate Argus and HornDroid.
- **New Findings:** The extended analysis led to significant new findings over the USENIX’18 study [50], which can be summarized into the following four points. (1) **New flaws.** We found 12 novel flaws in Argus and HornDroid, in addition to the 13 flaws discovered in FlowDroid in the USENIX’18 study [50], which brings the total number of discovered flaws to 25. (2) **New flaw class.** We discovered a *new flaw class*, aside from the four classes discussed in the USENIX’18 version, due to a distinctive and novel pattern exhibited by certain flaws, *i.e.*, a fundamentally flawed analysis of critical Android life-

cycle methods that are present in all applications (e.g., onCreate). (3) **New insight on propagation.** Furthermore, we discovered a *new insight in terms of how flaws propagate across tools*, relative to the USENIX’18 paper. That is, in the USENIX paper we found that flaws generally propagate when a direct inheritance relationship is present among two tools (*i.e.*, directly relying on the code base), but also observed that the flaws did not propagate to tools that did not have a direct relationship, but were simply built for similar design goals. However, in this work, on studying the propagation of flaws with four additional tools (*i.e.*, across total 7 tools), we discovered that every single flaw was present in at least one other tool, which means that flaws propagate across tools purely because of the shared design goal, even without any direct inheritance relationship. Such propagation based purely on the design goal was speculated in the USENIX’18 paper, but was not evident, as it is in this work. (4) **New insight on re-emergence of flaws.** Finally, we found that flaws that are fixed after reporting, *can re-emerge in future updates* of a tool. These findings demonstrate that soundness issues can be introduced at any point in tools’ lifecycles, which further necessitates continuous evaluation with μ SE.

We have released the μ SE framework, the security operators and mutation schemes constructed for evaluating data leak detectors, as well as all of the experimental data, to facilitate the reproducibility of the results, as well as to enable security researchers, tool designers, and analysts uncover undocumented flaws and unsound choices in sound security tools [203].

3.1 μ SE

We describe μ SE, a semi-automated framework for systematically evaluating Android static analysis tools that adapts the process of mutation analysis commonly used to evaluate software test suites [224]. That is, we aim to help discover concrete instances of flawed security design decisions made by static analysis tools, by exposing them to method-

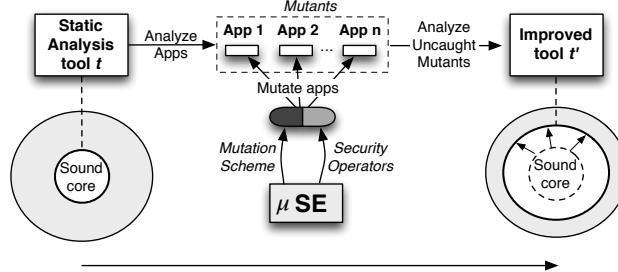


Figure 3.1: μ SE tests a static analysis tool on a set of mutated Android apps and analyzes uncaught mutants to discover and/or fix flaws.

ologically mutated applications. We envision two primary benefits from μ SE: *short-term* benefits related to straightforwardly fixable flaws that may be patched immediately, and *long-term* benefits related to the continuous documentation of assumptions and flaws, even those that may be hard to resolve. This section provides an overview of μ SE (Figure 3.1) and its design goals.

As shown in Figure 3.1, we take an Android static analysis tool to be evaluated (*e.g.*, FlowDroid [32] or MalloDroid [100]) as input. μ SE executes the tool on *mutants*, *i.e.*, apps to which *security operators* (*i.e.*, security-related mutation operators) are applied, as per a *mutation scheme*, which governs the placement of code transformations described by operators in the app (*i.e.*, thus generating mutants). The security operators represent anomalies that the static analysis tools are expected to detect, and hence, are closely tied to the security goal of the tool. The uncaught mutants indicate flaws in the tool, and analyzing them leads to the broader discovery and awareness of the unsound assumptions of the tools, eventually facilitating security-improvements.

Design Goals: Measuring the security provided by a system is a difficult problem; however, we may be able to better predict failures if the assumptions made by the system are known in advance. Similarly, although soundness may be a distant ideal for security tools, we assert that it should be feasible to articulate the boundaries of a tool’s sound core. Knowing these boundaries would be immensely useful for analysts who use security tools, for developers looking for ways to improve tools, as well as for end users who benefit from the security analyses provided by such tools. To this end, we design μ SE to provide an

effective foundation for evaluating Android security tools. Our design of μ SE is guided by the following goals:

- G1 *Contextualized security operators.*** Android security tools have diverse purposes and may claim various security guarantees. Security operators must be instantiated in a way that is sensitive to the context or purpose (*e.g.*, data leak identification) of the tool being evaluated.
- G2 *Android-focused mutation schemes.*** Android’s security challenges are notably unique, and hence require a diverse array of novel security analyses. Thus, the strategies for defining mutation schemes, *i.e.*, the *placement* of the target, unwanted behavior in the app, must consider Android’s abstractions and application model for effectiveness.
- G3 *Minimize manual-effort during analysis.*** Although μ SE is certainly more feasible than manual analysis, we intend to significantly reduce the manual effort spent on evaluating undetected mutants. Thus, our goal is to dynamically filter inconsequential mutants, and to develop a systematic methodology for resolving undetected mutants to flaws.
- G4 *Minimize overhead.*** We expect μ SE to be used by security researchers as well as tool users and developers. Hence, we must ensure that μ SE is efficient so as to promote a wide-scale deployment and community-based use of μ SE.

Threat Model: The design goals delineated above are ultimately meant to address a specific threats that arise from static analyses that are *thought* to be sound – but are not actually not sound – leading to undiscovered security vulnerabilities in Android apps. μ SE is designed to help security researchers evaluate tools that detect vulnerabilities (*e.g.*, SSL misuse), *and more importantly*, tools that detect malicious or suspicious behavior (*e.g.*, data leaks). Thus, the security operators and mutation schemes defined in this work are of an adversarial nature. That is, behavior like “data leaks” is intentionally malicious/curious, and generally not attributed to accidental vulnerabilities. Therefore, to evaluate the

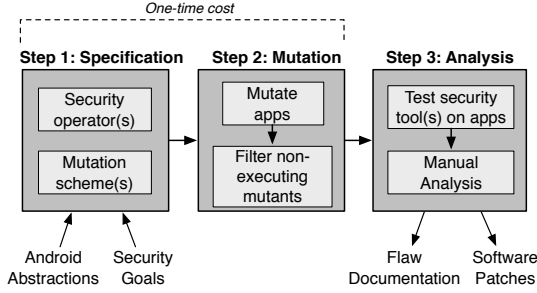


Figure 3.2: The components and process of the μ SE.

soundness of existing tools that detect such behavior, μ SE has to develop mutants that mimic such adversarial behavior as well, by defining mutation schemes of an adversarial nature. This is the key difference between μ SE and prior work on fault/vulnerability injection (e.g., LAVA [89]) that assumes the mutated program to be benign.

3.2 Design

Figure 3.2 provides a conceptual description of the process followed by μ SE, which consists of three main steps. In Step 1, we *specify* the security operators and mutation schemes that are relevant to the security goals of the tool being evaluated (e.g., data leak detection), as well as certain unique abstractions of Android that separately motivate this analysis. In Step 2, we *mutate* one or more Android apps using the security operators and defined mutation schemes using a *Mutation Engine (ME)*. After this step each app is said to contain one or more mutants. To maximize effectiveness, mutation schemes in μ SE stipulate that mutants should be systematically injected into all potential locations in code where operators can be instantiated. In order to limit the effort required for manual analysis due to potentially large numbers of mutants, we first filter out the non-executing mutants in the Android app(s) using a dynamic *Execution Engine (EE)* (Section 3.3). In Step 3, we apply the security tool under investigation to *analyze* the mutated app, leading it to detect some or all of the mutants as anomalies. We perform a methodological manual analysis of the undetected mutants, which may lead to documentation of flaws, and software patches.

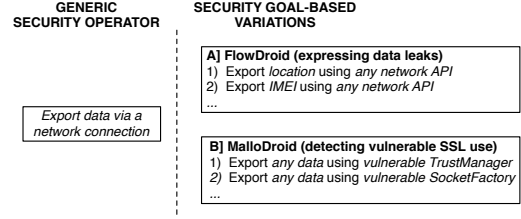


Figure 3.3: A generic “network export” security operator, and its more fine-grained instantiations in the context of FlowDroid [32] and MalloDroid [100].

Note that tools sharing a security goal (*e.g.*, FlowDroid [32], Argus [105], HornDroid [59] and BlueSeal [255] all detect data leaks) can be analyzed using the same security operators and mutation schemes, and hence the mutated apps, significantly reducing the overall cost of operating μ SE (Goal **G4**).

This section describes the design of μ SE, including the additional contributions made in this extension of our USENIX’18 study [50]. Moreover, in Section 3.2.4, we describe a general approach for leveraging the abstractions introduced in μ SE to evaluate security tools built for goals other than data leak detection (*e.g.*, cryptographic API misuse detection).

3.2.1 Security Operators

A security operator is a description of the unwanted behavior that the security tool being analyzed aims to detect. When designing security operators, we are faced with an important question: *what do we want to express?* Specifically, the operator might be too coarse or fine-grained; finding the correct granularity is the key. For instance, defining operators specific to the implementations of individual tools may not be scalable. On the contrary, defining a generic security operator for all the tools may be too simplistic to be effective. Consider the following example:

Figure 3.3 describes the limitation of using a generic security operator that describes code which “exports data to the network”. Depending on the tool being evaluated, we may need a unique, fine-grained, specification of this operator. For example, for evaluating FlowDroid [32], we may need to express the specific types of private data that can be exported via any of the network APIs, *i.e.*, the data portion of the operator is more important than what network API is used. However, for evaluating a tool that detects vulnerable SSL connections (*e.g.*, CryptoLint [93]), we may want to express the use of vulnerable SSL APIs (*i.e.*, of SSL classes that can be overridden, such as a custom TrustManager that trusts all certificates) without much concern for what data is exported. That is, the requirements are practically orthogonal for these two use cases, rendering a generic operator


```

1 Inject:
2   String dataLeak## = java.util.Calendar.getInstance().getTimeZone().getDisplayName
   ();
3   android.util.Log.d("leak-##", dataLeak##);

```

Listing 3.1: Security operator that injects a data leak from the Calendar API access to the device log.

useless, whereas precisely designing tool-specific operators may not scale.

In μ SE, we take a balanced approach to solve this problem: instead of tying a security operator to a specific tool, we define it in terms of the *security goal* of the concerned tool (Goal **G1**). Because the security goal influences the properties exhibited by a security analysis, security operators designed with a particular goal in consideration would apply to all the tools that claim to have that security goal, hence making them feasible and scalable to design. For instance, a security operator that reads information from a private source (*e.g.*, IMEI, location) and exports it to a public sink (*e.g.*, the device log, storage) would be appropriate to use for all the tools that claim to detect private data leaks (*e.g.*, Argus [105], HornDroid [59], BlueSeal [255]) (*e.g.*, see Listing 3.1 for one such implemented operator). Moreover, security operators generalize to other security goals as well; a simple operator for evaluating tools that detect vulnerable SSL use (*e.g.*, MalloDroid) could add a TrustManager with a vulnerable `isServerTrusted` method that returns true.

To derive security operators at the granularity of the security goal, we must examine the claims made by existing tools; *i.e.*, security tools must certainly detect the unwanted behavior that they claim to detect, unless affected by some unsound design choice that hinders detection. We inspect the following sources to precisely identify the security flaws considered by tools:

1) Research Papers: The tool’s research paper is often the primary source of information about what unwanted behavior a tool seeks to detect. We inspect the properties and variations of the unwanted behavior as described in the paper, as well as the examples provided, to formulate security operator specifications for injecting the unwanted behavior in an app. However, we do not create operators using the limitations and assumptions already documented in the paper or well-known in general (*e.g.*, reflection or dynamic

code loading), as μ SE seeks to find unknown assumptions.

2) Open source tool documentation: Due to space limitations or tool evolution over time, research papers may not have the most complete or up-to-date information considering what security flaws a tool can actually address. We used tool documentation available in online appendices and open source repositories to fill this knowledge gap.

3) Testing toolkits: Manually-curated testing toolkits (*e.g.*, DroidBench [32]) may be available and may provide examples of baseline operators.

3.2.2 Mutation Schemes

To enable the security evaluation of static analysis tools, μ SE must seed mutations within Android apps. For this purpose, we define *mutation schemes*, *i.e.*, the methods for choosing *where* to apply security operators to inject mutations within Android apps.

Our design of mutation schemes leverages a number of factors: (1) Android’s unique abstractions, (2), the intent to over-approximate reachability for coverage, and (3) the security goal of the tool being analyzed. We design mutation scheme *strategies* based on these factors (Section 3.2.2.1→Section 3.2.2.3), and describe them in the context of our running example first described in Section 2.1 (but elaborated as follows):

Recall that FlowDroid [32], the target of our analysis in Section 2.1, detects data leaks in Android apps. Hence, FlowDroid loosely defines a data leak as a flow from a sensitive *source* of information to some *sink* that exports it. FlowDroid lists all of the sources and sinks within a configurable “SourcesAndSinks.txt” file in its tool documentation. A simple data leak mutation may be implemented by using the data leak operator described previously, with a source (*e.g.*, `java.util.Calendar.getTimeZone()`) and sink (*e.g.*, `android.util.Log.d()`) from this file.

The strategies described in the rest of this section guide our implementation of *four* specific mutation schemes, using which we seed this data leak across target Android applications (see Section 3.3 for implementation). In particular, we significantly enhance the

goal-based mutation scheme strategy by introducing scope as a factor, and implement a scope-based mutation-scheme (Section 3.3), which further increases the expressiveness of the mutation introduced by μ SE.

3.2.2.1 Mutation Scheme Strategy 1: Leveraging Android Abstractions

The Android platform and app model support numerous abstractions that pose challenges to static analysis. One commonly stated example is the absence of a Main method as an entry-point into the app, which compels static analysis tools to scan for the various entry points, and treat them all similarly to a traditional Main method [32, 146].

Based on our domain knowledge of Android and its security, we choose the following features as a starting point in a mutation scheme strategy that models unique aspects of Android, and more importantly, tests the ability of analysis tools to detect unwanted behavior placed within these features (Goal **G2**):

1) Activity and Fragment lifecycle: Android apps are organized into a number of *activity* components, which form the user interface (UI) of the app. The activity lifecycle is controlled via a set of callbacks, which are executed whenever an app is launched, paused, closed, started, or stopped [83]. In addition, Fragments are UI elements that possess similar callbacks, although they are often used in a manner secondary to activities. We design our mutation scheme to place mutants within methods of fragments and activities where applicable, so as to validate a tool’s ability to model the activity and fragment life-cycles.

2) Callbacks: Because much of Android relies on callbacks triggered by events, these callbacks pose a significant challenge to traditional static analyses, as their code can be executed asynchronously in several different potential orders. We place mutants within these asynchronous callbacks to validate the tools’ ability to soundly model the asynchronous nature of Android. For instance, consider the example in Listing 3.2, where the `onClick()` callback can execute at any point of time.

```

1  final Button button = findViewById(R.id.button_id);
2  button.setOnClickListener(new View.OnClickListener() {
3      public void onClick(View v) {
4          // Code here executes on main thread after user presses button
5      }
6  });

```

Listing 3.2: Dynamically created onClick callback

3) Intent messages: Android apps communicate with one another and listen for system-level events using Intents, Intent Filters, and Broadcast Receivers [82, 81]. Specifically, Intent Filters and Broadcast Receivers form another major set of callbacks into the app. Moreover, Broadcast Receivers can be dynamically registered. Our mutation scheme not only places mutants in the statically registered callbacks such as those triggered by Intent Filters in the app’s Android Manifest, but also callbacks dynamically registered within the program, and even within other callbacks, *i.e.*, recursively. For instance, we generate a dynamically registered broadcast receiver inside another dynamically registered broadcast receiver, and instantiate the security operator within the inner broadcast receiver (see Listing A.1 in the Appendix for the code).

4) XML resource files: Although Android apps are primarily written in Java, they additionally include resource files that establish callbacks. Such resource files also allow the developer to register for callbacks from an action on a UI object (*e.g.*, the onClick event, for callbacks on a button being touched). As described previously, static analysis tools often list these callbacks on par with the Main function, *i.e.*, as one of the many entry points into the app. We incorporate these resource files into our mutation scheme, *i.e.*, mutate them to call our specific callback methods.

3.2.2.2 Mutation Scheme Strategy 2: Evaluating Reachability

The objective behind this simple, but important, mutation scheme is to exercise the reachability analysis of the tool being evaluated. We inject mutants (*e.g.*, data leaks from our example) at the start of every method in the app. Furthermore, as part of this extended study, we add leaks at the class declaration-level as well. While the previous schemes add methods to the app (*e.g.*, new callbacks), this scheme simply verifies if the app successfully

```

1 String dataLeak0 = java.util.Calendar.getInstance().getTimeZone().getDisplayName();
2 String[] leakArray0 = new String[] { "n/a", dataLeak0};
3 String dataLeakPath0 = leakArray0[leakArray0.length - 1];
4 android.util.Log.d("leak-0", dataLeakPath0);

```

Listing 3.3: Complex Path Operator Placement

```

1 public class ParentClass {
2     String dataLeak = "";
3     int methodA(){
4         android.util.Log.d("leak-0-1", dataLeak);
5         return 1; }
6     class ChildClass{
7         int childMethodA(){
8             dataLeak = java.util.Calendar.getInstance().getTimeZone().getDisplayName
9             ();
10            android.util.Log.d("leak-0-0", dataLeak);
11            return 1; }}}

```

Listing 3.4: Scope based operator placement at different levels of inheritance.
models the bare minimum.

3.2.2.3 Mutation Scheme Strategy 3: Leveraging the Security Goal

Like security operators, mutation schemes may also be designed in a way that accounts for the security goal of the tool being evaluated (Goal **G1**). Such schemes may be applied to any tool with a similar objective. In keeping with our motivating example (Section 2.1) and our evaluation (Section 3.4), we develop an example mutation scheme strategy that can be specifically applied to evaluate data leak detectors. This strategy can be instantiated in terms of the following three methods of seeding mutants:

1) Taint-based operator placement: This placement methodology tests the tools' ability to recognize an asynchronous ordering of callbacks, by placing *the source in one callback and the sink in another*. The execution of the source and sink may be triggered due to the user, and the app developer (*i.e.*, especially a malicious adversary) may craft the mutation scheme specifically so that the sources and sinks lie on callbacks that generally execute in sequence. However, this sequence may not be observable through just static analysis. A simple example is collecting the source data in the `onStart()` callback, and leaking it in the `onResume()` callback. As per the activity lifecycle, the `onResume()` callback *always* executes right after the `onStart()` callback.

2) Complex-Path operator placement: Our preliminary analysis demonstrated that static analysis tools may sometimes stop after an arbitrary number of hops when analyzing a call graph, for performance reasons. This finding motivated the complex-path operator placement. In this scheme, we make the path between source and sink as complex as possible (*i.e.*, which is ordinarily one line of code, as seen in Listing 3.1). That is, the design of this scheme allows the injection of code along the path from source to sink based on a set of predefined rules. For our evaluation, we instantiate this scheme with a rule that recreates the String variable saved by the source, by passing each character of the string into a StringBuilder, then sending the resulting string to the sink as shown in Listing 3.3. μ SE allows the analyst to dynamically implement such rules, as long as the input and output are both strings, and the rule complicates the path between them by sending the input through an arbitrary set of transformations.

3) Scope-based operator placement: This scope-based placement methodology was not defined in our USENIX’18 paper [50], and is a new addition to μ SE as part of this extension. As the name suggests, μ SE can inject code by analyzing scopes based on *visibility*. For example, as shown in Listing 3.4, childMethodA is visible from both ChildClass and ParentClass. Therefore, we declare a variable dataLeak at the ParentClass, while assigning the leak source at the childMethodA. Consequently, we insert corresponding sinks in childMethodA and methodA. Note that this scheme is not restricted by callbacks and can be useful for different variations of method declarations. As application developers may arbitrarily organize class scopes, seeding mutants into applications using this scheme generally results in interesting and often complicated mutant placement, which further assists in stress-testing security techniques that assume an adversarial threat model (e.g., data leak detectors).

In a traditional mutation analysis setting, the mutation placement strategy would seek to minimize the number of non-compilable mutants. However, as our goal is to evaluate the soundness of Android security tools, we design our mutation scheme to over-approximate.

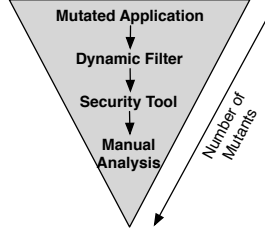


Figure 3.4: The number of mutants (*e.g.*, data leaks) to analyze drastically reduces at every stage in the process.

Once the mutated apps are created, for a feasible analysis, we pass them through a dynamic filter that removes the mutants that cannot be executed, ensuring that the mutants that each security tool is evaluated against are all executable *i.e.*, the data leaks can indeed happen in runtime.

Note that although mutation schemes using the first two strategies (Section 3.2.2.1 and Section 3.2.2.2) may be generally applied to any type of static analysis tool (*e.g.*, SSL vulnerability and malware detectors), the third strategy, as the description suggests, will need to be adapted per security goal (*e.g.*, data leak detection). We elaborate on this point by presenting a general approach for leveraging our mutation abstractions for other security goals, in Section 3.2.4.

3.2.3 Analysis Feasibility & Methodology

μ SE reduces manual effort by filtering out mutants whose security flaws are not verified by dynamic analysis (Goal **G3**). As described in Figure 3.2, for any given mutated app, we use a dynamic filter (*i.e.*, the Execution Engine (EE), described in Section 3.3) to purge non-executable leaks. If a mutant (*e.g.*, a data leak) exists in the mutated app, but is not confirmed as executable by the filter, (*i.e.*, data does not leak from source to sink), we discard it. For example, data leaks injected in dead code are filtered out. Thus, when the Android security tools are applied to the mutated apps, only mutants that were executed by EE are considered.

Furthermore, after the security tools were applied to mutant apps, only *undetected*

mutants are considered during analyst analysis. The reduction in the number of mutants subject to analysis at each step of the μ SE process is illustrated in Figure 3.4.

The following methodology is used by an analyst for each undetected mutant after testing a given security tool to isolate and confirm flaws:

1) Identifying the Source and Sink: During mutant generation, μ SE’s ME injects a unique mutant identifier, as well as the source and sink using `util.Log.d` statements. Thus, for each undetected mutant, an analyst simply looks up the unique IDs in the source to derive the source and sink.

2) Performing Leak Call-Chain Analysis: Since the data leaks under analysis went undetected by a given static analysis tool, this implies that there exists one (or multiple) method call sequences (*i.e.*, call-chains) invoking the source and sink that could not be modeled by the tool. Thus, a security analyst inspects the code of a mutated app and identifies the observable call sequences from various entry points. This is aided by dynamic information from the EE so that an analyst can examine the order of execution of detected data leaks to infer the propagation of leaks through different call chains.

3) Synthesizing Minimal Examples: For each of the identified call sequences invoking a given undetected data leak’s source and sink, an analyst then attempts to synthesize a minimal example by re-creating the call sequence using only the required Android APIs or method calls from the mutated app. This info is then inserted into a pre-defined skeleton app project so that it can be again analyzed by the security tools to confirm a flaw.

4) Validating the Minimal Example: Once the minimal example has been synthesized by the analyst, it must be validated against the security tool that failed to detect it earlier. If the tool fails to detect the minimal example, then the process ends with the confirmation of a flaw in the tool. If the tool is able to detect the examples, the analyst can either iteratively refine the examples, or discard the mutant, and move on to the next example.

3.2.4 Leveraging Security Operators and Mutation Schemes for other Security Goals

Our design describes the μ SE framework in terms of the security goal of data leak detection, primarily due to the popularity of the goal, as well as the proliferation of data leak detectors in academic research. However, μ SE may be applied as a general framework, to evaluate detection techniques that target other security goals. We briefly describe the generality of μ SE, and specifically, its abstractions of security operators and mutation schemes for evaluating tools that detect *cryptographic-API misuse*, which is the *second most* prolific cause of vulnerabilities, after data leaks [290]. In this context, μ SE will evaluate tools such as CryptoGuard [243] and CrySL [175], which try to detect misuse of crypto APIs.

μ SE’s abstractions may be naturally leveraged to evaluate crypto-API misuse detectors. To elaborate, the *security operators* for evaluating such tools would represent well-known cryptographic vulnerabilities, such as passing a weak algorithm name as parameter to an encryption related cryptography API. For example, one operator could be represented by passing insecure parameters in an API such as `Cipher.getInstance()`, such as *only passing* AES as a parameter, which would initialize an insecure default cipher that use the ECB mode.

Similarly, μ SE’s mutation schemes may also be directly leveraged to place such mutations in a hard-to-detect manner. First, the scheme that evaluates reachability (Section 3.2.2.2) can be directly applied, by placing such vulnerable API invocations (i.e., mutations) at as many reachable locations within the target app as possible. Similarly, the Android-specific scheme-strategies (Section 3.2.2.1) can be leveraged to place the vulnerable invocation in commonly used Android abstractions such as fragments and broadcast receivers. Finally, the security goal-based scheme-strategies (Section 3.2.2.3) can be leveraged to evaluate crypto-API misuse detectors as well. For example, we can consider the parameter passed to the `Cipher.getInstance()` API, *i.e.*, the AES string, as a

source value, and the API call as the sink, and use the taint-based mutation scheme to distribute these source and sink values across different Android lifecycle methods, such that they would execute in the right sequence, but be hard to detect. Similarly, our newly defined scope-based mutation scheme can be used to inject the source parameter and the API in different program scopes, in a way that would execute flawlessly at runtime, but would be hard to detect. Other than syntactical changes required to implement the security operator (*i.e.*, defining new *source-sink pairs*, but with Cipher APIs) and adjusting it for mutation using the schemes (*i.e.*, scoping them within required *try-catch* block), no other changes will be required to the framework, for making it applicable for evaluating crypto-API misuse detectors. We elaborate on the estimated effort required to make these changes in Section 3.9.

3.3 Implementation

This section provides the implementation details of μ SE’s components: (1) the ME for mutating apps, and (2) the EE for exercising and filtering out non-executable mutants. We have made μ SE available to the research community [203], along with all the data and code generated or used.

1. Mutation Engine (ME): The ME allows μ SE to automatically mutate apps according to a fixed set of security operators and mutation schemes. The ME is implemented in Java and extends the MDroid+ mutation framework for Android [184]. More specifically, μ SE’s ME implements the seeding of mutants according to our defined *mutation schemes*. This required extensions to MDroid+’s implemented static analyses to identify a more diverse array of source code locations (*e.g.*, analyzing the visibility scope for the *scope-based operator placement* scheme). MDroid+’s previous implementation only supported identifying strings and specific API patterns. These additions help to make μ SE’s ME generic, as it could be applied to support additional security operators or mutation schemes in the future. To achieve these extensions, we designed ME to do the following:

Firstly, the ME derives a mutant injection profile (MIP) of all possible injection points for a given mutation scheme, security operator, and target app source code. The MIP is derived through one of two types of analysis: (i) text-based parsing and matching of `xml` files in the case of app resources; or (ii) using Abstract Syntax Tree (AST) based analysis for identifying potential injection points in code. μ SE takes a systematic approach toward applying mutants to a target app, and for each mutant location stipulated by the MIP for a given app, a mutant is seeded. The injection process additionally uses either text or AST-based code transformation rules to modify the code or resource files. In the context of our evaluation, μ SE further marks injected mutants in the source code with log-based indicators that include a unique identifier for each mutant, as well as the source and sink for the injected leak. This information can be customized for future security operators and exported as a ledger that tracks mutant data. μ SE can be extended to additional security operators and mutation schemes by adding methods to derive the MIP and perform target code transformations.

Given the time cost in running the studied security-focused static analysis tools on a set of `apks`, μ SE breaks from the process used by traditional mutation analysis frameworks that seed each mutant into a separate program version, and seeds all mutants into a single version of a target app. Finally, the target app is automatically compiled using its build system (*e.g.*, Gradle [158], ant [108]) so that it can be dynamically analyzed by the EE.

2. Mutation Schemes: We implemented four mutation schemes using the strategies outlined in Section 3.2.2. First, we implemented the Reachability scheme, using the reachability-based strategy (Section 3.2.2.2). Then, we implemented three other schemes, Scope, Taint, and Complex Reachability, using the security goal-based strategy (Section 3.2.2.3). Finally, we integrated the Android-specific mutation scheme strategy (Section 3.2.2.1) in these four schemes by prioritizing placement into several abstractions unique to Android (*e.g.*, fragments, receivers, asynchronous task classes). Among these, the Scope based scheme was not implemented in our prior USENIX’18 paper [50].

3. Execution Engine (EE): To facilitate a feasible manual analysis of the mutants that are undetected by a security analysis tool, μ SE uses the EE to dynamically analyze target apps, verifying whether or not injected mutants can be executed in practice. To further elaborate with respect to our implementation, an *executable mutant* represents a data leak from source to sink as observed through our execution engine. This EE builds upon prior work in automated input generation for Android apps by adapting the systematic exploration strategies from the CRASHSCOPE tool [201, 199] to explore a target app’s GUI. We made several improvements to CRASHSCOPE in order to tailor the automated app execution to the goal of discovering as many *executable* mutants as possible. For instance, CRASHSCOPE’s execution strategies were originally intended to uncover crashes in Android apps and thus included several mechanisms that executed common crash inducing actions on apps including rotating the screen, and injecting text with special characters. In order to be used in μ SE, we discarded these crash inducing operations, and developed strategies that focused upon uncovering as many execution states of the app as possible. Additionally, as part of extending our USENIX’18 paper [50], we made a practical improvement to the manner in which CRASHSCOPE analyzes applications as part of our current study. That is, during initial testing we discovered that even after uninstalling an application from an Android virtual device used in CRASHSCOPE, certain background services may persist, which can contaminate the runtime mutant execution logs across applications. We addressed this problem by modifying CRASHSCOPE to instantiate a new, clean Android virtual device for each application analyzed, thereby fully isolating application execution logs.

We discuss the limitations of the EE in Section 3.9 (see the Appendix for additional details).

3.4 Evaluation Overview

The primary objective of this evaluation is to measure the effectiveness of μ SE at uncovering flaws in security-focused static analysis tools for Android apps, and to demonstrate the extent of such flaws. As a case study, we focus our evaluation on the security goal of data leak detection, which has received significant attention from the security research community in the last decade [96, 94, 115, 181, 94, 180, 168, 244, 179]. Our evaluation is guided by 6 key research questions that provide an objective measure of the effectiveness and practicality of μ SE, while also shedding light on the general nature of unsound decisions/flaws in static analysis security tools:

RQ₁ *Effectiveness of μ SE.* *Can μ SE find security problems in static analysis tools for Android, and help resolve them to flaws/unsound choices?*

RQ₂ *Relative effectiveness of mutation schemes.*¹ *How effective are the individual mutation schemes for (a) generating executable mutants and (b) discovering security flaws?*

RQ₃ *Propagation of flaws.* *Are flaws inherited when a tool is reused, or built based on similar principles?*

RQ₄ *Addressing flaws.* *Are all flaws unearthed by μ SE hard to address, or can some be patched?*

RQ₅ *Scalability of μ SE.* *Does the semi-automated methodology of μ SE for analyzing mutants allow for a feasible analysis (i.e., in terms of the manual effort)?*

RQ₆ *Performance of μ SE.* *What is the runtime performance of μ SE?*

To address **RQ₁→RQ₆**, we performed several experiments for a period of over two years (i.e., October 2017 → January 2020). We started by creating a data-leak security operator (i.e., as described in Section 3.2.1), and used μ SE’s expressive mutation schemes (Section 3.2.2) to seed the corresponding mutated code in a set of 15 open source Android applications obtained from F-droid [102] (see Table A.1 in the Appendix for the list), cre-

¹**RQ₂** was not investigated in our USENIX’18 paper [50].

ating 54,936 mutants representing injected data leaks. Section 3.5 describes this process, along with the refinement made possibly by μ SE’s EE, and addresses questions pertaining to μ SE’s intrinsic evaluation (**RQ**₂, **RQ**₅ and **RQ**₆). We selected three prominent data leak detectors, namely, FlowDroid [32], Argus [105] (previously known as AmanDroid), and HornDroid [59], as the target of our in-depth evaluation using the end-to-end approach as described in Section 3.2. The in-depth study of FlowDroid was reported in our USENIX’18 paper [50] and is not re-performed in this study. Due to the longitudinal nature of this study, we strived to use the latest release of the data leak detectors whenever available. Section 3.6 describes this evaluation, the 13 flaws that we previously discovered (**RQ**₁) in FlowDroid and the 12 new flaws found in HornDroid and Argus. Further, we also briefly describe the one flaw that we could fix (**RQ**₄), and the relative effectiveness of μ SE’s mutation schemes in unearthing flaws (**RQ**₂). We developed minimal examples of the discovered flaws, and performed a *flaw propagation study* (Section 3.7) to discover the extent to which flaws discovered in one tool manifest in others developed for the same security goal (**RQ**₃). Particularly, we used four additional data leak detection tools, *i.e.*, in addition to the three tools analyzed in depth in Section 3.6, bringing the total to seven tools, for which propagation was studied. Note that compared to our USENIX’18 paper [50], which only studied the propagation of flaws discovered in FlowDroid, we describe the propagation of flaws discovered in three tools, namely FlowDroid, Argus, and HornDroid. Our results demonstrate that flaws generally propagate to other tools, and more so if the tools rely on common design principles.

Table 3.1: The number of leaks inserted by μ SE, and the final number marked as executable by μ SE’s EE. Note that the “-” indicates that the scheme is not applicable to a particular app, due to the app’s particular characteristics (*e.g.*, the absence of fragments)

App ID	Inserted Leaks per Scheme				Executable Leaks per Scheme			
	Reachability	Scope	Taint	Complex Reachability	Reachability	Scope	Taint	Complex Reachability
App 01	24	48	66	22	13 ($\approx 54\%$)	12 ($\approx 12\%$)	8 ($\approx 12\%$)	11 ($\approx 50\%$)
App 02	106	-	231	83	63 ($\approx 59\%$)	-	80 ($\approx 34\%$)	44 ($\approx 14\%$)
App 03	248	-	668	191	51 ($\approx 20\%$)	-	162 ($\approx 24\%$)	36 ($\approx 26\%$)
App 04	45	192	346	41	20 ($\approx 44\%$)	156 ($\approx 81\%$)	51 ($\approx 14\%$)	23 ($\approx 2\%$)
App 05	3181	-	12104	2777	598 ($\approx 18\%$)	-	1362 ($\approx 11\%$)	508 ($\approx 42\%$)
App 06	434	-	2699	384	57 ($\approx 13\%$)	-	110 ($\approx 4\%$)	51 ($\approx 5\%$)
App 07	204	-	551	174	111 ($\approx 54\%$)	-	243 ($\approx 44\%$)	95 ($\approx 34\%$)
App 08	975	1193	5266	839	215 ($\approx 22\%$)	56 ($\approx 4\%$)	567 ($\approx 10\%$)	168 ($\approx 32\%$)
App 09	23	8	20	21	15 ($\approx 65\%$)	6 ($\approx 75\%$)	13 ($\approx 65\%$)	12 ($\approx 7\%$)
App 10	476	-	5283	449	59 ($\approx 12\%$)	-	681 ($\approx 12\%$)	54 ($\approx 11\%$)
App 11	316	1111	827	277	50 ($\approx 15\%$)	287 ($\approx 25\%$)	83 ($\approx 10\%$)	41 ($\approx 5\%$)
App 12	250	354	428	213	77 ($\approx 30\%$)	111 ($\approx 31\%$)	59 ($\approx 13\%$)	38 ($\approx 7\%$)
App 13	156	203	828	147	89 ($\approx 57\%$)	112 ($\approx 55\%$)	295 ($\approx 35\%$)	84 ($\approx 71\%$)
App 14	125	844	663	107	79 ($\approx 63\%$)	55 ($\approx 6\%$)	27 ($\approx 4\%$)	59 ($\approx 9\%$)
App 15	1304	3478	2790	1143	211 ($\approx 16\%$)	456 ($\approx 13\%$)	172 ($\approx 6\%$)	154 ($\approx 41\%$)
Total	7867	7431	32770	6868	1708 ($\approx 22\%$)	1251 ($\approx 16\%$)	3913 ($\approx 11\%$)	1378 ($\approx 20\%$)

3.5 Executing μ SE to Create Mutants Representing Data Leaks

We applied μ SE to 15 target Android apps obtained from F-Droid [102], and created 54,936 mutants (*i.e.*, data leaks).² These leaks were generated by the μ SE’s Mutation Engine (ME) using the data-leak security operator, and the four mutation schemes described in Section 3.3, namely, (1) *reachability*, (2) *scope*, (3) *taint*, and (4) *complex reachability*.

Filtering non-executable leaks: We then used our Execution Engine (EE) to filter out non-executable leaks, as described in Section 3.3, and confirmed 8,250 out of 54,936 leaks as executable. The remaining 46,686 non-executable leaks were then removed. Note that this number is independent of the tools involved, *i.e.*, the filtering only happens once, and the mutated APKs can then be passed to any number of tools for analysis. By filtering out a large number of potentially non-executable leaks

²We use “mutants” and “leaks” interchangeably to refer to data-leak-related mutants used in the evaluation, *i.e.*, Sections 3.4–3.7.

3.5. EXECUTING μ SE TO CREATE MUTANTS REPRESENTING DATA LEAKS 47

(*i.e.*, 46,686/54,936 or about 85%), our dynamic filtering is tremendously effective at reducing the number of mutants used to evaluate security tools, and in turn, the manual effort required to analyze the uncaught mutants, which demonstrates the feasibility of μ SE (**RQ₅**).

Runtime performance: μ SE took 19 hours in total to create the mutants and filter out those that were non-executable, which is a one-time cost for each security goal, *i.e.*, which does not have to be repeated for any of the tools we analyze in particular (**RQ₆**). To elaborate, it took us 74 minutes on average to mutate each of the 15 apps, with minimum and maximum times of 16 and 170 minutes, and a standard deviation of about 53 minutes. The increase in runtime compared to our original work (92 minutes in worst case) [50] is due to two reasons: (1) heterogeneity of the applications selected for mutation, and (2) a bulk of the time spent can be attributed to our *improvements* to CrashScope (see Section 3.3) that require the Android virtual device it uses to be recreated more frequently, leading to an increase in runtime. However, the improvements also increase the reliability of CrashScope’s mutant detection by preventing cross-contamination of mutation logs across apps, and hence, are desirable.

Correlation between executable mutants and μ SE’s schemes: To improve our understanding of what factors contribute to more executable leaks, we further examined the number of executable leaks generated as a factor of the mutation schemes used to generate them (*i.e.*, since there was a single security operator used) in this extended study.

Table 3.1 shows the number of executable and non-executable leaks seeded using each of the mutation schemes described in Section 3.2.2 (**RQ₂**). As seen in the table, the Reachability and Complex Reachability result in the insertion of a somewhat similar number of leaks (*i.e.*, 7,867 and 6,868 respectively), and the fraction of leaks deemed executable by our EE is similar as well, *i.e.*, 1,708 (about 22%) and 1378 (*i.e.*, 20%) for the Reachability and Complex Reachability scheme, respectively. Our intuition is that this equivalence

3.6. IN-DEPTH EVALUATION OF DATA LEAK DETECTION TOOLS WITH μ SE 48

results from the inherent similarity in the nature of the two schemes. Moreover, these two schemes produced the highest fraction of executable mutants (*i.e.*, over 20%). Further, the Scope-based scheme inserted a total of 7,431 leaks, out of which, 1,251 (*i.e.*, or 16%) were confirmed as executable by the EE. Note that the number of leaks inserted using the Scope-based scheme for individual apps is highly variable, primarily due to wide variations in the developers' usage of scope. Finally, the Taint-based scheme was the most numerous, both in terms of the number of leaks inserted (*i.e.*, 32,770) as well as the number of executable leaks (*i.e.*, 3,913) it caused. However, the Taint-based scheme did not lead to a high rate of executable leaks, which we suspect to be due to the sheer number of leaks seeded with it. The high number of leaks (and more importantly, non-executable leaks) seeded by the Taint-based scheme is primarily due to its design, *i.e.*, it places one source per method in a class, thus creating a total of n sources distributed in n methods. Then, it places n sinks per source in each method as scope allows. As a result, around n^2 number of sinks are created in total for n sources in each class.

The superior performance of the Reachability and Complex Reachability schemes over the other two is expected, *i.e.*, as both Scope-based and Taint-based schemes insert leaks with sources and sinks distributed across methods, thereby creating unreachable sinks at a higher rate. In contrast, the Reachability and Complex Reachability schemes place leaks with the sources and sinks placed together, resulting in a lower number of unreachable sinks, and hence, a higher rate of creating executable mutants.

3.6 In-depth Evaluation of Data Leak Detection Tools with μ SE

To demonstrate the utility of μ SE, we evaluated three prominent data leak detectors with it: FlowDroid [32], HornDroid [59], and Argus [105]. Among these, FlowDroid was analyzed in-depth as part of the previous study [50], whereas we analyze Argus and HornDroid as part of our current, extended study. We selected FlowDroid and Argus for

the in-depth analysis as they are regularly maintained, with multiple publicly available versions, and form a representative sample of the current state-of-the-art. Additionally, we selected HornDroid as it is the first Android static analysis tool with a formal proof of soundness, making it an interesting case for a soundness evaluation. Compatibility with our mutant apps and general analysis feasibility were also major factors that influenced tool selection for the in-depth analysis. Specifically, we avoided tools that had not been updated recently (*i.e.*, were built for outdated Android versions), and hence were incompatible with many of our mutant apps.

Methodology: Our methodology for evaluating a security tool with μ SE is as follows: First, we analyze executable mutants with the tool being evaluated. Then, we systematically examine the surviving (*i.e.*, undetected) mutants using the methodology described in Section 3.2.3, and resolve the undetected mutants to design/implementation flaws. Using this approach, our analysis of FlowDroid, HornDroid, and Argus led to the discovery of 25 unique flaws, among which 13 were reported in our USENIX’18 paper [50]. *We confirmed that these flaws were undocumented, i.e., mentioned neither in the tools’ corresponding papers or documentation.*

3.6.1 Evaluating FlowDroid

FlowDroid was introduced by Artz et al. [32] in 2014 as a data leak detection tool for Android. It models the Android life-cycle to handle callbacks invoked by the Android Framework to perform information flow analysis and data leak detection. Furthermore, it applies context, flow, field, and object-sensitivity to reduce the number of false positive sensitive data leaks the tool detects. FlowDroid has been cited over 1,400 times, and the tool has been continuously maintained since 2014, which motivated its analysis.

For our in-depth analysis, we evaluated FlowDroid v2.0, which was the version available during our USENIX’18 study [50], using the 7,584 mutants originally created from our first seven base apps (*i.e.*, apps 01-07), leveraging the Reachability, Complex Reachability

3.6. IN-DEPTH EVALUATION OF DATA LEAK DETECTION TOOLS WITH μ SE 50

and Taint schemes. In this extension, we used the substantially improved version of μ SE to create 30,117 mutants (4,385 executable) from the remaining eight apps to evaluate HornDroid and Argus, using all four mutation schemes (*i.e.*, including the additional Scope scheme developed in this extension). This separation is mainly due to the order in which the tools were evaluated, and the longitudinal nature of our study. Moreover, this two-phase evaluation with disjoint sets of mutants led to the discovery of flaws in all three tools, which indicates that μ SE may be effective at revealing flaws in security tools, irrespective of the apps used as input.

Results: Out of the 2,026 mutants that we analyzed using FlowDroid, 987 were undetected. On analyzing the undetected mutants, we discovered 13 unique flaws in FlowDroid, demonstrating that μ SE can be effectively used to find problems that can be resolved to flaws (**RQ₁**). Using the approach from Section 3.2.3, we needed less than one hour to isolate a flaw from the set of undetected mutants, in the worst case. In the best case, flaws were found in a matter of minutes, demonstrating that the amount of manual effort required to quickly find flaws using μ SE is minimal (**RQ₅**). We provide descriptions of the flaws discovered in FlowDroid in Table 3.2.

We have reported these flaws and are working with the FlowDroid developers to resolve them. In fact, we developed two patches [281] to correctly implement Fragment support (*i.e.*, F5 in Table 3.2), which were accepted by developers. To gain insight about the practical challenges faced by static analysis tools, and their design flaws, we further categorize the flaws into the following classes:

FC1: Missing Callbacks: The security tool (*e.g.*, FlowDroid) did not recognize some callback method(s) and will not find leaks placed within them. Tools that use lists of APIs or callbacks are susceptible to this problem, as prior work has demonstrated that the generated list of callbacks (1) may not be complete, and (2) or may not be updated as the Android platform evolves. We observed both such cases in our analysis of FlowDroid. That is, DialogFragments was added in API 11 *before FlowDroid was released*, and

NavigationView was added after. These limitations are well-known in the community of researchers at the intersection of program analysis and Android security, and have been documented by prior work [60]. However, μ SE helps evaluate the robustness of existing security tools against these flaws and helps in uncovering these undocumented flaws for the wider security audience. Additionally, *some of these flaws may not be resolved even after adding the callback to the list; e.g., PhoneStateListener and SQLiteOpenHelper*, both added in API 1, are not interfaces, but abstract classes. Therefore, adding them to FlowDroid’s list of callbacks (*i.e.*, AndroidCallbacks.txt) does not resolve the issue.

FC2: Missing Implicit Call: The security tool did not identify leaks within some method that is implicitly called by another method. For instance, FlowDroid does not recognize the path to Runnable.run() when a Runnable is passed into the ExecutorService.submit(Runnable). The response from the developers indicated that this class of flaws was due to an unresolved design challenge in Soot’s [287] SPARK algorithm, upon which FlowDroid depends. This limitation is also generally well known within the program analysis community [60]. However, the documentation of this gap in analysis, thanks to μ SE, would certainly benefit researchers in the wider security community.

FC3: Incorrect Modeling of Anonymous Classes: The security tool did not detect data leaks expressed within an anonymous class. For example, FlowDroid did not recognize leaks in the onReceive() callback of a dynamically registered BroadcastReceiver, which is implemented within another dynamically registered BroadcastReceiver’s onReceive() callback. It is important to note that finding such complex flaws is only possible due to μ SE’s semi-automated mechanism and may be rather prohibitive for an entirely manual analysis.

FC4: Incorrect Modeling of Asynchronous Methods: The security tool did not recognize a data leak whose source and sink are called within different methods that are asynchronously executed. For instance, FlowDroid did not recognize the flow between data leaks in two callbacks (*i.e.*, onLocationChanged and onStatusChanged) of the Loca-

tionListener class, which the adversary may cause to execute sequentially (*i.e.*, as our EE confirmed).

Apart from **FC1**, which may be patched with limited effort, the other three categories of flaws may require a significant amount of research effort to resolve. However, documenting them is critical to increase awareness of real challenges faced by Android static analysis tools.

3.6.2 Evaluating Argus

As part of this extended study, we evaluated Argus v3.1.2 (*i.e.*, the latest version at the time of our analysis) with 4,385 executable mutants generated from the 8 new base apps, *i.e.*, apps 08-15. Further, we analyzed the 7,708 uncaught mutants (*i.e.*, the leaks not detected by Argus) using the methodology previously described in Section 3.2.3. Through the addition of 8 new base apps, we aim to add diversity to the created mutants, as these new apps are likely to encompass a new set of development practices that might impact our findings. Therefore, this new set of apps might also help us find new flaws through μ SE due to the additional, diverse mutations created. While we do not consider the original apps from our USENIX’18 study for Argus and HornDroid, we consider this to be a balanced trade-off as we later study the propagation of flaws from both old and new sets of apps across tools.

Results: Through analyzing the uncaught mutants, we discovered 9 unique flaws in Argus, as shown and classified in Table 3.3. Note that these flaws are *separate from* the ones found in FlowDroid (Section 3.6.1). This demonstrates that the flaws found from our USENIX’18 study [50] were not inherently coupled to the base apps chosen, and that μ SE can be effective with different sets of apps representing a reasonable level of diversity. A description of each flaw is provided in Table 3.3. Our project repository provides minimal APKs representing the flaws in Argus [203].

Of the 9 flaws, 3 are based on Fragment usage and are of the FC4 flaw class, while the

remaining 6 fall in FC1. Note that while we found fragment-based flaws in FlowDroid, the 4 fragment related flaws discovered for Argus are independent, although they could be interpreted as variants of those found in FlowDroid. The flaws discovered in Argus are described as follows:

Argus flaws in FC4: Similar to FlowDroid, Argus struggles with identifying leaks in fragments. We observe that most of these problems are at the design-level, and occur because Argus does not track flows between the GUI components defined in fragments. To elaborate, Argus misses asynchronous flows between GUI components defined in the relevant XML files for fragments and the corresponding click-event listeners. All of our fragment-based flaws (F14-16) exploit this design-gap in Argus to avoid detection.

Argus flaws in FC1: Of the remaining 6 flaws we discovered in Argus, 5 are based on RecyclerView widgets. The RecyclerView is used to display a collection of data within a limited, scrolling, window. Each RecyclerView widget implementation includes classes extending RecyclerView.ViewHolder and RecyclerView.Adapter. Furthermore, these classes contain several abstract methods which must be implemented, namely onCreateViewHolder, onBindViewHolder and getItemCount. Our analysis reveals that Argus fails to detect leaks placed in any of these components and methods. In addition, Argus does not detect flaws when leaks are placed in click event listeners statically connected to fragment components via XML resource files. This demonstrates that further work is required to analyze relations in between methods not only through source code, but resource files as well. Summary of these flaws are tabulated in Table 3.3, and the example minimal APKs, each exhibiting a single flaw are available in our online appendix [203].

3.6.3 Evaluating HornDroid

HornDroid was proposed by Calzavara et al. [59] as the first static analysis tool for Android with a formal proof of soundness. This unique attribute motivated our choice of HornDroid for in-depth evaluation in this extended study. HornDroid abstracts Android applications

3.6. IN-DEPTH EVALUATION OF DATA LEAK DETECTION TOOLS WITH μ SE 54

as a set of Horn clauses to formulate security properties, which can then be processed by Satisfiability Modulo Theories (SMT) solvers.

HornDroid’s formal proof and over-approximation require far more resources than the other tools we evaluate, *i.e.*, as stated in the HornDroid paper, the authors tested HornDroid on a server with 64 multi-thread cores and 758 Gb of memory, although they reported that the most memory utilization was around 10 GB. To match this maximum 10 GB memory utilization, we conducted our study on a server with 32GB of RAM and 8 cores. When analyzing a mutated app with HornDroid, we set a time-out of 36 hours, after which we would abort the analysis and report whatever mutants were caught until that time.

Results: We were not able to analyze many of our mutated apps using HornDroid. Specifically, out of the 31 mutated APKs (*i.e.*, created after mutating the base APKs 08-15), HornDroid could only analyze 4 without crashing or timing out. This was in spite of us taking care to compile the APKs with the API level that HornDroid was built to analyze, *i.e.*, API 19. This outcome is unfortunately not surprising; indeed, prior work on analyzing the feasibility of Android static analysis tools has reported similar results for research prototypes [246]. As a result, HornDroid successfully analyzed 46 executable data leaks, out of which, it caught 14, leaving 32 undetected mutants for further analysis. We discovered 3 flaws from analyzing these undetected mutants, as shown in Table 3.4.

All of the 3 flaws (F23-F25) we discovered are related to the lack of appropriate support for fragments; however, unlike prior fragment-related flaws, these flaws are generally centered around the Android lifecycle methods for fragments and activities. These flaws could broadly fit in FC1; however, they demonstrate a more fundamental, design-level lack of support for fragments, as HornDroid fails to detect leaks in even basic lifecycle methods (*e.g.*, `Fragment onCreateView`). Hence, we create a new flaw class to represent such flaws, *FC5: Android Lifecycle Callbacks*, as a special variant of FC1, described as follows:

FC5: Android Lifecycle Callbacks: This class incorporates flaws in detecting leaks in fundamental lifecycle methods, such as `onCreate`. To elaborate, there are only six lifecycle call back methods in total for any Android activity [83], where `onCreate` is the only one considered mandatory. It is also the first method to be called when the Activity is initialized *i.e.*, it is the starting point of the Activity. Hence, analyzing such callbacks is imperative for a practical analysis of data flows within the app. However, HornDroid fails to detect leaks in three specific instances of lifecycle callbacks for activities and fragments (F23-F25), which is concerning, as the tool strongly claims soundness, *i.e.*, quoted as follows: “*In order to support a sound analysis of fragments, HornDroid over-approximates their life-cycle by executing all the fragments along with the containing activity in a flow-insensitive way*”.

3.6.4 Effectiveness of Individual Schemes in Finding Flaws

As described in Section 3.2.2, μ SE uses four different mutation schemes to place operators in an app. These schemes may sometimes create overlapping mutants wherein operators are placed in the same position according to two or more schemes. As part of this extended study, we trace the flaws we found back to the operator placement strategies described in Section 3.2.2.2 and 3.2.2.3 to determine the usefulness of these approaches in finding flaws (RQ₂).

As shown in Table 3.5, of the 25 flaws, 24 could be discovered using the scope-based scheme, and 22 using the taint-based scheme. In addition, we found one flaw (*i.e.*, F18) that could only be reached using the reachability-based scheme. Another interesting observation is that both the Reachability and Complex Reachability schemes are similar in terms of usefulness in finding flaws, which may be a positive indicator of a general reachability-based approach over more complex strategies.

Based on the data in Table 3.5 alone, it may seem that only using the scope-based scheme is a viable option. However, recall from Section 3.5 that mutation schemes may also display widely different performance in terms of creating *executable* mutants. As a result,

using a single scheme may improve flaw detection, but may also generate a tremendous overhead in terms of non-executing mutants. Furthermore, even if the same mutation/leak placement can be achieved using multiple schemes, the common placement may help the researcher or tool developer resolve the flaw behind an uncaught mutant faster, *i.e.*, by helping them see the common or different factors (*i.e.*, among the schemes). For example, the Complex Reachability scheme makes the path between source and sink indirect while residing within the same scope, whereas the reachability scheme places the source and sink at the same location without establishing any indirect path. Failure to detect the leak placed through Complex Reachability would indicate that the flaw is due to the indirect path, rather than the location of the leak.

3.7 Flaw Propagation Study

The objective of this study is to determine whether flaws from one tool propagate to other tools, which are either implemented directly on top of the original tool, or inherit certain design attributes (**RQ₃**). To carry out this study, we utilized the minimal APKs created for each of our 25 flaws, and analyzed them with other tools built with the same security goal (*i.e.*, other data leak detectors excluding the tools from which the specific flaws were discovered). Each minimal APK contains only one type of flaw, *i.e.*, we built 25 APKs, one for each flaw mentioned in Tables 3.2, 3.3 and 3.4.

Moreover, in order to prevent tools from crashing due to backwards compatibility issues, we built several versions of the minimal APKs from the same code base, varying the SDK versions, as well as the build tools (*i.e.*, Android Studio and Gradle, vs building manually using Android SDK tools). This was done because APK building procedure has changed over the years, and as a result, many of the studied tools, which are well over 4-5 years old, would break for apps built using the latest build configuration and procedure. For example, Android Studio's Gradle based building overrides the target SDK version defined in the AndroidManifest.xml file, and also uses newer versions of build tools and

target platforms. We discovered that decompilers such as dare [217] used in some of the tools we analyzed do not function correctly when analyzing such builds (*i.e.*, either crash, or run into infinite loops). Considering these factors, we try our best to customize minimal apks for individual tools, in order to minimize crashes or timeouts, and only report results from variants of minimal APKs that worked., *i.e.*, resulted in a successfully completed analysis.

3.7.1 Propagation of FlowDroid’s Flaws (F1-F13)

To determine if the flaws present in FlowDroid are also present in other data leak detectors, as well as tools that inherit it, we checked if the newer release versions of FlowDroid (*i.e.*, v2.5, v2.5.1, v2.7.1), as well as 6 other tools (*i.e.*, Argus, DroidSafe, IccTA, BlueSeal, HornDroid, and DidFail), are susceptible to any of the flaws discussed in Table 3.2. Among these, flaw propagation across versions of FlowDroid was done for v2.5, v2.5.1 in our original μ SE study. As part of this extended study, due to availability of FlowDroid v2.7.1, we include it.

Results: Table 3.6 provides an overview of the propagation of F1-F13. In the Table 3.6, all the versions of FlowDroid are susceptible to the flaws discovered from our analysis of FlowDroid v2.0. Note that while we fixed the Fragment flaw and our patch was accepted to FlowDroid’s codebase, the latest releases of FlowDroid (*i.e.*, v2.5, v2.5.1, and v2.7.1) still seem to have this flaw. We have reported this issue to the developers.

A significant observation from Table 3.6 is that the tools that directly inherit FlowDroid (*i.e.*, IccTA, DidFail) are similarly flawed as FlowDroid. This is especially true when the tools do not augment FlowDroid in any manner, and use it as a black box (**RQ₃**). On the contrary, Argus, which is motivated by FlowDroid’s design, but augments it on its own, does not inherit as many of FlowDroid’s flaws.

BlueSeal, HornDroid, and DroidSafe use a significantly different methodology, and are also not susceptible to many of μ SE’s uncovered flaws for FlowDroid. Interestingly,

BlueSeal and DroidSafe are similar to FlowDroid in that they use Soot to construct a control flow graph and rely on it to identify paths between sources and sinks. However, BlueSeal and DroidSafe both augment the graph in novel ways, and thus do not exhibit most of the flaws found in FlowDroid.

Finally, it is important to note that our analysis does not imply that FlowDroid is weaker than the tools, which have fewer flaws in Table 3.6. However, it does indicate that the flaws discovered may be typical of the design choices made in FlowDroid and inherited by the tools such as IccTA and DidFail.

3.7.2 Propagation of Argus’s Flaws (F14-F22)

Unique to this extended study (*i.e.*, previously not reported in our original μ SE paper [50]), we analyzed the minimal APKs developed for F14-F22 with FlowDroid v2.5.1, v2.6, v2.6.1, v2.7, and v2.7.1; BlueSeal, DroidSafe, DidFail, HornDroid, and IccTA to examine their prevalence. Note that FlowDroid 2.0 had become unavailable at this point in the study (*i.e.*, deprecated in favor of later versions), which is why we focus on FlowDroid versions from 2.5 to 2.7.1.

Results: As shown in the Table 3.7, flaws found in Argus largely affect FlowDroid, HornDroid and IccTA. Specifically, recall that while some of the fragment and RecyclerView-based flaws were missing callbacks that could potentially be fixed with patches, some were design-level gaps in the data flow tracking performed by these tools. For example, in F14, an event listener reads sensitive data (*i.e.*, the source), and calls another method, which leaks the data (*i.e.*, the sink). While such event listeners may not be directly called from any of the lifecycle methods in the app, they are likely to be invoked when user interacts with the GUI components of the app. The fact that none of the RecyclerView-based (*i.e.*, F18-F22) or fragment-based (*i.e.*, F14-F17) leaks were detected by any of the tools we evaluated demonstrates the fragility of these tools in face of commonly-used Android GUI abstractions.

DroidSafe and BlueSeal crashed when analyzing the minimal APKs for F18-F22, while DidFail crashed for all flaws, *i.e.*, F14-F22. These crashes are expected, and primarily occur due to lack of support for newer API. To elaborate, DroidSafe was specifically built to focus on Android 4.4.1 (API 19), and hence crashes on encountering RecyclerView, *i.e.*, the set of APIs at the root of F14-F22, which was introduced in Android 5.0 (API 21). Similarly, BlueSeal customizes an older version of Soot for analysis, which fails when analyzing RecyclerView-based APKs. Finally, DidFail was built using customized but outdated versions of several other tools, namely FlowDroid, Epicc [220], and dare [217], and hence crashes on all the newer flaws. Note that while tools such as FlowDroid and dare may be separately updated and maintained, the current implementation of DidFail customizes these dependencies to a sufficient degree, which prevents us from simply replacing the dependency with a newer version. However, we can still infer that since DidFail uses FlowDroid as a component, it is likely to inherit all of FlowDroid’s flaws.

3.7.3 Propagation of HornDroid’s Flaws (F23-F25)

To understand whether the flaws we found in HornDroid propagate to other tools, we prepared minimal APKs based on the found flaws of HornDroid, and analyzed them using Argus, FlowDroid versions v2.5.1, v2.6, v2.6.1, v2.7, and v2.7.1; BlueSeal, DroidSafe, DidFail, and IccTA. None of these flaws, nor their propagation, were reported in our original μ SE paper [50].

Results: As shown in the Table 3.8, we find that Argus did not exhibit any of the fragment-based flaws identified in HornDroid. This is surprising, considering that HornDroid was susceptible to every single flaw discovered in Argus, as seen previously in Table 3.7. This finding may indicate that Argus may be relatively more sound than HornDroid in practice, in spite of the latter providing a formal proof of soundness.

Furthermore, although FlowDroid’s earlier versions (v2.5.1, v2.6.1) were able to detect leaks F23 and F24, FlowDroid version v2.7.1 was not. This is not an isolated case: recall

that our patch for another fragment flaw F5 [281] fixed it in FlowDroid v2.0, but future versions of FlowDroid (*i.e.*, v2.5 onwards) still exhibit the flaw, as seen in Table 3.6. Interestingly, we note that F23 and F24 are absent in IccTA, an approach that relies on FlowDroid. This is because IccTA uses an older version of FlowDroid as a component, which is resistant to these flaws, and hence, remains unaffected as well. Moreover, F25, a fragment-based flaw, is exhibited by all versions of FlowDroid. This finding is an indicator of the lack of systematic fragment support in FlowDroid, as well as other major tools, when in fact fragments are a widely-used GUI element that may contain data leaks.

Finally, the propagation study also allows us to derive certain general conclusions regarding the quality of the tools studied. First, we conclude that all of the tools we analyzed are incapable of finding leaks that are exhibited in RecyclerViews. Second, we find evidence to suggest that Argus is relatively better for detecting fragment-based leaks, relative to HornDroid and FlowDroid. This may seem counterintuitive considering there was only one fragment related flaw (F5) in FlowDroid, two in HornDroid (F23, F25) and four in Argus (F14-17). However, when we consider the propagation of fragment related flaws as well, Argus is affected by five (F5, F14-17), while HornDroid and FlowDroid (considering its latest release, v2.7.1) are both affected by seven (*i.e.*, both are affected by flaws F5, F14-17, and F23, F25). Thus, we argue that Argus provides more holistic support for fragments, relative to the other tools studied.

3.8 Discussion

μ SE has demonstrated efficiency and effectiveness at revealing real undocumented flaws in prominent Android security analysis tools. While experts in Android static analysis may be familiar with some of the flaws we discovered (*e.g.*, some flaws in FC1 and FC2), we aim to document these flaws for the entire scientific community. Further, μ SE indeed found some design gaps that were surprising to expert developers; *e.g.*, FlowDroid’s design does not consider callbacks in anonymous inner classes (flaws 8-9, Table 3.6), and in our

interaction with the developers of FlowDroid, they acknowledged handling such classes as a non-trivial problem. During our evaluation of μ SE we were able to glean the following pertinent insights:

Insight 1: *Most mutation schemes are generally effective.* While certain mutation schemes may be Android-specific, our results demonstrate limited dependence on these configurations. Out of the 25 flaws discovered using μ SE (*i.e.*, both in our USENIX’18 paper [50] as well as this extension), we discover that each mutation scheme is necessary for detecting certain flaws (*i.e.*, which may not be detected with other schemes), as shown in Section 3.6.4.

Insight 2: *Security-focused static analysis tools exhibit undocumented flaws that require further evaluation and analysis.* Our results clearly demonstrate that previously unknown security flaws or undocumented design assumptions, which can be detected by μ SE, pervade existing Android security static analysis tools. Our findings not only motivate the dire need for systematic discovery, fixing and documentation of unsound choices in these tools, but also clearly illustrate the power of mutation based analysis adapted in security context.

Insight 3: *Current tools inherit flaws from legacy tools.* A key insight from our work is that while inheriting code of the foundational tools (*e.g.*, FlowDroid) is a common practice, some of the researchers may not necessarily be aware of the unsound choices they are inheriting as well. As our study results demonstrate, when a tool inherits another tool directly (*e.g.*, IccTA inherits FlowDroid), all the flaws propagate.

Insight 4: *Tools which follow similar design principles but do not have a direct relationship (*e.g.*, inheriting a codebase), have similar flaws.* Through our experiments and evaluation performed in this extended study, we effectively demonstrate that FlowDroid, HornDroid, and Argus; three different static analysis tools built independently from each other, but which share similar design principles, can and do exhibit similar flaws. This indicates that certain unsound decisions or flaws may be tied to the common security goal or could

be occurring due to fundamental gaps in the high-level design decisions that are common across the board for such tools.

Insight 5 : *Flaws which were not present in previous versions of a static analysis tool can appear in later versions, as the tool evolves.* As we found in this extended study, certain flaws found in HornDroid were not present in earlier versions of FlowDroid but appeared in the latest version (Table 3.8). This shows that unsound choices can be made at any stage and at any iteration of software life cycle, and further establishes the necessity of automatically and systematically evaluating these tools. μ SE lays the groundwork for the development of such a holistic, dynamic, testing framework.

Insight 6: *As tools, libraries, and the Android platform evolve, security problems become harder to track down.* Due the nature of software evolution, all the analysis tools, underlying libraries, and the Android platform itself evolve asynchronously. A few changes in the Android API may introduce undocumented flaws in analysis tools. μ SE handles this fundamental obstacle of continuous change by ensuring that each version of an analysis tool is systematically tested, as we realize while tracking the Fragment flaw in multiple versions of FlowDroid.

Insight 7: *Benchmarks need to evolve with time.* While manually-curated benchmarks (e.g., DroidBench [32]) are highly useful as a “first line of defense” in checking if a tool is able to detect well-known flaws, the downside of relying too heavily on benchmarks is that they only provide a known, finite number of tests, leading to a false sense of security. Due to constant changes (insight #6) benchmarks are likely to become less relevant unless they are constantly augmented, which requires tremendous effort and coordination. μ SE significantly reduces this burden on benchmark creators via its suite of extensible and expressive security operators and mutation schemes, which can continuously evaluate new versions of tools. The key insight we derive from our experience building μ SE is that *while benchmarks may check for documented flaws, μ SE’s true strength is in discovering new flaws.*

3.9 Limitations

1) Soundness of μ SE: As acknowledged in Section 2.1.3, μ SE does not aim to supplant formal verification (which would be sound) and does not claim soundness guarantees. Rather, μ SE provides a systematic approach to semi-automatically uncover flaws in existing security tools, which is a significant advancement over manually-curated tests.

2) Manual Effort: Presently, the workflow of μ SE requires an analyst to manually analyze the result of μ SE (*i.e.*, uncaught mutants). However, as described in Section 3.5, μ SE possesses enhancements that mitigate the manual effort by dynamically eliminating non-executable mutants that would otherwise impose a burden on the analyst examining undetected mutants. In our experience, this analysis was completed in a reasonable time using the methodology outlined in Section 3.2.3.

3) Limitations of Execution Engine: Like any dynamic analysis tool, the EE will not explore all possible program states, thus, there may be a set of mutants marked as non-executable by the EE, that may actually be executable under certain scenarios. However, the CRASHSCOPE tool, which μ SE’s EE is based upon, has been shown to perform comparably to other tools in terms of coverage [201]. Future versions of μ SE’s EE could rely on emerging input generation tools for Android apps [196].

4) Dependency on Android Framework APIs: We designed μ SE to be as generic as possible, as discussed in Section 3.2.4 and Section 3.3. For example, the mutation seeding methodology relies on the AST of the target source code that *selects* the target location for mutation. As a result, as long as the Android framework changes through extension and target code are parse-able as AST, the seeding methodology will not have to be changed. Indeed, the base apps we used for μ SE (Table A.1 in the Appendix) rely on different versions of Android SDK with both Gradle [158] and pre-Gradle build system, which demonstrates the *versatility* of μ SE. On the other hand, calls to functions/methods from the Android framework APIs are introduced through mutation. Therefore, these

will have to be changed as Android Framework changes over time, as it is *not possible* to generalize such calls.

5) Adaptation to Different Goals: μ SE requires defining security operator through manual examination of the claims made by existing tools *i.e.*, the security goal of the concerned tool (Section 3.2.1). Further changes might be necessary for satisfying syntactical requirements related to the defined, new, security operator. For example, for `Cipher.getInstance`, the security operator will have to be enclosed within a try-catch scope because of its `throws-exception` signature. Thus, the implementation of security operators to mutate cryptographic APIs may require manual intervention by domain experts initially, but we expect it to be a one-time effort, similar to how we defined security operators for data leak APIs once for detecting flaws in this work. Finally, we note that μ SE’s modular implementation separates the operators from its core components, *i.e.*, the Mutation Engine (ME) and the Execution Engine (EE) simply seed and execute security operators as per the operator specification, and hence, are decoupled from the security goal per se. Hence, the implementation of μ SE framework would not have to change for different security goals, limiting the amount of code change to only the addition of goal-specific security operators.

3.10 Chapter Summary

We proposed the μ SE framework for performing systematic security evaluation of Android static analysis tools to discover (undocumented) unsound assumptions, adopting the practice of mutation testing from SE to security. μ SE not only detected major flaws in popular, open-source Android security tools, but also demonstrated how these flaws propagated to other tools that inherited the security tool or followed similar principles. With μ SE, we demonstrated how mutation analysis can be feasibly used for gleaning unsound assumptions in existing tools, benefiting developers, researchers, and end users, by making such tools more secure and transparent.

Table 3.2: Descriptions of flaws uncovered* in FlowDroid v2.0

ID: Flaw Name	Description
FC1: Missing Callbacks	
F1: DialogFragmentShow	FlowDroid misses the DialogFragment.onCreateDialog() callback registered by DialogFragment.show().
F2: PhoneStateListener	FlowDroid does not recognize the onDataConnectionStateChanged() callback for classes extending the PhoneStateListener abstract class from the telephony package.
F3: NavigationView	FlowDroid does not recognize the onNavigationItemSelectedListener() callback of classes implementing the interface NavigationView.OnNavigationItemSelectedListener.
F4: SQLiteOpenHelper	FlowDroid misses the onCreate() callback of classes extending android.database.sqlite.SQLiteOpenHelper.
F5: Fragments	FlowDroid 2.0 does not model Android Fragments correctly. We added a patch, which was promptly accepted. However, FlowDroid 2.5 and 2.5.1 remain affected. We investigate this further in the next section.
FC2: Missing Implicit Calls	
F6: RunOnUiThread	FlowDroid misses the path to Runnable.run() for Runnables passed into Activity.runOnUiThread().
F7: ExecutorService	FlowDroid misses the path to Runnable.run() for Runnables passed into ExecutorService.submit().
FC3: Incorrect Modeling of Anonymous Classes	
F8: ButtonOnClickToDialogOnClick	FlowDroid does not recognize the onClick() callback of DialogInterface.OnClickListener when instantiated within a Button's onClick="method_name" callback defined in XML. FlowDroid will recognize this callback if the class is instantiated elsewhere, such as within an Activity's onCreate() method.
F9: BroadcastReceiver	FlowDroid misses the onReceive() callback of a BroadcastReceiver implemented programmatically and registered within another programmatically defined and registered BroadcastReceiver's onReceive() callback.
FC4: Incorrect Modeling of Asynchronous Methods	
F10: LocationListenerTaint	FlowDroid misses the flow from a source in the onStatusChanged() callback to a sink in the onLocationChanged() callback of the LocationListener interface, despite recognizing leaks wholly contained in either.
F11: NSDManager	FlowDroid misses the flow from sources in any callback of a NsdManager.DiscoveryListener to a sink in any callback of a NsdManager.ResolveListener, when the latter is created within one of the former's callbacks.
F12: ListViewCallbackSequential	FlowDroid misses the flow from a source to a sink within different methods of a class obtained via AdapterView.getItemAtPosition() within the onItemClick() callback of an AdapterView.OnItemClickListener.
F13: ThreadTaint	FlowDroid misses the flow to a sink within a Runnable.run() method started by a Thread, only when that Thread is saved to a variable before Thread.start() is called.

* reported in our USENIX'18 paper [50]

Table 3.3: Descriptions of flaws uncovered in Argus v3.1.2.

ID: Flaw Name	Description
FC4: Incorrect Modeling of Asynchronous Methods	
F14: FragmentEventToExternalMethod	Fragment declared within an Activity class requires its click event listening methods to be defined in the Activity class. When we placed source in such click event listening method and the sink in an Activity method callable by, Argus missed the leak.
F15: FragmentCrossClickEventListeners	In similar construction of F14, when source and sink are distributed across click event listener methods connected to Fragment GUI components, Argus missed the leak.
F16: onCreateFragmentClickListener	Android Activity lifecycle method onCreate can be used to create a source for leak. If it is then leaked through a method called by a fragment component click event listening method, it is undetected by Argus.
FC1: Missing Callbacks	
F17: FragmentClickListener	When sink and source are placed in an event listener method defined in an Activity class are coupled with components in a Fragment through relevant XML resource file, Argus misses the leak. In similar construction to F14, if both source and sink are placed in event listener methods for fragment components, Argus does not report it.
F18: RecyclerViewHolder	A RecyclerView.ViewHolder abstract class is used to describe each item within a RecyclerView. This is required to be extended when used inside the extending class of RecyclerView.Adapter. When leak source and sink are placed within the scope of the class implementing the RecyclerView.ViewHolder, Argus is unable to detect it.
F19: RecyclerViewConstructor	This flaw is similar to F18, but where the leak source and sink are placed within the constructor of the class extending RecyclerView.ViewHolder. Argus is unable to detect the placed leak in this scenario.
F20: RecyclerViewOnCreateViewHolder	The class extending RecyclerView.Adapter implements the abstract method onCreateViewHolder, when ViewHolder needs to represent an item. If a leak is placed within onCreateViewHolder, Argus's analysis can't detect it.
F21: RecyclerViewOnBindViewHolder	Similar to RecyclerViewCreateViewHolder, the class extending RecyclerView.Adapter implements the abstract method onBindViewHolder, when ViewHolder needs to display the data at the specified position. If a leak is placed within onBindViewHolder, Argus's analysis cannot detect it.
F22: RecyclerViewGetItemCount	getItemCount is an abstract method required to be overridden to return the total number of items bound in RecyclerView. This method is placed within a class extending RecyclerView.Adapter. Argus is unable to find a leak if the source and sink are placed within getItemCount.

Table 3.4: Descriptions of flaws uncovered in HornDroid.

ID: Flaw Name	Description
FC5: Android Lifecycle Callbacks	
F23: OnCreateFragmentConstructor	HornDroid does not detect a leak if the source is placed in the onCreate method of an activity, and the sink is placed in the constructor of the fragment within the activity.
F24: ActivityOncreate	When a leak is placed in the onCreate method of Activity, i.e. both source and sink are placed in the method, HornDroid is unable to detect the leak.
F25: FragmentOnCreateView	When the source of the leak is placed at the onCreate method of activity class, and the sink at the onCreateView method of the fragment class, HornDroid does not detect the leak.

Table 3.5: Impact of Operator Placement Approaches in Finding Flaws.*

Flaw ID	Reachability	Complex-Reachability	Taint	Scope
Flaws found from FlowDroid v2.0				
F1	✓	✓	✓	✓
F2	✓	✓	✓	✓
F3	✓	✓	✓	✓
F4	✓	✓	✓	✓
F5	✓	✓	✓	✓
F6	✓	✓	✓	✓
F7	✓	✓	✓	✓
F8	✓	✓	✓	✓
F9	✓	✓	✓	✓
F10	-	-	✓	✓
F11	-	-	✓	✓
F12	-	-	✓	✓
F13	-	-	✓	✓
Flaws found from Argus v3.1.2				
F14	-	-	✓	✓
F15	-	-	✓	✓
F16	-	-	✓	✓
F17	✓	✓	✓	✓
F18	✓	-	-	-
F19	✓	✓	✓	✓
F20	✓	✓	✓	✓
F21	✓	✓	✓	✓
F22	✓	✓	✓	✓
Flaws found from HornDroid				
F23	✓	✓	✓	✓
F24	✓	✓	✓	✓
F25	-	-	-	✓

*A “✓” indicates flaw may be resolved through relevant operator placement approach, whereas “-” indicates it may not.

Table 3.6: Analysis of the propagation of flaws in FlowDroid 2.0 to other data leak detectors.*

Flaw ID	FD v2.0	FD v2.5	FD v2.5.1	FD v2.7.1	Argus	BlueSeal	DidFail	DroidSafe	HornDroid	IccTA
F1	✓	✓	✓	✓	x	x	✓	x	✓	✓
F2	✓	✓	✓	✓	x	x	✓	x	✓	✓
F3	✓	✓	✓	✓	✓	-	✓	-	-	✓
F4	✓	✓	✓	✓	✓	x	✓	x	✓	✓
F5	✓	✓	✓	✓	✓	✓	✓	-	✓	✓
F6	✓	✓	✓	✓	✓	x	✓	x	✓	✓
F7	✓	✓	✓	✓	✓	x	✓	x	✓	✓
F8	✓	✓	✓	✓	x	x	✓	✓	x	✓
F9	✓	✓	✓	✓	x	x	✓	x	x	✓
F10	✓	✓	✓	✓	x	x	✓	x	x	✓
F11	✓	✓	✓	✓	✓	x	✓	x	x	✓
F12	✓	✓	✓	✓	x	x	✓	x	x	✓
F13	✓	✓	✓	✓	x	x	✓	x	x	✓

* Note that a “-” indicates tool crash with the minimal APK, a “✓” indicates presence of the flaw, and a “x” indicates absence; FD* = FlowDroid.

Table 3.7: Analysis of the propagation of flaws in Argus 3.1.2 to other data leak detectors.*

Flaw ID	Argus	FD* v2.5.1	FD* v2.6	FD* v2.6.1	FD* v2.7	FD* v2.7.1	BlueSeal	DidFail	DroidSafe	HornDroid	IccTA
F14	✓	✓	-	✓	-	✓	✓	-	✓	✓	✓
F15	✓	✓	-	✓	-	✓	✓	-	✓	✓	✓
F16	✓	✓	-	✓	-	✓	✓	-	✓	✓	✓
F17	✓	✓	-	✓	-	✓	✓	-	✓	✓	✓
F18	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓
F19	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓
F20	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓
F21	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓
F22	✓	✓	✓	✓	✓	✓	-	-	-	✓	✓

* Note that a “-” indicates tool crash with the minimal APK, a “✓” indicates presence of the flaw, and a “x” indicates absence; FD* = FlowDroid.

Table 3.8: Analysis of the propagation of flaws in HornDroid to other data leak detectors.*

Flaw ID	Argus	FD* v2.5.1	FD* v2.6	FD* v2.6.1	FD* v2.7	FD* v2.7.1	BlueSeal	DidFail	DroidSafe	HornDroid	IccTA
-	Argus	FD2.5.1	2.6	2.6.1	2.7	2.7.1	BlueSeal	DidFail	DroidSafe	HornDroid	IccTA
F23	x	x	-	x	-	✓	x	-	✓	✓	x
F24	x	x	-	x	-	✓	x	-	✓	✓	x
F25	x	✓	-	✓	-	✓	x	-	✓	✓	✓

* Note that a “-” indicates tool crash with the minimal APK, a “✓” indicates presence of the flaw, and a “x” indicates absence; FD* = FlowDroid.

Chapter 4

Evaluating Crypto-API Misuse Detectors

Effective cryptography is critical in ensuring the security of confidential data in modern software. However, ensuring the *correct use* of cryptographic primitives has historically been a hard problem, whether we consider the vulnerable banking systems from Anderson’s seminal work [23], or the widespread misuse of cryptographic APIs (*i.e.*, *crypto-APIs*) in mobile and Web apps that can lead to the compromise of confidential financial or medical data and even the integrity of IoT devices [100, 243, 266, 283, 314, 307, 169]. In response, security researchers have developed a wide array of techniques and tools for detecting crypto-API misuse [176, 243, 100, 93, 266, 267, 300, 284, 263, 129, 257, 178] that can be integrated into the software development cycle, thereby preventing vulnerabilities at the source. These crypto-API misuse detectors, or *crypto-detectors*, play a crucial role in the security of end-user software.

Crypto-detectors have been independently used by developers for decades [47]. They are integrated into IDEs (*e.g.*, the CogniCrypt plugin for Eclipse [71]), incorporated in the internal testing suites of organizations (*e.g.*, CryptoGuard [243], integrated into Oracle’s testing suite [76]), or are currently targeted for commercialization and widespread deployment [243, 75]. In fact, several crypto-detectors are also being formally provisioned

by code hosting services as a way of allowing developers to ensure compliance with data security standards and security best-practices (*e.g.*, Github’s CodeScan initiative [120]). Thus, the importance of crypto-detectors in ensuring data security in modern Web and mobile software cannot be overstated, as key stakeholders (*i.e.*, researchers, code-hosting services, app markets, and developers) are increasingly reliant on them. However, what is concerning is that while stakeholders are optimistically adopting crypto-detectors, *we know very little regarding their actual effectiveness at finding crypto-API misuse*. That is, beyond manually-curated benchmarks, there is no approach for *systematically* evaluating crypto-detectors. This example in Listing 4.1 illustrates the gravity of this problem:

```
1 String algorithm = "DES";  
2 Cipher cipher = Cipher.getInstance(algorithm);
```

Listing 4.1: Instantiating “DES” as a cipher instance.

In this example, we define DES as our algorithm of choice, and instantiate it using the `getInstance` API from `Cipher`. Given that DES is not secure, one would expect any crypto-detector to detect this relatively straightforward misuse. However, two very popular crypto-detectors, *i.e.*, Tool_X¹ (used by over 3k+ open source Java projects), and QARK [186] (promoted by LinkedIn and recommended in security testing books [174, 141, 150]), are unable to detect this trivial misuse case as we discuss later in the chapter. Further, one might consider manually-curated benchmarks (*e.g.*, CryptoAPIBench [12], or the OWASP Benchmark [231]) as practical and sufficient for evaluating crypto-detectors to uncover such issues. However, given the scale and diversity of crypto protocols, APIs, and their potential misuse, benchmarks may be incomplete, incorrect, and impractical to maintain; *e.g.*, the OWASP benchmark considered using ECB mode with DES as secure until it was reported in March 2020 [232]. Thus, it is imperative to address this problem through a reliable and evolving evaluation technique that scales to the volume and diversity of crypto-API misuse.

In this chapter, we describe the first systematic, data-driven framework that leverages the well-founded approach of Mutation Analysis for evaluating Static Crypto-API misuse

¹We have anonymized this tool in the work as requested by its developers.

detectors – the MASC framework, pronounced as *mask*. Stakeholders can use MASC in a manner similar to the typical use of mutation analysis in software testing: MASC *mutates* Android/Java apps by seeding them with *mutants*, *i.e.*, code snippets exhibiting crypto-API misuse. These mutated apps are then analyzed with the crypto-detector that is the target of the evaluation, resulting in mutants that are *undetected*, which when analyzed further reveal design or implementation-level flaws in the crypto-detector. To enable this workflow for practical and effective evaluation of crypto-detectors, MASC addresses three key **research challenges** (RCs) arising from the unique scale and complexity of the problem domain of crypto-API misuse:

RC₁: *Taming the Complexity of Crypto-API Misuse* - An approach that effectively evaluates crypto-detectors must comprehensively express (*i.e.*, test with) relevant misuse cases *across all existing crypto-APIs*, which is challenging as crypto-APIs are as vast as the primitives they enable. For instance, APIs express the initialization of secure random numbers, creation of ciphers for encryption/decryption, computing message authentication codes (MACs), and higher-level abstractions such as certificate and hostname verification for SSL/TLS.

RC₂: *Instantiating Realistic Misuse Case Variations* - To evaluate crypto-detectors, code *instances* of crypto-API misuse must be seeded into apps for analysis. However, simply injecting misuse identified in the wild *verbatim* may not lead to a robust analysis, as it does not express the variations with which developers may use such APIs. Strategic and expressive *instantiation* of misuse cases is critical for an effective evaluation, as even subtle variations may evade detection, and hence lead to the discovery of flaws (*e.g.*, passing DES as a variable instead of a constant in Listing 4.1).

RC₃: *Scaling the Analysis* - Efficiently creating and seeding large numbers of *compilable* mutants *without significant manual intervention* is critical for identifying as many flaws in crypto-detectors as possible. Thus, the resultant framework must efficiently scale to thousands of tests (*i.e.*, mutants).

To address these research challenges, this chapter makes the following major contributions

in its conference version of the paper S&P'22 [19]:

- **Crypto-API Misuse Taxonomy:** We construct the first comprehensive taxonomy of crypto-API misuse cases (109 cases, grouped into nine clusters), using a data-driven process that systematically identifies, studies, and extracts misuse cases from academic and industrial sources published over the last 20 years. The taxonomy provides a broad view of the problem space, and forms the core building block for MASC's approach, enabling it to be grounded in real misuse cases observed in the wild (**RC₁**).
- **Crypto-Mutation Operators and Scopes:** We contextualize mutation testing for evaluating crypto-detectors by designing abstractions that allow us to instantiate the misuse cases from the taxonomy to create a diverse array of feasible (*i.e.*, *compilable*) mutants. We begin by formulating a threat model consisting of 3 adversary-types that represent the threat conditions that crypto-detectors may face in practice. We then design *usage-based mutation operators*, *i.e.*, general operators that leverage the common usage characteristics of diverse crypto-APIs, to expressively instantiate misuse cases from the taxonomy (addresses **RC₂**). Similarly, we also design the novel abstraction of *mutation scopes* for seeding mutants of variable fidelity to realistic API-use and threats.
- **The MASC Framework:** We implement the MASC framework for evaluating Java-based crypto-detectors, including 19 mutation operators that can express a majority of the cases in our taxonomy, and 3 mutation scopes. We implement the underlying static analysis to automatically instantiate thousands of *compilable* mutants, with manual effort limited to configuring the mutation operators with values signifying the misuse (**RC₃**).
- **Empirical Evaluation of Crypto-Detectors:** We evaluate 9 major crypto-detectors using 20,303 mutants generated by MASC, and reveal 19 previously unknown flaws (several of which are design-level). A majority of these discoveries of flaws in individual detectors (*i.e.*, 45/76 or 59.2%) are due to mutation (vs. being unable to detect the base/verbatim instantiations of the misuse case). Through the study of open source

apps, we *demonstrate that the flaws uncovered by MASC are serious and would impact real systems*. Finally, we disclose our findings to the designers/maintainers of the affected crypto-detectors, and further leverage these communication channels to obtain *their perspectives* on the flaws. These perspectives allow us to present a balanced discussion on the factors influencing the current design and testing of crypto-detectors, as well as a path forward towards more robust tool.

This study substantially extends upon the previous work, described as follows:

- **Updating the Taxonomy:** By applying the data-driven approach from the S&P'22 [19] paper for recent crypto-API misuse reported in industry and academic sources from the year 2019 to 2022, we have extended the crypto-API misuse taxonomy. We found 4 new misuse cases from literature, thus increasing the number of misuse from 105 to 109 (**RC₁**).
- **Additional Mutation Operators:** Bolstered by our experience of evaluating the crypto-detectors from the previous study, we have extended several mutation-operators to facilitate better evaluation of crypto-detectors with the goal of finding flaws. Furthermore, we have created additional mutation operators, which we used in the extended evaluation of crypto-detectors (**RC₂**).
- **Extended Evaluation of Additional crypto-detectors:** In this extension, we evaluated five crypto-detectors from industry, namely, SonarQube, Snyk, Codiga, DeepSource, and Amazon CodeGuru Security). Furthermore, we evaluated the updated versions of all the previously evaluated crypto-detectors with the original and additional mutations, except Xanitizer and Toolx. Moreover, we expanded the base applications for mutation by adding 15 open source, visible (at least 200 stars in GitHub) applications from the wild. With the combination of new base applications and new mutation operators, we created 30,236 new mutants, totaling 50,539 mutants, which we used to evaluate both the new version of previously used crypto-detectors, and newly acquired crypto-detectors in this study.

Additionally, we have made several maintainability and extensibility improvements in MASC framework over the original implementation of the S&P’22 [19] paper, which we detail in Section 4.5.

Artifact Release: To foster further research in the evaluation and development of effective cryptographic misuse detection techniques, and in turn, more secure software, we have released all code and data associated with this chapter [16].

4.1 The MASC Framework

We propose a framework for Mutation-based Analysis of Static Crypto-misuse detection techniques (or MASC). Fig. 4.1 provides an overview of the MASC framework. As described previously (**RC**₁), cryptographic libraries contain a sizable, diverse set of APIs, each with different potential misuse cases, leading to an exponentially large design space. Therefore, we initialize MASC by developing a *data-driven taxonomy of crypto-API misuse*, which grounds our evaluation in a unified collection of misuse cases observed in practice (Sec. 4.2).

The misuse cases in the taxonomy must be *instantiated* in an *expressive manner* to account for the diverse ways for expressing a misuse, *i.e.*, *misuse instances*, that crypto-detectors may face in practice. For example, we previously described two ways of encrypting with DES: (1) providing DES as a variable in `Cipher.getInstance(<parameter>)` (Listing 4.1), or (2) using it in lowercase (Listing 2.1), which both represent something a benign developer might do (*i.e.*, threat **T1**). To represent all such instances without having to hard-code instantiations for every misuse case, we identify *usage-characteristics* of cryptographic APIs (particularly, in JCA), and leverage them to define *general, usage-based mutation operators*, *i.e.*, functions that can create misuse instances (*i.e.*, *mutants*) by instantiating one or more misuse cases from the taxonomy (Sec. 4.3).

Upon instantiating mutants by applying our mutation operators to the misuse cases from the taxonomy, MASC *seeds*, *i.e.*, injects, the mutants into real Java/Android applications. The challenge here is to seed the mutants at specific locations that reflect the

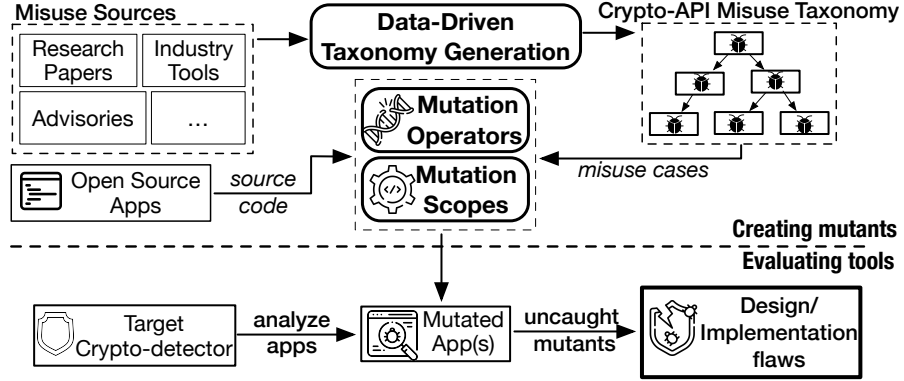


Figure 4.1: A conceptual overview of the MASC framework.

threat scenarios described in Sec. 2.2.2, because crypto-detectors may not only face various instances of misuse cases, but also variations in *where* the misuse instances appear, *e.g.*, evasive (T3) developers may attempt to actively hide code to evade analysis. Thus, we define the abstraction of *mutation scopes* that place the instantiated mutants at strategic locations within code, emulating practical threat scenarios (Sec. 4.4). Finally, we analyze these *mutated apps* with the crypto-detector that is targeted for evaluation (Sec. 4.6), which results in *undetected/unkilled mutants* that can be then inspected to uncover design or implementation flaws (Sec. 4.7).

4.2 Taxonomy of Cryptographic Misuse

To ground MASC in real cases of crypto API misuses, we systematically developed a taxonomy that provides a unified perspective of previously-known crypto-API misuse. Particularly, we focus on identifying instances of crypto-API misuse in the popular and ubiquitous Java ecosystem, as most crypto-detectors that we seek to evaluate were designed to analyze Java/Android apps.

As crypto-API misuse has been widely studied, it is likely that a majority of the misuse cases that we are interested in codifying are already present in existing literature. Therefore, our methodology identifies crypto-API misuses in existing artifacts sourced from

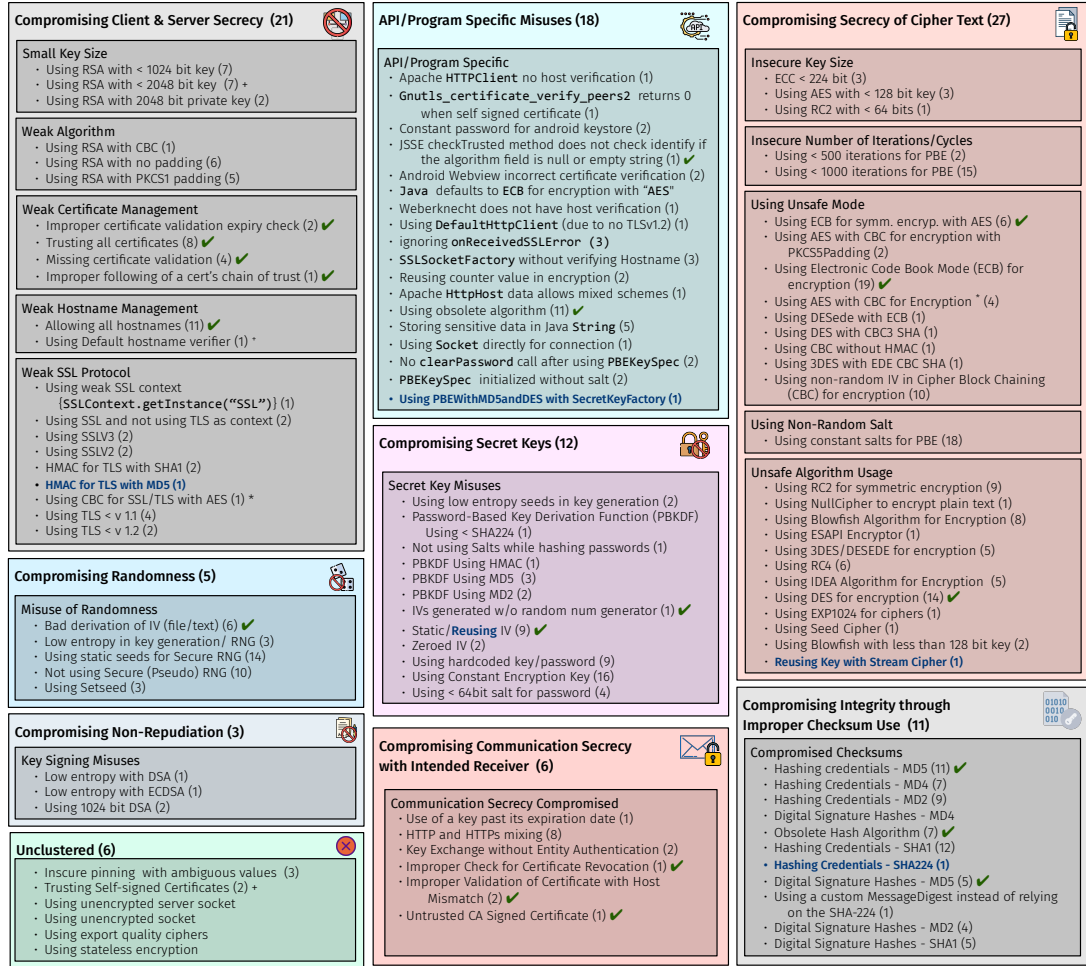


Figure 4.2: The derived taxonomy of cryptographic misuses. (n) indicates misuse was present across n artifacts. A ✓ indicates that the specific misuse case was instantiated with MASC's mutation operators for our evaluation (Sec. 4.6). New misuse cases found as part of this extension are highlighted in Blue.

both industry and academia, following Kitchenham et al.'s [171] guidelines for identifying relevant artifacts, as well as Petersen et al.'s [238] recommendations for constructing a systematic mapping, in three main steps: (1) identifying information sources, (2) defining the search, inclusion, and exclusion criteria for identifying artifacts, and (3) extracting misuse cases from artifacts and clustering them for ease of representation and extensibility. Two authors executed this methodology and created the taxonomy illustrated in Fig. 4.2. Data from each step is provided in the original artifact [15].

4.2.1 Identifying Information Sources

We considered information sources from both academia and industry. More specifically, we considered the proceedings of top-tier venues in security and software engineering (*i.e.*, USENIX Security, ACM CCS, IEEE S&P, NDSS, ICSE, ASE, FSE), *published after 1999, i.e., in the last 20 years*. Moreover, we also performed a thorough search for relevant keywords (Sec. 4.2.2) in digital libraries, *i.e.*, the ACM Digital Library, IEEE Explore, and Google Scholar, which aided in identifying artifacts that may have fallen outside the top conferences. Finally, to incorporate sources outside academia, we studied guidelines from the Open Web Application Security Project (OWASP) [230], and documentation of several industry tools.

4.2.2 Search, Inclusion, and Exclusion Criteria

We select artifacts from the identified sources using a keyword-based search with precise inclusion/exclusion criteria. We defined 3 classes of keyword phrases and enumerated several keyword combinations for each class, drawing from domain expertise and a random sample of artifacts.

To decide whether to consider an artifact for further analysis, we defined a simple ***inclusion criterion***, that the artifact should discuss crypto API misuse or its detection. We also defined an ***exclusion criterion***, *i.e.*, that the crypto-API misuse described by the artifact relates to a programming environment outside the Java ecosystem, was published prior to 1999, or does not contain relevant information. Following this methodology, we short-listed 40 artifacts for misuse extraction, *i.e.*, 35 from academia and 5 from industry. Note that we count multiple documents for a single industry tool as one artifact.

4.2.3 Misuse Extraction and Clustering

Two authors independently extracted, described, and grouped individual misuse cases from the 40 artifacts. More specifically, each identified misuse case was labeled using a

specifically designed data extraction form (see online appendix for figure [16]). The two authors met and resolved disagreements, to eventually identify *109 unique misuse cases*.

Such a large set of misuse cases could prove intractable for direct analysis or extension. Hence, we constructed a *categorized* taxonomy by grouping the discovered misuse cases into *semantically meaningful clusters*. Each author constructed the clusters as per two differentiating criteria: (1) the *security goal/property* represented by the misuse cases (*e.g.*, secrecy, integrity, non-repudiation), and (2) its *level of abstraction* (*i.e.*, specific context) within the communication/computing stack (*e.g.*, confidentiality in general, or confidentiality with respect to SSL/TLS). The two authors met and reached agreement on a taxonomy consisting of *105 misuse cases* grouped into *nine semantic clusters*, as shown in Fig. 4.2. The process of taxonomy generation took over *two person-months* effort.

4.2.4 Extending the Taxonomy

To expand the taxonomy of crypto-API misuse cases for the current iteration of our work, we applied the existing methodology, covering information sources from both academia and industry, specifically focusing within the year range 2019 - 2022. Additionally, we manually went through through the Common Weakness Enumeration (CWE) database to identify crypto-API misuse cases. As a result, we were able to identify 4 crypto-API misuse cases from 19 additional information sources, as highlighted in blue color in the Taxonomy (Figure 4.2).

4.3 Usage-based Mutation Operators

In designing our mutation operators, we must balance the dichotomous tradeoff between representing as many misuse cases (and their corresponding variations) as possible, while also creating a tractable number of operators that can be reasonably maintained in the future. Thus, building a large set of hard-coded operators that are tightly coupled with specific misuse cases would be infeasible from an engineering/maintenance perspective.

Further, to discover new issues in crypto-detectors, these operators should not exploit general soundness-related [265, 190]) limitations, such as dynamic code execution and implicit calls. Therefore, we seek to build operators that are general enough to be maintainable, but which also provide expressive instantiation of several misuse cases, guided by the threat model in Section 2.2.2, and without focusing on any specific static analysis technique or soundness issue.

We define the abstraction of *usage-based mutation operators*, inspired by a key observation: misuse cases that are unrelated in terms of the security problem may still be related in terms of *how the crypto APIs corresponding to the misuse cases are expected to be used*. Thus, characterizing the common usage of crypto APIs would allow us to mutate that characterization and define operators that apply to multiple misuse cases, while remaining independent of the specifics of each misuse.

Common Crypto-API Usage Characteristics: We identified two common patterns in crypto-API usage by examining crypto-API documentation from JCA, and our taxonomy, which we term as (1) *restrictive* and (2) *flexible* invocation. To elaborate, a developer can only instantiate certain objects by providing values from a predefined set, hence the name *restrictive* invocation; *e.g.*, for symmetric encryption with the `Cipher.getInstance(<parameter>)` method, JCA *only* accepts predefined configuration values for the algorithm name, mode, and padding, in String form. Conversely, JCA allows significant extensibility for certain crypto APIs, which we term as *flexible* invocation; *e.g.*, developers can customize the hostname verification component of the SSL/TLS handshake by creating a class extending the `HostnameVerifier`, and overriding its `verify` method, with *any content*. We leverage these notions of *restrictive* & *flexible* usage to define our operator types.

4.3.1 Operators based on Restrictive Crypto API Invocation

Our derived taxonomy indicates that several parameter values used in restrictive API invocations may not be secure (*e.g.*, DES, or MD5). Therefore, we designed six mutation operator types (**OPs**) that apply a diverse array of transformations to such values and generate API invocations that are acceptable/compilable as per JCA syntax and conventions, but not secure.

OP₁: Atypical case – This operator changes the case of algorithm specification misuse to an atypical form (*e.g.*, lowercase), and represents accidental misuse/typos by developers (*i.e.*, **T1**). For example, as previously shown in Listing 4.1, this operator instantiates the DES misuse by specifying “des” (lowercase) in the `Cipher.getInstance(<parameter>)` API.

OP₂: Weak Algorithm in Variable – This operator represents the relatively common usage of *expressing API arguments in variables* before passing them to an API, and can be applied to instantiate all misuse cases that are represented by restrictive API invocations (*e.g.*, DES instantiation in Listing 2.1).

OP₃: Explicit case fix – This operator instantiates misuse cases by using the atypical case for an argument (*e.g.*, algorithm name) in a restrictive API invocation, as seen in **OP₁**, but also explicitly *fixes* the case, emulating a developer attempting to follow conventions by explicitly invoking String manipulation methods (*i.e.*, **T2**); *e.g.*, `SSLContext.getInstance("ssl".toUpperCase())` is one instantiation of the misuse of using SSL2 or SSLv3 protocols.

OP₄: Removing noise – This operator extends **OP₃** by defining transformations that are more complex than simple case changes, such as removing extra characters or “noise” present in the arguments, which is likely if the arguments are acquired from a properties/database file; *e.g.*, `Cipher.getInstance("DES//".replace("//", ""))`.

OP₅: Method chains – This operator performs arbitrary transformations (*e.g.*, among those in **OP₁ – OP₄**) on the restricted argument string, but splits the transforma-

tion into a chain of procedure calls, thereby hiding the eventual value (see Listing A.2 in Online Appendix [16]). Such behavior can be attributed to an evasive developer (**T3**).

OP₆: Predictable/Non-Random Derivation – This operator emulates a benign developer (**T1**) attempting to *derive a random value* (*i.e.*, instead of using a cryptographically-secure random number generator), by performing seemingly complex operations resulting in a predictable value, and then using the derived value to obtain other cryptographic parameters, such as IVs. For example, Listing A.3 in Online Appendix [16] shows such an instantiation of the “Bad derivation of IV” misuse from the taxonomy that uses the system time to derive the IV.

We relied on the **T3** threat model while creating additional mutation operators in this extension for two reasons. First, our S&P’22 [19] paper demonstrated that the crypto-API misuse cases MASC creates is similar to misuse cases found in the wild. However, the real-life misuse cases can be based on more complex abstractions while still being statically analyzable [195]. Second, practitioners expect SASTs, including crypto-detectors, to report any vulnerabilities that fall within the scope of program analysis techniques [22]. Considering these two factors, we extended several of the existing mutation operators of MASC and created additional mutation operators to further improve the evaluation capabilities of MASC. Next, we discuss the mutation-operators built for this extension.

OP₁₃: Iterative Method Chaining – Extending **OP₅**, this operator takes a user-specified, arbitrary numeric value (**n**) and creates **n** methods. Furthermore, those method calls are then chained in succession. Similar to **OP₅**, user can specify two return values, one of which is used for **n**th method, whereas the other is returned from the rest of the methods. The goal of this operator is to analyze the internal limits of crypto-detectors when following a method chain.

OP₁₄: Iterative Nested Conditionals – This mutation operator extends **OP₁₃** and creates nested conditionals based on user-specified, arbitrary numeric value. The resulting misuse consists of two nested branches. The “*true*” branch always returns an insecure parameter after traversing, whereas the other returns a secure parameter. Mutations

created through **OP₁₄** can be used to evaluate whether a crypto-detector is following through the expected length of nested conditions.

OP₁₅: *Method Builder* – This operator breaks down a crypto-API String parameter into a chain of method calls. Furthermore, it then creates a concatenation method that invokes the character containing methods to chain together the string parameter. For example, for the parameter “DES”, this operator creates three methods. Combined, these methods return ‘D’, ‘E’, and ‘S’. The concatenation method then adds the letters together by calling each method and returns the final, concatenated value as the parameter at the crypto-API call site. This operator also simulates (**T3**) behavior.

OP₁₆: *Object Sensitivity* – To evaluate object sensitivity of a crypto-detector, this operator creates two objects, `varA` and `varB`. Together, this pair of objects are initialized to contain a pair of secure and insecure parameters for the target crypto-API. Next, assuming that `varB` contains secure parameter, `varA` is assigned to `varB`. Finally, `varB` is passed to crypto-API call site.

OP₁₇: *Build Variable* – This operator at first introduces an array of characters based on a pre-specified string parameter. Then, it passes the string value of the array to a target crypto API by using the `toString()` without doing any additional operations, emulating a **T1** developer.

OP₁₈: *Substring* – This operator creates a random string with an insecure parameter as a substring, *e.g.*, `''HelloWorldDES''`. Next, it uses the `substring` method to extract the insecure parameter from the original string and passes it to the target crypto-API, *e.g.*, `''HelloWorldDES''.substring(10);`.

OP₁₉: *Constant Value Derivation* – This operator is similar to **OP₆**, as in it derives a constant/predictable value, without using any complex operations. For example, Listing A.23 shows such an instantiation of constant IV derivation from a fixed string.

4.3.2 Operators based on Flexible Crypto API Invocations

In contrast with restrictive APIs, Java allows several types of flexible extensions to crypto APIs represented by interfaces or abstract classes, *only enforcing typical syntactic rules*, with little control over what semantics developers express. Thus, we consider three particular types of flexible extensions developers may make and our **OPs** may emulate, in the context of API misuse cases that involve flexible invocations: (1) *method overriding*, (2) *class extension*, and (3) *object instantiation*.

4.3.2.1 Method Overriding

Crypto APIs are often declared as interfaces or abstract classes containing abstract methods, to facilitate customization. These abstract classes provide a fertile ground for defining mutation operators with a propensity for circumventing detectors (*i.e.*, considering threats **T3**).

OP₇: Ineffective Exceptions – If the correct behavior of a method is to throw an exception, such as invalid certificate, this operator creates misuse instances of two types: (1) not throwing any exception, and (2) throwing an exception within a conditional block that is only executed when a highly unlikely condition is true. For example, as shown in Listing A.4 in Online Appendix [16], this operator instantiates a weak `TrustManager` by implementing a `checkServerTrusted` method containing a condition block is unlikely to be executed.

OP₈: Ineffective Return Values – If the correct behavior of a method is to return a specific value to denote security failure, this operator modifies the return value to create two misuse instances: (1) if the return type is boolean, return a predefined boolean value that weakens the crypto feature, or return *null* otherwise, and (2) introduce a condition block that will always, prematurely return a misuse-inducing boolean/null value before the secure return statement is reached. Contrary to **OP₇**, this operator ensures that the condition will always return the value resulting in misuse (see Listing A.5 in Online

Appendix [16] .

OP₉: Irrelevant Loop – This operator adds loops that *seem to perform some work*, or security checks, before returning a value, but in reality *do nothing to change the outcome*, emulating an evasive (T3) developer.

4.3.2.2 Class Extension

Creating abstractions on top of previously-defined on crypto classes is fairly common; *e.g.*, the abstract class `X509ExtendedTrustManager` implements the interface `X509TrustManager` in the JCA, and developers can be expected to extend/implement both these constructs in order to customize certificate verification for their use-cases. Similarly, developers may create abstract subtypes of an interface or abstract class; *e.g.*, as shown in Listing A.6 (Online Appendix [16]), the `X509TrustManager` can be both extended and implemented by an abstract type interface and an abstract class respectively. Our next set of operators is motivated by this observation:

OP₁₀: Abstract Extension with Override – This mutation operator creates an abstract sub-type of the parent class (*e.g.*, the `X509TrustManager` as shown in Listing A.6), but this time, overrides the methods *incorrectly*, *i.e.*, adapting the techniques in **OP₇ – OP₉** for creating various instances of misuse.

OP₁₁: Concrete Extension with Override – This mutation operator creates a concrete class based on a crypto API, *incorrectly* overriding methods similar to **OP₁₀**.

4.3.2.3 Object Instantiation Operators

In Java, objects can be created by calling either the default or a parametrized constructor, and moreover, it may also be possible to override the properties of the object through Inner class object declarations. We leverage these properties to create **OP₁₂** as follows:

OP₁₂: Anonymous Inner Object – Creating an instance of a flexible crypto API through constructor or anonymous inner class object is fairly common, as seen for `HostnameVerifier` in Oracle Reference Guide [229] and Android developer documenta-

tion [25], respectively. Similarly, this operator creates anonymous inner class objects from abstract crypto APIs, and instantiates misuse cases by overriding the abstract methods using **OP**₇ – **OP**₉, as shown in Listing A.7 (Online Appendix [16]), where the misuse is introduced through **OP**₈.

MASC’s 19 operators are capable of instantiating **69/105** (65.71%), misuse cases distributed across *all 9 semantic clusters*. This indicates that MASC’s operators can express a *diverse majority* of misuse cases, signaling a reasonable trade-off between the number of operators and their expressivity. Of the remaining 36 cases that our operators do not instantiate, 16 are trivial to implement (*e.g.*, using AES with a < 128 bit key, see Listing A.13, Online Appendix [16]). Finally, 20 cases (19.01%) would require a non-trivial amount of additional engineering effort; *e.g.*, designing an operator that uses a custom MessageDigest algorithm instead of a known standard such as SHA-3, which would require a custom hashing algorithm.

4.4 Threat-based Mutation Scopes

We seek to emulate the typical placement of vulnerable code by benign (**T1**, **T2**), and evasive (**T3**) developers, for which we design three *mutation scopes*:

1. Main Scope: The *main scope* is the simplest of the three, and seeds mutants at the beginning of the main method of a simple Java or Android template app developed by the authors (*i.e.*, instead of the real, third-party applications mutated by the other two scopes). This specific seeding strategy ensures that the mutant would always be reachable and executable, emulating basic placement by a benign developer (**T1**, **T2**).

2. Similarity Scope: The *similarity scope* seeds security operators at locations in a target application’s source code where a similar API is already being used, *i.e.*, akin to modifying existing API usages and making them vulnerable. Hence, this scope emulates code placement by a typical, well-intentioned, developer (**T1**, **T2**), assuming that the target app we mutate is also written by a benign developer. This helps us evaluate if the

crypto-detector is able to detect misuse at *realistic* locations.

3. *Exhaustive Scope*: As the name suggests, this scope *exhaustively* seeds mutants at *all locations* in the target app’s code, *i.e.*, class definitions, conditional segments, method bodies as well as anonymous inner class object declarations. Note that some mutants may not be seeded in all of these locations; *e.g.*, a `Cipher.getInstance(<parameter>)` is generally surrounded by `try-catch` block, which cannot be placed at class scope. This scope emulates placement by an evasive developer (**T3**).

4.5 Implementation

The first iteration of MASC, as discussed in our S&P’22 [19] paper, involved three components, namely, (1) selecting misuse cases from the taxonomy for mutation, (2) implementing mutation operators that instantiate the misuse cases, and (3) seeding/inserting the instantiated mutants in Java/Android source code at targeted locations.

1. Selecting misuse cases from the Taxonomy: We chose 19 misuse cases from the taxonomy for mutation with MASC’s 19 operators (indicated by a ✓ in Fig. 4.2), focusing on ensuring broad *coverage* across our taxonomy as well as on their *prevalence*, *i.e.*, prioritizing misuse cases that are discussed more frequently in the artifacts used to construct the taxonomy, and which most crypto-detectors can be expected to target. We expand these choices in the S&P’22 [19] paper, and in the Online Appendix [16].

2. Implementing mutants: The mutation operators described in Sec. 4.3 are designed to be applied to one or more crypto APIs, for instantiating specific misuse cases. To ensure compilable mutants by design, MASC carefully considers the syntactic requirements of the API being implemented (*e.g.*, the requirement of a surrounding try-catch block with appropriate exception handling), as well as the semantic requirements of a particular misuse being instantiated, such as the need to throw an exception only under a truly improbable condition (*e.g.*, as expressed in **OP**₇). MASC uses Java Reflection to deter-

mine all the “syntactic glue” for automatically instantiating a compilable mutant, *i.e.*, exceptions thrown, requirements of being surrounded by a try-catch block, the need to implement a certain abstract class and certain methods, etc. MASC then combines this automatically-derived and compilable syntactic glue with parameters, *i.e.*, values to be used in arguments, return statements, or conditions, which we manually define for specific operators (and misuse cases), to create mutants.

To further ensure compilability and evaluation using only compilable mutants, we take two steps: (1) We use Eclipse JDT’s AST-based checks for identifying syntactic anomalies in the generated mutated apps, and (2) compile the mutated app automatically using build/test scripts *provided with the original app*. In the end, every single mutant analyzed by the target crypto-detector is compilable and accurately expresses the particular misuse case that is instantiated. This level of automation allows MASC to create thousands of mutants with very little manual effort, and makes MASC extensible to future evolution in Java cryptographic APIs (addressing **RC**₃).

3. Identifying Target Locations and Seeding Mutants: To identify target locations for the similarity scope, we extended the MDroid+ [200, 184] mutation analysis framework, by retargeting its procedure for identifying suitable mutant locations, adding support for dependencies that crypto-based mutations may introduce, and enabling identification of anonymous inner class objects as mutant-seeding locations, resulting in 10 additional, custom AST- and string-based location detectors. Further, MASC extends μ SE [51] to implement the *exhaustive* scope, *i.e.*, to identify locations where crypto-APIs can be feasibly inserted to instantiate compilable mutants (*e.g.*, a `Cipher.getInstance(<parameter>)` has to be contained in a try-catch block, making it infeasible to insert at class-level).

Based on our prior experience of developing and using the implementation of the MASC framework, we determined several design goals, namely Diversity of Crypto-APIs (**DG1**), Open to Extension (**DG2**), ease of evaluating crypto-detectors (**DG3**), and adapting to users with different levels of skills (**DG4**), as we demonstrated at the FSE’23 [18].

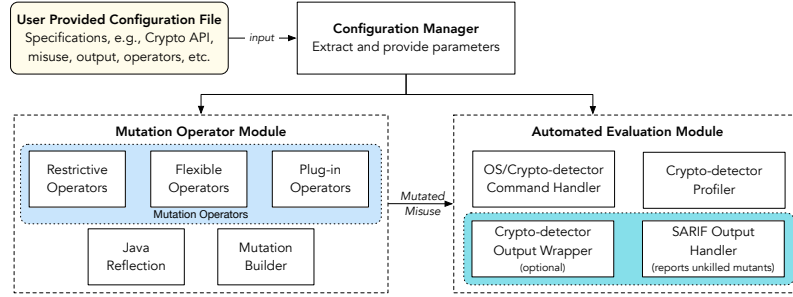


Figure 4.3: Architecture Overview of the Main Scope of MASC

```

scope = main # name of the scope to be used for mutation
type = StringOperator #name of the operator tor be used for mutation
outputDir = app/outputs
apiName = javax.crypto.Cipher
invocation = getInstance # Method call from crypto-API
secureParam = AES/GCM/NoPadding # Secure parameter to use with crypto-API
insecureParam = AES # insecure parameter to use with crypto-API
noise = ~ # noise value used with mutation
variableName = cryptoVariable # variable, class name used to create necessary
structures
className = CryptoTest
appName = <Name of the App> # name of the app for similarity-scope specific mutation

```

Listing 4.2: Example configuration file for MASC

To satisfy these design goals (**DG1–DG4**), we implemented MASC (11K+ effective Java source line of code) following single/responsibility principle across modules, classes, and functions. While current implementation of MASC inherits the *mutation scopes* of the original implementation with internal structural changes, the bulk of the changes with new features in the current implementation of MASC are based on the *Main Scope*. Therefore, we describe the implementation details of MASC with a focus on *Main Scope* in the context of the design goals and provide an overview of the architecture in Figure 4.3.

Configuration Manager: To make MASC as flexible as possible, we decoupled the crypto-API specific parameters from the internal structure of MASC. As a result, user can specify any crypto-API along with its necessary parameters through an external configuration file defining the base crypto-API misuse case. The configuration file follows a key-value format, as shown in Listing 4.2. Additionally, user can specify the mutation operators and scope to be used, along with other configuration values, thus satisfying **DG1**.

Mutation Operator Module: MASC analyzes the specified crypto-API and uses the values specified by the user (*e.g.*, secure, and insecure parameters to be used with the API) for creating mutated crypto-API misuse instances. Internally, the decoupling of crypto-APIs from MASC is made possible through the use of *Java Reflection* based API analysis and Java Source Generation using the *Java Poetry* Library (**DG1**). While both the original and current implementation of MASC comes with several generalizable mutation operators, the current implementation of MASC includes an additional plug-in structure that facilitates creating custom mutation operators and custom key-value pairs for the configuration file. Both of these can be done *externally*, *i.e.*, no modification to source code of MASC is necessary (**DG2**). We provide additional details about MASC’s mutation operators in Section 4.3.

Automated Evaluation Module: The current implementation of MASC leverages the SARIF formatted output to automate evaluation of crypto-detectors. To make end-to-end analysis automated, MASC’s can be configured to use crypto-detector specific commands, such as *e.g.*, compiling a mutated source code for analysis, evaluation stop conditions, command for running crypto-detector, output directory, and more (**DG3–DG4**).

Finally, MASC is implemented to produce verbose logs. With the combination of flexible configuration, it is therefore possible to use the stand-alone binary MASC jar file as a module of another software. As a proof of concept, we implemented MASC Web, a *python-django* based front-end that offers all the functionalities of the MASC that uses the binary jar of MASC as a module (**DG4**).

4.6 Evaluation Overview and Methodology

The two main goals of our evaluation are to (1) measure the effectiveness of MASC at uncovering flaws in crypto-detectors, and (2) learn the characteristics of the flaws and their real-world impact, in terms of the security of end-user applications. Therefore, we formulate the following research questions:

- **RQ₁**: *Can MASC discover flaws in crypto-detectors?*
- **RQ₂**: *What are the characteristics of these flaws?*
- **RQ₃**: *What is the impact of the flaws on the effectiveness of crypto-detectors in practice?*

To answer **RQ₁** – **RQ₃**, we performed experiments throughout several years, spanning from 2021 to 2024. As part of our S&P’22 [19] paper, we first used MASC to evaluate a set of *nine* major crypto-detectors, namely CryptoGuard, CogniCrypt, Xanitizer, Tool_x, SpotBugs with FindSecBugs, QARK, LGTM, Github Code Security (GCS), and ShiftLeft Scan, prioritizing practically relevant tools that were recent and under active maintenance. As part of this study, we evaluated all of these crypto-detectors with their updated versions, except Tool_x and Xanitizer, in addition to five new crypto-detectors, namely Amazon CodeGuru Security, Snyk, DeepSource, Codiga, and SonarQube, totaling 12 crypto-detectors.

To be consistent, we addressed several changes in crypto-detectors life-cycle during this study. First, LGTM was merged with Github Code Security and ceased to exist separately. As Github Code Security still stores the test suites from LGTM suites, we used Github Code Security with *lgtm* test suite as a proxy for the LGTM service. Furthermore, Github Code Security has two regularly maintained, security-specific test suites ². First, the *default* test suite is described to be more precise by giving “*fewer low confidence*” results to reduce false-positives. Second, *security-extended* is optionally available, and consists of all rules from the *default* test suite and additional rules with lower precision. Since we are interested in identifying flaws in crypto-detectors, we chose *security-extended* test suite for this extended study for evaluating Github Code Security. Finally, several of these new crypto-detectors are web-based and can only be used as part of CI/CD workflow. As those crypto-detectors do not offer a “fixed version” for evaluation, we used the latest version whenever possible, and confirmed the existence of the flaws as of *April 2024*.

²<https://docs.github.com/en/code-security/code-scanning/managing-your-code-scanning-configuration/codeql-query-suites>

As MASC’s usefulness is in systematically evaluating individual crypto-detectors by characterizing their detection-ability, with the goal of enabling improvement in the tools, and hence, the results of our evaluation indicate gaps in individual tools, and not comparative advantages.

Step 1 – Selecting and mutating apps: During the first iteration of this study for our S&P’22 [19] paper, we used MASC to mutate 13 open source Android apps from F-Droid [102] and Github [123], and four sub-systems of Apache Qpid Broker-J [242], a large Java Apache project (for list see [15]). Our selection criteria was biased towards popular projects that did not contain obsolete dependencies (*i.e.*, compilable using Android Studio 4/Java LTS 1.8). Moreover, we specifically used the similarity scope on 3/13 Android apps, and all 4 Qpid sub-systems, as they contained several types of crypto API usage (*e.g.*, Cipher, MessageDigest, X509TrustManager). In total, we generated 2,515 mutants using the 13 Android apps (producing `src` and `apk`) and 17,788 mutants using the 4 Java programs (producing `src` and `jar`), totaling 20,303 mutants. We confirmed that each mutated app was compilable and runnable. Generating these 20k mutants took MASC roughly 15 minutes, and did not require any human intervention, addressing **RC**₃. As the cost to generate this volume of mutants is feasible, MASC may not benefit from generating a focused subset of mutants (as we detail in S&P’22 [19]).

In this extension of the study, for mutation using MASC, we selected 10 additional Android applications for the exhaustive scope, four Android applications for the selective scope, and one Java-based application from Google. Similar to the previous study, these apps were chosen based on few specific criteria to represent popular, regularly updated applications, such as having more than 200 stars in GitHub for Android-based projects, and latest commit dating back to 2022, and having no obsolete dependencies. Additionally, we relied on GitHub’s advance code search feature to identify Android applications with existing crypto-API calls to ensure that those apps can be mutated using MASC’s selective scope. Finally, we manually confirmed that those 5 newly selected applications indeed

contained crypto-API call sites.

Using the extended MASC with its additional mutation operators, we mutated these 15 applications and generated 30,236 new mutants. We used 20,303 from the previous study, and 30,236 from this study, totaling 50,539 mutations, for evaluating the crypto-detectors.

Step 2 – Evaluating crypto-detectors and identifying *unkilled/undetected* mutants: To evaluate a crypto-detector, we analyzed the mutants using the crypto-detector, and identified the mutants that were *not killed* by it, *i.e.*, *undetected* as misuse using similar methodology in both the previous and current study. To facilitate accurate identification of killed mutants, we compare the mutation log MASC generates when inserting mutants (which describes the precise location and type of mutant injected, for each mutant), and the reports from crypto-detectors, which for all the tools contained precise location features such as the class/method names, line numbers, and other details such as associated variables. To elaborate, we use the following methodology: We first compare the analysis report generated by the crypto-detector/target on a mutated app, with its analysis report on the original (*i.e.*, unmutated) version of the same app. Any difference in the two reports can be attributed to mutants inserted by MASC, *i.e.*, denotes an instance of a “mutant being killed”. To identify the mutant that was killed, we obtain location features (*i.e.*, type, file, class, method, line number and/or variables associated) of the specific “killed mutant” from the crypto-detectors report on the mutated app, and use them to search for a unique mutant in MASC’s mutation log. If a match is found, we mark that specific mutant as killed. We additionally confirm the location of each killed mutant by referring to the mutated app’s source code. Once all the killed mutants are identified, the remaining mutants (*i.e.*, inserted in MASC’s mutation but not killed) are flagged as unkilld.

This approach ensures that alarms by the crypto-detector for anomalies not inserted by MASC are not considered in the evaluation, and all mutants inserted by MASC are identified as either killed or unkilld. Our semi-automated implementation of this methodology,

which adapts to the disparate report formats of the studied crypto-detectors, is detailed in S&P’22 [19].

Step 3 – Identifying *flaws* (RQ₁): We analyzed over 400 randomly chosen, undetected mutants to discover *flaws*, wherein a flaw is defined as a misuse case that a particular crypto-detector *claims* to detect in its documentation, but fails to detect in practice. We took care to also exempt the exceptions/limitations explicitly stated in the crypto-detector’s documentation, ensuring that all of our identified flaws are *novel*. On a similar note, while a crypto-detector may seem flawed because it does not detect a newer, more recently identified misuse pattern, we confirm that all the flaws we report are due to misuse cases that are older than the tools in which we find them. This can be attributed to two features of our evaluation: our choice of the most frequently discussed misuse cases in a taxonomy going back 20 years, and, our choice of tools that were recently built or maintained. Finally, to confirm the flaw without the intricacies of the mutated app, we created a corresponding minimal app that contained only the undetected misuse instance (*i.e.*, the flaw), and re-analyzed it with the detector.

Step 4 – Characterizing flaws (RQ₂): We characterized the flaws by grouping them by their most likely cause, into *flaw classes*. We also tested each of the nine tools with the minimal examples for all the flaws, allowing us to further understand each flaw given its presence/absence in certain detectors, and their documented capabilities/limitations. We reported the flaws to the respective tool maintainers, and contributed 3 patches to CryptoGuard that were all integrated [15].

Step 5 – Understanding the practical *impact* of flaws (RQ₃): To gauge the impact of the flaws, we studied publicly available applications *after* investigating RQ₁ and RQ₂ during the previous study. We first tried to determine if the misuse instances similar to the ones that led to the flaws were also found in real-world apps (*i.e.*, public GitHub repositories) using GitHub Code Search [128], followed by manual confirmation. Additionally, we manually searched Stack Overflow [274] and Cryptography Stack Exchange [271] for

keywords such as “unsafe hostnameverifier” and “unsafe x509trustmanager”. Finally, we narrowed our search space to identify the impact on apps *that were in fact analyzed with a crypto-detector*. As only LGTM showcases the repos that it scans, we manually explored the top 11 Java repositories from this set (prioritized by the number of contributors), and discovered several misuse instances that LGTM tool may have failed to detect due to the flaws discovered by MASC.

Step 6 – Attributing flaws to mutation vs. base instantiation: To determine whether a flaw detected by MASC can be attributed to mutation, versus the crypto-detector’s inability to handle even a base case (*i.e.*, the most literal instantiations of the misuse case from the taxonomy), we additionally evaluated each crypto-detector with base instantiations for each misuse that led to a flaw exhibited by it.

4.7 Results and Findings

Table 4.1: Descriptions of Flaws discovered by Analyzing crypto-detectors

ID	Flaw Name (Operator)	Description of Flaws
Flaw Class 1 (FC1): String Case Mishandling +		
F1	smallCaseParameter (OP ₁)	Not detecting an insecure algorithm provided in lower case; <i>e.g.</i> , <code>Cipher.getInstance("des");</code>
Flaw Class 2 (FC2): Incorrect Value Resolution +		
F2	value in variable (OP ₂)	Not resolving values passed through variables. <i>e.g.</i> , <code>String value = "DES"; Cipher.getInstance(value);</code>

Continued on next page.

ID	Flaw Name (Operator)	Description of Flaws
F3*	secure parameter replaced by insecure (OP ₄)	Not resolving parameter replacement; <i>e.g.</i> , <code>MessageDigest.getInstance("SHA-256".replace("SHA-256", "MD5"));</code>
F4*	insecure parameter replaced by insecure (OP ₄)	Not resolving an <i>insecure</i> parameter's replacement with another <i>insecure</i> parameter <i>e.g.</i> , <code>Cipher.getInstance("AES".replace("A", "D"));</code> (<i>i.e.</i> , where "AES" by itself is insecure as it defaults to using ECB).
F5*	string case transform (OP ₃)	Not resolving the case after transformation for analysis; <i>e.g.</i> , <code>Cipher.getInstance("des".toUpperCase(Locale.English));</code>
F6*	noise replace (OP ₄)	Not resolving noisy versions of insecure parameters, when noise is removed through a transformation; <i>e.g.</i> , <code>Cipher.getInstance("DE\$\$".replace("\$", ""));</code>
F7	parameter from method chaining (OP ₅)	Not resolving insecure parameters that are passed through method chaining, <i>i.e.</i> , from a class that contains both secure and insecure values; <i>e.g.</i> , <code>Cipher.getInstance(obj.A().B().getValue());</code> where <code>obj.A().getValue()</code> returns the secure value, but <code>obj.A().B().getValue()</code> , and <code>obj.B().getValue()</code> return the insecure value.

Continued on next page.

ID	Flaw Name (Operator)	Description of Flaws
F8*	deterministic byte from characters (OP ₆)	Not detecting <i>constant IVs</i> , if created using complex loops, casting, and string transformations; <i>e.g.</i> , a new <code>IvParameterSpec(v.getBytes(),0,8)</code> , which uses a <code>String</code> <code>v=""</code> ; <code>for(int i=65; i<75; i++){ v+=(char)i;}</code>
F9	predictable byte from system API (OP ₆)	Not detecting <i>predictable IVs</i> that are created using a predictable source (<i>e.g.</i> , system time), converted to bytes; <i>e.g.</i> , <code>new IvParameterSpec(val.getBytes(),0,8)</code> ; , such that <code>val = new Date(System.currentTimeMillis()).toString();</code>
Flaw Class 3 (FC3): Incorrect Resolution of Complex Inheritance and Anonymous Objects		
F10	X509ExtendedTrustManager (OP ₁₂)	Not detecting <i>vulnerable SSL verification</i> in <i>anonymous inner class objects</i> created from the <code>X509ExtendedTrustManager</code> class from JCA; <i>e.g.</i> , see Listing A.8 in Appendix).
F11	X509TrustManager SubType (OP ₁₂)	Not detecting <i>vulnerable SSL verification</i> in anonymous inner class objects <i>created from an empty abstract class</i> which implements the <code>X509TrustManager</code> interface; <i>e.g.</i> , see Listing A.11).

Continued on next page.

ID	Flaw Name (Operator)	Description of Flaws
F12	Interface of Hostname Verifier (OP ₁₂)	Not detecting <i>vulnerable hostname verification</i> in an anonymous inner class object that is created from an <i>interface that extends</i> the <code>HostnameVerifier</code> interface from JCA; <i>e.g.</i> , see Listing A.12 in Appendix.
F13	Abstracted Hostname Verifier (OP ₁₂)	Not detecting <i>vulnerable hostname verification</i> in an anonymous inner class object that is created from an <i>empty abstract class</i> that <i>implements</i> the <code>HostnameVerifier</code> interface from JCA; <i>e.g.</i> , see Listing A.10 in Appendix.
Flaw Class 4 (FC4): Insufficient Analysis of Generic Conditions in Extensible Crypto-APIs		
F14	X509TrustManager Generic Conditions (OP ₇ , OP ₉ , OP ₁₂)	Insecure validation of a overridden <code>checkServerTrusted</code> method created within an anonymous inner class (constructed similarly as in F13), due to the failure to detect <i>security exceptions thrown under impossible conditions</i> ; <i>e.g.</i> , <code>if(!(true arg0 == null arg1 == null)) throw new CertificateException();</code>
F15	Interface Hostname Verifier Generic Conditions (OP ₈ , OP ₁₂)	Insecure analysis of vulnerable hostname verification, <i>i.e.</i> , the <code>verify()</code> method within an anonymous inner class (constructed similarly as in F14), due to the failure to detect <i>an always-true condition block that returns true</i> ; <i>e.g.</i> , <code>if(true session == null) return true; return false;</code>

Continued on next page.

ID	Flaw Name (Operator)	Description of Flaws
F16	Abstract Hostname Verifier Generic Conditions (OP ₈ , OP ₁₂)	Insecure analysis of vulnerable hostname verification, <i>i.e.</i> , the <code>verify()</code> method within an anonymous inner class (constructed similarly as in F15), due to the failure to detect <i>an always-true condition block that returns true</i> ; <i>e.g.</i> , <code>if(true session == null)</code> <code>return true; return false;</code>

Continued on next page.

ID	Flaw Name (Operator)	Description of Flaws
Flaw Class 5 (FC5): Insufficient Analysis of Context-specific, Conditions in Extensible Crypto-APIs		
F17	X509TrustManager Specific Conditions (OP ₇ , OP ₁₂)	Insecure validation of a overridden <code>checkServerTrusted</code> method created within an anonymous inner class created from the <code>X509TrustManager</code> , due to the failure to detect <i>security exceptions thrown under impossible but <u>context-specific</u> conditions, i.e., conditions that seem to be relevant due to specific variable use, but are actually not</i> ; e.g., <code>if (!(null != s s.equalsIgnoreCase("RSA")) certs.length >= 314)) throw new CertificateException("RSA");</code>
F18	Interface Hostname Verifier Specific Conditions (OP ₈ , OP ₁₂)	Insecure analysis of vulnerable hostname verification, i.e., the <code>verify()</code> method within an anonymous inner class (constructed similarly as in F14), due to the failure to detect <i>a <u>context-specific</u> always-true condition block that returns true</i> ; e.g., <code>if(true session.getCipherSuite().length()>=0) return true; return false;</code>

Continued on next page.

ID	Flaw Name (Operator)	Description of Flaws
F19	Abstract Hostname Verifier Specific Conditions (OP ₈ , OP ₁₂)	Insecure analysis of vulnerable hostname verification, <i>i.e.</i> , the <code>verify()</code> method within an anonymous inner class (constructed similarly as in F15), due to the failure to detect a <u>context-specific</u> <i>always-true</i> condition block that returns <code>true</code> ; <i>e.g.</i> , <code>if(true session.getCipherSuite().length()>=0) return true;</code> <code>return false;</code>

+ flaws were observed for multiple API misuse cases, * Certain seemingly-unrealistic flaws may be seen in or outside a crypto-detector’s “scope”, depending on the perspective; see Section 4.9 for a broader treatment of this caveat.

Table 4.2: Descriptions of Flaws discovered by Analyzing crypto-detectors in Current Iteration

ID	Flaw Name (Operator)	Description of Flaws
Flaw Class 2 (FC2): Incorrect Value Resolution +		
F20	Static Bytes in Keystore (OP ₁₉)	Not detecting bytes that are created using a static source converted to bytes and passed into IV; <i>e.g.</i> , <code>new IvParameterSpec(val.getBytes(), 0, 8);</code> , such that <code>byte[] val = "12345678".getBytes();</code>

Continued on next page.

ID	Flaw Name (Operator)	Description of Flaws
F21	Char Array to String (OP ₁₇)	Being unable to convert an array of Char into a String object and parsing the value; <i>e.g.</i> , <code>javax.crypto.Cipher.getInstance(String.valueOf(cryptoVariable));</code> , such that <code>char[] cryptoVariable = "DES".toCharArray();</code>
F22	Unsafe Value from Substring (OP ₁₈)	Not detecting a Substring misuse being parsed from a longer string; <i>e.g.</i> , <code>javax.crypto.Cipher.getInstance("secureParamAES".substring(11));</code> ,
F23	Object Sensitivity (OP ₁₆)	Not being able to differentiate between two instances of the same object type one containing a misuse and one containing a safe value; <i>e.g.</i> , <code>String securecipher = new CipherPack().safe().getpropertyName();</code> <code>String unsecurecipher = new CipherPack().unsafe().getpropertyName();</code> <code>securecipher = unsecurecipher;</code> <code>javax.crypto.Cipher cryptoVariable = javax.crypto.Cipher.getInstance(securecipher);</code>
F24	parameter Built From Method Calls (OP ₁₅)	Unable to detect the construction of a misuse String based on calls to methods that construct the misuse; <i>e.g.</i> , Listing A.19 in Appendix
F25	parameter From Nested Conditionals (OP ₁₄)	Being unable to keep track of a variable as it is passed through a nested number of conditional statements; <i>e.g.</i> , see Listing A.18 in Appendix

+ flaws were observed for multiple API misuse cases, * Certain seemingly-unrealistic flaws may be seen in or outside a crypto-detector's "scope", depending on the perspective; see Section 4.9 for a broader treatment of this caveat.

Our manual analysis of undetected mutants revealed 19 flaws in our previous S&P’22 [19] study, and six additional flaws in this study across crypto-detectors we evaluated, which we resolved to both design and implementation-gaps (**RQ₁**). We organize these flaws into five *flaw classes* (**RQ₂**), representing the shared limitations that caused them. Table 4.1 and Table 4.2 provides the complete list of the flaws, categorized along flaw classes, while Table 4.3 and Table 4.4 provides a mapping of the flaws to the crypto-detectors that exhibit them.

As we have shown in Table 4.3 in our previous study, a majority of the total flaws (computed by adding \mathbf{X} and \emptyset instances) identified in crypto-detectors, *i.e.*, 45/76 or 59.21% can be *solely* attributed to our mutation-based approach, whereas only 31/76, *i.e.*, 40.79% could *also* be found using base instantiations of the corresponding misuse cases. Further, all flaws in 6/9 crypto-detectors were only identified using MASC. This demonstrates the advantage of using mutation, over simply evaluating with base instantiations of misuses from the taxonomy. GCS, LGTM, and QARK fail to detect base cases due to incomplete rule sets (see online Appendix [16] for discussion). Similarly, in our current study, a significant portion of the total flaws (122/196) we identified in crypto-detectors can be attributed to mutation based approach, whereas 74 could be found using base instantiations of the corresponding misuse cases as shown in Table 4.4. Moreover, 6 fixed flaws ($\checkmark\checkmark$ in Table 4.4) in this extended study, *i.e.*, flaws that were present in our previous study, but are no longer present, can be attributed to MASC because of the responsibly disclosed reports submitted in S&P’22 [19].

In our previous study, at the initial stage of our evaluation (*i.e.*, before we analyzed uncaught mutants), we discovered that certain crypto-detectors [243, 176] analyze only a limited portion of the target applications, which greatly affects the reliability of their results. As these gaps were not detected using MASC’s evaluation, we do not count these in our flaws or flaw classes. However, due to their impact on the basic guarantees provided by crypto-detectors, we believe that such gaps must be discussed, addressed, and avoided by future crypto-detectors, which is why we discuss them under a flaw class *zero*.

Flaw Class Zero (FC0) – *Incomplete analysis of target code:* Starting from Android API level 21 (Android 5.0), apps with over 64k reference methods are automatically split to multiple Dalvik Executable (DEX) byte code files in the form of `classes<N>.dex`. Most apps built after 2014 would fall into this category, as the Android Studio IDE automatically uses *multidex* packaging for any app built for Android 5.0 or higher [84]. However, we discovered that CryptoGuard and CogniCrypt do not handle multiple dex files (see Table 4.3), despite being *released 4 and 5 years after Android 21, respectively*. Given that this flaw affected CryptoGuard’s ability to analyze a majority of Android mutants (*i.e.*, detecting only only 871/2515), we developed a patch that fixes this issue, and used the patched version of CryptoGuard for our evaluation. The CogniCrypt maintainers confirmed that they are working on this issue in their upcoming release, which is why we did not create a patch, but used non-multidex minimal examples to confirm that each flaw discovered in CogniCrypt is not due to the multidex issue. Similarly, we discovered that CryptoGuard ignores any class whose package name contains “android.” or ends in “android”, which prevents it from analyzing important apps such as LastPass (`com.lastpass.lpandroid`) and LinkedIn (`com.linkedin.android`), which indicates the severity of even trivial implementation flaws. We submitted a patch to address this flaw and evaluated the patched version [15]. Finally, Codiga was unable to detect any of the crypto-API misuse we analyzed, even though it explicitly claims that it “*supports hundreds of rules for Java, checking that your code is safe and secure*” [70]. When we reached out to Codiga (April 2023), we were informed that Codiga would shut down by May 3rd, 2023 without addressing this issue. As a result, we have marked all flaws accordingly (\emptyset), *i.e.*, claims to handle but does not detect base version of misuse in Table 4.4.

The remainder of this section discusses each flaw class with representative examples as they manifest in crypto-detectors and the *impact* of the flaws in terms of examples found in real software.

FC1: String Case Mishandling (F1): As discussed in the motivating example (and seen in **F1** in Table 4.1), a developer may use `des` or `dEs` (instead of `DES`) in `Cipher.getInstance(<parameter>)` without JCA raising exceptions. As shown in Table 4.3, CryptoGuard did not detect such misuse in our previous study. We submitted a patch to CryptoGuard to address this flaw, which was accepted, and demonstrates that this flaw was recognized as an *implementation gap* by the CryptoGuard developers [15]. Since the latest version of CryptoGuard can identify this misuse, we have marked it as such (✓✓) in Table 4.4. Snyc is able to detect when the misuse is instantiated using the `MessageDigest.getInstance(<parameter>)` API (using “md5”), but not using the `Cipher.getInstance(<parameter>)` API (using “aes”, which defaults to ECB mode). Similarly, DeepSource could detect it for the `Cipher.getInstance(<parameter>)` API, but not `MessageDigest.getInstance(<parameter>)` API, indicating a partial *implementation gap* in both these crypto-detectors.

FC2: Incorrect Value Resolution (F2 – F9, F20 – F25): The flaws in this class occur due to the inability of 13/12 of the crypto-detectors to resolve parameters passed to crypto-APIs. For example, consider **F2**, previously discussed in Listing 4.1 in Section 4, where the string value of an algorithm type (*e.g.*, `DES`) is passed through a variable to `Cipher.getInstance(<parameter>)`. Tool_x was not able to detect this (mis)use instance, hence exhibiting **F2** (Table 4.3), and in fact, demonstrated a consistent inability to resolve parameter values (flaws **F2 – F7**), indicating a *design gap*. On the contrary, as SpotBugs partially detects the type of misuse represented in **F5**, *i.e.*, when it is instantiated using the `Cipher.getInstance(<parameter>)` API, but not using the `MessageDigest.getInstance(<parameter>)` API, which indicates an *implementation gap*, which continues to exist throughout the previous and current study.

Further, LGTM and GCS are partially susceptible to F2 because of an intricate problem in their rule-sets. That is, both tools are capable of tracking values passed from variables, and generally detect mutations similar to the one in F2 (*i.e.*, and also the one

in Listing 4.1, created using **OP₂**). However, one of the mutant instances that we created using **OP₂** used AES in `Cipher.getInstance(<parameter>)`, which may seem correct but is actually a misuse, since specifying AES alone initializes the cipher to use the highly vulnerable ECB mode as the block chaining mode. Unfortunately, both LGTM and GCS use mostly similar CodeQL rule-set [124] which does not consider this nuance, leading both tools to ignore this misuse.

Interestingly, SonarQube, as shown in Table 4.4, is partially susceptible to F2, as it does detect misuse for `Cipher.getInstance(<parameter>)` API, but not for `MessageDigest.getInstance(<parameter>)` because of an entirely different reason. As we found from the discussion with SonarQube during the responsible disclosure, this flaw stems from the tendency of avoiding false-positives. As they elaborated, evaluating a non-constant value may result in increased number of false-positives, and as a result, the analyzer avoids evaluating non-constant identifiers in this particular case. While both weak hashing algorithm and weak encryption are within OWASP’s TOP 10 crypto API issues and are similarly considered within scope by SonarQube, this reveals that developers of the same crypto-detector may treat crypto-APIs differently based on internal prioritization factors, which led to this particular flaw. We address this particular insight in Section 4.9 later on.

Finally, we observe that CogniCrypt detects some of the more complex behaviors represented by flaws in **FC2** (*i.e.*, **F7 – F9**), but does not detect the simpler ones (**F3 – F6**). From our interaction with CogniCrypt’s maintainers, we have discovered that CogniCrypt should be able to detect such transformations by design, as they deviate from CrySL rules. However, in practice, CogniCrypt cannot reason about certain transformations at present (but could be modified to do so in the future), and produces an ambiguous output that neither flags such instances as misuse, nor as warnings for manual inspection, due to an implementation gap. The developers agree that CogniCrypt should clearly flag the API-use that it does not handle in the report, and refer such use to manual inspection.

Impact (FC2): We found misuse similar to the instance in **F2** in Apache Druid, an app

with 10.3K stars and 400 contributors on Github (see Listing A.16 in online Appendix [16] . Further, we found real apps that convert the case of algorithm values before using them in a restrictive crypto API [134] (**F5**, instantiated using **OP**₃), or process values to replace “noise” [126] (**F3**, **F4**, **F6**, instantiated using **OP**₄). We observed that ExoPlayer, a media player from Google with over 16.8K stars on GitHub, used the predictable *Random* API for creating IvParameterSpec objects [127] until 2019, similar in nature to **F9**. Developers also use constants for IVs (**F8**), as seen in UltimateAndroid [275] (2.1K stars), and JeeSuite (570 stars) [273].

We also found instances of these flaws in apps that were analyzed with a crypto-detector, specifically, LGTM. Apache Ignite [121] (360 contributors, 3.5K stars) contains a misuse instance similar to one that led to **F2**, where only the name of the cipher is passed to the `Cipher.getInstance(<parameter>)` API [131] which causes it to default to “ECB” mode. LGTM does not report this as it considers ECB use as insecure for Java (but oddly secure for JavaScript). We found similar instances of ECB misuse in Apache-Hive [130] (250 contributors, 3.4K stars), Azure SDK for Java [122] (328 contributors, 857 stars), which LGTM would not detect. Finally, in Apache Ignite [121], we found a `Cipher.getInstance(<parameter>)` invocation that contained a method call in place of the cipher argument (*i.e.*, a chain of length 1, a basic instance of **F7**) [131].

FC3: Incorrect Resolution of Complex Inheritance and Anonymous Objects

(F10 – F13): The flaws in this class occur due to the inability of crypto-detectors to resolve complex inheritance relationships among classes, generally resulting from applying *flexible* mutation operators (Sec. 4.3.2) to certain misuse cases. For example, consider **F11** in Table 4.1, also illustrated in Listing A.11 in online Appendix [16] . Further, we find that several crypto-detectors, *e.g.*, Xanitizer, SpotBugs and SonarQube are immune to these flaws, which indicates that traversing intricate inheritance relationships is a *design consideration for some crypto-detectors*, and a *design gap* in others such as CryptoGuard and QARK. Moreover, such indirect relationships can not only be expected from evasive

developers (*i.e.*, **T3**) but is also found in real apps investigated by the crypto-detectors, as described below.

Impact (FC3): We found an exact instance of the misuse representing **F10** (generated from **OP**₁₂) in the class `TrustAllSSLConnectionFactory` in Apache JMeter [133] (4.7K stars in GitHub). **F11** is the generic version of **F10**, and fairly common in repositories and libraries (*e.g.*, BountyCastle [45, 46]). **F12** and **F13** were also generated using **OP**₁₂, but with the `HostnameVerifier`-related misuse, and we did not find similar instances in the wild in our limited search.

FC4: Insufficient Analysis of Generic Conditions in Extensible Crypto-APIs (F14 – F16): The flaws in this class represent the inability of certain crypto-detectors to identify fake conditions within overridden methods, *i.e.*, unrealistic conditions, or always true condition blocks (*e.g.*, as Listing A.15 in online Appendix [16] shows for **F14**). Flaws in this class represent the behavior of an evasive developer (**T3**). Several crypto-detectors, *e.g.*, Xanitizer and SpotBugs, can identify such spurious conditions.

FC5: Insufficient Analysis of Context-specific Conditions in Extensible Crypto-APIs (F17 – F19): The flaws in this class represent misuse similar to **FC4**, except that the fake conditions used here are contextualized to the overridden function, *i.e.*, they check *context-specific* attributes (*e.g.*, the length of the certificate chain passed into the method, **F17**). An evasive developer may attempt this to add further realism to fake conditions to evade tools such as Xanitizer that are capable of detecting generic conditions. Indeed, we observe that Xanitizer fails to detect misuse when context-specific conditions are used, for both **F17** and **F18**. Our suspicion is that this weakness is due to an optimization, which exempts conditions from analysis if they seem realistic.

Particularly, we observe that Xanitizer correctly detects the fake condition in **F19**, and that the only difference between **F19** and **F18** is that the instances of misuse they represent occur under slightly different class hierarchies. Hence, our speculation is that this an *accidental benefit*, *i.e.*, the difference could be the result of an *incomplete imple-*

mentation of the unnecessary optimization across different class hierarchies. SpotBugs with FindSecBugs shows a similar trend, potentially because Xanitizer uses SpotBugs for several SSL-related analyses. Finally, we observe that Tool_X is immune to both generic **FC4**) and context-specific fake conditions **FC5**).

Impact (FC4, FC5): In this Stack Overflow post [272], the developer describes several ways in which they tried to get Google Play to accept their faulty TrustManager implementation, one of which is exactly the same as the misuse instance that led to **F17** (generated using **OP₇** and **OP₁₂**), which is a more specific variant of **F14** (generated **OP₇**, **OP₉** and **OP₁₂**), as illustrated in Listing A.9 in online Appendix [16] . We observe similar evasive attempts towards vulnerable hostname verification [269] which are similar in nature to **F15** and **F16**, and could be instantiated using **OP₈** and **OP₁₀**. We also found developers trying to evade detection by applying context-specific conditions in the hostname verifier [270], similar to **F18** and **F19**.

4.8 Limitations

MASC does not attempt to replace formal verification, and hence, does not guarantee that all flaws in a crypto-detector will be found. Instead, it enables systematic evaluation of crypto-detectors, which is an advancement over manually curated benchmarks. Aside from this general design-choice, our approach has the following limitations:

1. Completeness of the Taxonomy: To ensure a taxonomy that is as comprehensive as possible, we meticulously follow best-practices learned from prior work [171, 73], and also ensure labeling by two authors. However, the fact remains that regardless of how systematic our approach is, due to the manual and generalized of the SLR, we may miss certain subtle contexts during the information extraction phase (see specific examples in Online Appendix [16]). Thus, while not necessarily complete, this taxonomy, generated through over *2 person months of manual effort*, is, to the best of our knowledge, the most comprehensive in recent literature.

2. Focus on Generic Mutation Operators: We have currently constructed generic operators based on typical *usage conventions*, *i.e.*, to apply to as many misuse instances from the taxonomy as possible. However, currently, MASC does not incorporate operators that may fall outside of usage-based conventions, *i.e.*, which may be more tightly coupled with specific misuse cases, such as an operator for calling `create` and `clearPassword` in a specific order for `PBEKeySpec`. We plan to incorporate such operators into MASC in the future.

3. Focus on Java and JCA: MASC’s approach is largely informed by JCA and Java. Additional operators and adjustments will be required to adapt MASC to JCA-equivalent frameworks in other languages, specially when adapting our usage-based mutation operators to non-JCA conventions.

4. Evolution of APIs: Future, tangential changes in how JCA operates might require changing the implementation of MASC’s mutation operators. Furthermore, incremental effort will be required to incorporate new misuse cases that are discovered with the evolution of crypto APIs. We have designed MASC to be as flexible as possible by means of reflection and automated code generation for mutation operators, which should make adapting to such changes easier.

5. Relative Effectiveness of Individual Operators: This work demonstrates the key claims of MASC, and its overall effectiveness at finding flaws, but does not evaluate/claim the *relative usefulness of each operator* individually. A comprehensive investigation of relative usefulness would require the mutation of all/most misuse cases from the taxonomy, with every possible operator and scope, a broader set of apps to mutate, and a complete set of crypto-detectors, which is outside the scope of MASC, but a direction for future work.

4.9 Discussion and Summary

Designing crypto-detectors is in no way a simple task; tool designers have to balance several orthogonal requirements such as detecting as many vulnerabilities as possible without introducing false positives, while also scaling to large codebases. Yet, the fact remains that there is significant room for improvement in how crypto-detectors are built and evaluated, as evidenced by the flaws discovered by MASC.

To move forward, we need to understand the divergent perspectives regarding the design of crypto-detectors, and reach a consensus (or at least an agreeable minima) in terms of what is expected from crypto-detectors and how we will design and evaluate them to satisfy the expectations. We seek to begin this discourse within the security community by integrating several views on the design decisions behind crypto-detectors, informed by our results and conversations with tool designers (quoted with consent) during the vulnerability reporting process.

4.9.1 Security-centric Evaluation vs. Technique-centric Design

Determining what misuse is within or outside the scope for a crypto-detector is a complex question that yields several different viewpoints. We argue for *security-centric*, *i.e.*, even if some misuse instances may seem unlikely or evasive, crypto-detectors that target security-focused use cases (*e.g.*, compliance, auditing) should attempt to account for them. In other words, crypto-detectors should be able to detect any variation of vulnerability as long as those are statically analyzable. Furthermore, crypto-detectors should also explicitly state or document their limitations related to analysis techniques, so that the software developers are aware of such limitations, and handle such limitations accordingly, *e.g.*, by adopting additional detectors based on different analysis techniques, and/or by supplementing manual techniques. However, we observe that tool designers typically adhere to a *technique-centric* perspective, *i.e.*, the design of crypto-detectors is not influenced by a threat model, but mainly by what static analysis can and cannot accomplish (while im-

plicitly assuming a benign developer). This quote from the maintainers of CryptoGuard highlights this view, wherein they state that the “lines” between what is within/outside scope *“seen so far were technically motivated – not use-case motivated..should we use alias analysis?...”*. This gap in perspective does not mean that crypto-detectors may not detect any of the mutants generated by MASC using operators based on the **T3** threat model; rather, it only means that detection (or lack thereof) may not be *caused* by a security-centric design.

4.9.2 Defining “Scope” for the Technique-centric Design

We observe that even within crypto-detectors that take a *technique-centric* approach, there is little agreement on the appropriate scope of detection. For instance, Xanitizer focuses on catching every possible misuse instance, regardless of any external factors such as whether that kind of misuse is observed in the wild, or a threat model, as the designers believe that *“the distinction should not be between ‘common’ and ‘uncommon’, but instead between ‘can be (easily) computed statically’ and ‘can not be computed’.”* This makes it possible for Xanitizer to detect unknown or rare problems, but may also result in it not detecting a commonly observed misuse that is hard to compute statically, although we did not observe such cases. For SonarQube, the “cost” of analyzing a pattern versus whether it is easy to detect through manual analysis is an additional consideration, *“... easy to identify for simple human readers. For our analyzer ... this kind of method is expensive ... issue is more of a misconfiguration on the developer’s side, trying to cover this type of case seems a bit out of scope.”*

In contrast, CryptoGuard, CogniCrypt, and GCS/LGTM (same developers) would *consider seeminglyunlikely/evasive flaws within scope* (e.g., $\mathcal{F}_3 - \mathcal{F}_6, \mathcal{F}_8$), because they were found in the wild (unlike Xanitizer, for which this is not a consideration). This view aligns with our perspective, that regardless of how it was generated, if a misuse instance (representing a flaw) is discovered in real apps (which is *true for all flaws except \mathcal{F}_{12} and \mathcal{F}_{13}* , it should be within the detection scope. However, GCS/LGTM maintainers extend

this definition with the condition that the observations in the wild be frequent, to motivate change. Note that this scope defining factor also extends to the additional, extended security test-suite available in Github Code Security, while the default test-suite focuses on precision, *i.e.*, as the number of false-positives is often a concern for developers of crypto-detectors, likely inherited from the program analysis community studies [167, 156, 67]. These divergent perspectives motivate the need to clearly define the expectations from crypto-detectors by the designers, *i.e.*, crypto-detectors need to clearly communicate if their use cases are for hostile/adversarial context (*e.g.*, security audit, certification, or compliance), or for developer-friendly, *helping-with-coding* context. Furthermore, to avoid creating a false sense of security, it is important to document known limitations, such as design considerations, and share them with (potential) users, so that the users can make informed decisions about their capabilities.

4.9.3 Utility of Seemingly-Uncommon or Evasive Tests

As Bessey et al. state from practical deployment experience in 2010 [47], “*No bug is too foolish to check for*”, and that “*Given enough code, developers will write almost anything you can think of...*”. The results from our evaluation and the impact study corroborate this sentiment, *i.e.*, $\mathcal{F}_3 - \mathcal{F}_6$ and \mathcal{F}_8 were all obtained using operators (\mathbf{OP}_3 , \mathbf{OP}_4 , and \mathbf{OP}_6) modeled to emulate threats **T1** and **T2**, *i.e.*, representing *benign* behavior (however unlikely); and indeed, these flaws were later found in supposedly benign applications. This suggests that the experience of Bessey et al. is valid a decade later, making it important to evaluate crypto-detectors with “more-than-trivial” cases to not only test their detection prowess, but to also account for real problems that may exist in the wild.

4.9.4 The Need to Strengthen Crypto-Detectors

We argue that it is not only justified for tools to detect uncommon cases (*e.g.*, given that even benign developers write seemingly-unlikely code), but also critical for their sustained relevance. As designers of Coverity found [47], false negatives matter from a commercial

perspective, because *“Potential customers intentionally introduced bugs into the system, asking ‘Why didn’t you find it?’”*.

Perhaps more significantly, the importance of automated crypto-detectors with the ability to guarantee *assurance* is rising with the advent of new *compliance* legislation such as the IoT Cybersecurity Improvement Act of 2020 [159], which seeks to rein in vulnerabilities in billions of IoT systems that include vulnerable server-side/mobile components. Vulnerabilities found *after* a compliance certification may result in penalties for the developers, and financial consequences for the compliance checkers/labs and crypto-detectors used. Complementing static detection with manual or dynamic analysis may be infeasible at this scale, as tool designers noted: *e.g.*, *“...review an entire codebase at once, manual review can be difficult.”* (LGTM) and *“Existing dynamic analysis tools will be able to detect them only if the code is triggered (which can be notoriously difficult)”* (CryptoGuard). Thus, static crypto-detectors will need to become more robust, and capable of detecting hard-to-detect misuse instances.

4.9.5 Towards Crypto-Detectors Strengthened by a Security-Centric Evaluation

Fortunately, we observe that there is support among tool designers for moving towards stronger security guarantees. For instance, CogniCrypt designers see a future research direction in expressing evasive scenarios in the CrySL language *i.e.*, *“...what would be a nice future direction is to tweak the SAST with such optimizations/ more analysis but still allow the CrySL developer to decide if he wants to switch these ‘evasive user’ checks...”*, but indicate the caveat that developers may not use such additional configuration options [167]. However, we believe that such options will benefit *independent evaluators*, and developers who are unsure of the quality of their own supply chain, to perform a hostile review at their disposal. Similarly, the CryptoGuard designers state that *“...this insight of evasive developers is very interesting and timely, which immediately opens up new directions.”*

This potential paradigm-shift towards a *security-focused* design of crypto-detectors is

timely, and MASC’s rigorous evaluation with expressive test cases can play an effective role in it, by enabling tool designers to proactively address gaps in detection during the design phase itself, rather than reactively (*i.e.*, after a vulnerability is discovered in the wild). More importantly, further large-scale evaluations using MASC, and the flaws discovered therein, will enable the community to continually examine the design choices made by crypto-detectors and reach consensus on what assurances we can feasibly expect. We envision that such development aided by MASC will lead to a mature ecosystem of crypto-detectors with a well-defined and strong security posture, and which can hold their own in adversarial situations (*e.g.*, compliance assessments) within certain *known* bounds, which will eventually lead to long-lasting security benefits for end-user software.

Table 4.3: Flaws observed in crypto-detectors in previous iteration

Class	ID	CG	CC	SB	XT	TX	QA	SL	GCS	LGTM
FC1	F1	✗	✓	✓	✓	✓	∅	✓	✓	✓
	F2	✓	✓	✓	✓	✗	∅	✓	●	●
	F3*	✗	✗	✗	✓	✗	∅	✗	✓	✓
	F4*	✗	✗	✗	✓	✗	∅	✗	✗	✗
FC2	F5*	✗	✗	●	✓	✗	∅	●	✓	✓
	F6*	✗	✗	✗	✓	✗	∅	✗	✗	✗
	F7	✗	✓	✗	✓	✗	∅	✗	✓	✓
	F8	✗	✓	✓	✓	–	∅	✓	∅	✗
	F9	✗	✓	✓	✓	–	∅	✓	∅	✗
FC3	F10	✗	–	✓	✓	●	●	✓	∅	∅
	F11	✗	–	✓	✓	●	●	✓	∅	∅
	F12	✗	–	✓	✓	✓	–	✓	∅	∅
	F13	✗	–	✓	✓	✓	–	✓	∅	∅
FC4	F14	✗	–	✓	✓	✓	✗	✓	∅	∅
	F15	✗	–	✓	✓	✓	–	✓	∅	∅
	F16	✗	–	✓	✓	✓	–	✓	∅	∅
FC5	F17	✗	–	✗	✗	✓	✗	✗	∅	∅
	F18	✗	–	✓	✗	✓	–	✓	∅	∅
	F19	✗	–	✓	✓	✓	–	✓	∅	∅

✗ = Flaw Present, ✓ = Flaw Absent, ● = Flaw partially present, – = detector does not claim to handle the misuse associated with the flaw, ∅ = detector claims to handle but did not detect base version of misuse; CG = CryptoGuard, CC = CogniCrypt, SB = Spot-Bugs, XT = Xanitizer, TX = Tool_x, QA = QARK, SL = ShiftLeft, GCS = Github Code Security.

*Certain seemingly-unrealistic flaws may be seen in/outside a crypto-detector’s “scope”, depending on the perspective; see Section 4.9 for a broader treatment of this caveat.

Table 4.4: Flaws observed in crypto-detectors in current iteration

Class	ID	New versions of crypto-detectors							Newly introduced crypto-detectors				
		nCG	nCC	nSB	nQA	nSL	nGCS	nLGTM	AC	SQ	SY	CD	DS
FC1	F1	✓✓	✓	✓	∅	✓	✓	✓	✓	✓	●	∅	●
	F2	✓	✓	✓	∅	✓	●	●	✓	●	✓	∅	✓
	F3*	✓✓	✗	✗	∅	✗	✓	✓	✗	✗	✗	∅	✗
	F4*	✓✓	✗	✗	∅	✗	✓✓	✗	✓	✗	✗	∅	✗
FC2	F5*	✓✓	✗	●	∅	●	✓	✓	✗	✗	✗	∅	✗
	F6*	✗	✗	✗	∅	✗	✓✓	✗	✓	✗	✗	∅	✗
	F7	✗	✓	✗	∅	✗	✓	✓	–	✗	✗	∅	✗
	F8	✗	✓	✓	∅	✓	∅	∅	–	✓	✗	∅	∅
	F9	✗	✓	✓	∅	✓	∅	∅	∅	✓	✗	∅	∅
	F20	✓	✓	✓	∅	✓	∅	∅	∅	✓	✓	∅	∅
	F21	✓	✗	✗	∅	✗	✓	✗	✗	✗	✗	∅	✗
	F22	✗	✗	✗	∅	✗	✓	✗	✗	✗	✗	∅	✗
	F23	✗	✗	✗	∅	✗	✗	✗	✗	✗	✗	∅	✗
	F24	✗	✗	✗	∅	✗	✗	✗	✗	✗	✗	∅	✗
	F25	✗	✗	✗	∅	✗	✗	✗	✗	✗	✗	∅	✗
FC3	F10	✗	–	✓	●	✓	✗	✗	∅	✓	✓	∅	✓
	F11	✗	–	✓	●	✓	✗	✗	∅	✓	✓	∅	✓
	F12	✗	–	✓	–	✓	∅	∅	∅	✓	✓	∅	✓
	F13	✗	–	✓	–	✓	∅	∅	∅	✓	✓	∅	✓
FC4	F14	✗	–	✓	✗	✓	✗	✗	∅	✗	✗	∅	✗
	F15	✗	–	✓	–	✓	∅	∅	∅	✗	✗	∅	✗
	F16	✗	–	✓	–	✓	∅	∅	∅	✗	✗	∅	✗
FC5	F17	✗	–	✗	✗	✗	✗	✗	∅	✗	✗	∅	✗
	F18	✗	–	✓	–	✓	∅	∅	∅	✗	✗	∅	✗
	F19	✗	–	✓	∅	✓	∅	∅	∅	✗	✗	∅	✗

✗ = Flaw Present, ✓ = Flaw Absent, ✓✓ = Flaw was present in the previous study, is no longer present after responsible disclosure ● = Flaw partially present, – = detector does not claim to handle the misuse associated with the flaw, ∅ = detector claims to handle but did not detect base version of misuse AC = Amazon CodeGuru Security, SQ = SonarQube, SY = Snyk, CD = Codiga, DS = DeepSource, nCG = CryptoGuard version 04.05.03, nCC = CogniCrypt version 2.8.0, nSB = SpotBugs 4.0.4 with FindSecBugs version 1.12.0, nQA = QARK version 4.0.0, nSL = ShiftLeft version 2.1.1, nGCS = GitHub Code Security CI/CD Apr 2024, nLGTM = GitHub Code Scan v2.12.6 with LGTM Ruleset.

*Certain seemingly-unrealistic flaws may be seen in/outside a crypto-detector’s “scope”, depending on the perspective; see Discussion for a broader treatment of this caveat.

Chapter 5

Identifying the Gaps in the Practice of SASTs

Software security has gained continued international attention in recent years due to the increase of high-profile cyberattacks and exploits across the public sector. For example, incidents such as the SolarWinds Cyberattack prompted the U.S. Government Accountability Office to elicit responses from both private and public sectors in 2021 to increase the effectiveness of security practices [223]. Consequently, corporate and government entities alike are now increasingly emphasizing the security of software and services through a combination of (1) new approaches (*e.g.*, Software Bill of Materials (SBOM) [285]), (2) adoption of security focused certifications of software (*e.g.*, Cyber Shield Act [286], IoT Compliance [161]), and (3) improvement of existing approaches (*e.g.*, identifying and employing recommended types of automated software security testing [147]). As a result, the existing multi-billion dollar industry of automated security analysis tools [157], particularly Static Application Security Testing (SAST), have continued to proliferate to meet the increased security needs of organizations worldwide. Further, such tools are now being incorporated into nearly every stage of the software development and maintenance lifecycle, from requirements engineering to fault localization and fixing (*e.g.*, the GitHub Code Scan initiative [2]).

However, SAST tools have been found to suffer from design and implementation flaws [19, 21] that prevent them from detecting vulnerabilities that they claim to detect, or which can be expected for certain critical use cases they support (*e.g.*, compliance, audits). Particularly, while SAST tools from industry, the open-source community, and academia have been found to support similar use cases, their design goals often differ dramatically [19]. That is, tools may adopt a *technique-centric* approach, wherein what they can detect is tied to the limitations of a set of chosen static analysis techniques, or, a *security-centric* approach, wherein the tool aims to use whichever static analysis techniques necessary to detect vulnerabilities falling under a specific security goal. These different design ethos carry with them various trade-offs that impact the applicability, efficiency, and effectiveness of the security tools. These trade-offs and their implications for cybersecurity in practice are currently poorly understood, at best.

In other words, we are increasingly heading towards a future where software developers will be depending more than ever on security focused program-analysis techniques for security assurance, compliance, and audits. While we know of potential flaws in SASTs (as discussed previously), there exists a **key gap** in prior research: *the research community has only a limited understanding of how software developers perceive SASTs, what they expect from SASTs and believe in (particularly in terms of their ability to detect vulnerabilities), and how these perceptions and beliefs impact the adoption and use of SASTs in practice.* Without addressing this gap through an understanding of the *practitioners' perspective*, we may not be able to develop SASTs that are truly effective in practice, *i.e.*, possess key properties that practitioners desire in order to improve software security, and moreover, will be unable to uncover gaps in what the practitioners (*i.e.*, users of the tools) expect or believe, versus what the tools actually provide, leading to a false sense of security.

Contributions: This chapter describes a qualitative study that investigates the assumptions, expectations, beliefs, and challenges experienced by practitioners who use program-analysis based security-assurance tools, specifically SASTs. Our study is guided by 3 key

research questions (**RQ1 – RQ3**), which we explore via in-depth interviews ($n = 20$) with software developers, project managers, research engineers and practitioners, who together cover a broad range of security, product, and business contexts:

RQ1: *How do practitioners at organizations, with different types of business and security needs, choose and depend on SASTs for ensuring security in their services/products?*

Various factors may influence an organization's process for selecting a SAST tool, ranging from security or business needs (*e.g.*, compliance), brand reputation, or inclusion of safety-critical components. Thus, we are interested in exploring what individual practitioners and their organizations care about in terms of security, and how those needs affect the selection of SASTs, as well as their incorporation into their overall vulnerability detection processes. We also seek to explore *how* SASTs are selected, *i.e.*, the subjective or objective processes involved in choosing a particular SAST.

RQ2: *What do practitioners know and believe about the limitations of SASTs, and what do they expect from them?* While certain limitations, such as false positives, are relatively well-known, potential issues related to design and/or implementation flaws that result in security-specific false negatives are often unknown and unaccounted for in SASTs. We are interested to understand the awareness, expectations, and beliefs of practitioners about such limitations, both known and unknown, of SASTs, particularly in terms of false positives and negatives.

RQ3: *How do practitioners navigate, address, or work around flaws of SASTs?* A SAST that does not detect vulnerabilities may lead to vulnerabilities in otherwise security-assured software. We are interested in learning about practitioners' experiences regarding the impact of the flaws in SASTs (*e.g.*, product-related security incidents). Furthermore, we are interested to know how practitioners balance the possibility of unsound SASTs that may make their product vulnerable, and the decision to release potentially vulnerable software. Moreover, if practitioners do happen to find a flaw in a SAST, we seek to uncover their experiences in reporting the issues to the SASTs. Finally, we seek to investigate the typical pain points that practitioners experience regarding SASTs, in order to understand

the key properties they desire more than anything else.

5.1 Methodology

To understand the potentially diverse perspectives of practitioners related to SASTs, we performed a two-phase study, composed of a survey, followed by in-depth, detailed interviews. The purpose of the survey was to develop an initial understanding of the landscape, and more importantly, to guide the design of the interview protocol. Therefore, this section (and the rest of the chapter) focuses on the interviews and their qualitative analysis. Moreover, the artifacts associated with the survey and the interview, including the informed consent forms, survey questionnaires, and the interview guide, are available in our online appendix [228].

We now provide a brief summary of the survey protocol and results, which are further detailed in Appendix A.5 and Appendix A.6 respectively. The interview protocol is described in Section 5.1.2.

5.1.1 Summary of the Survey Protocol and Results

To understand how practitioners perceive and use SASTs and security tools more broadly, and how security is prioritized by individuals and organizations, we prepared an online survey questionnaire (the questionnaire is in the online appendix [228]) consisting of Likert-based questions, with optional open-ended responses to clarify their selected choice(s). The survey protocol was approved by the Institutional Review Boards (IRBs) at the authors' universities.

We used two recruitment channels. First, we relied on *snowball sampling* [135] from our professional networks primarily by sending invitation emails and requesting forwards to colleagues. Second, we *emailed OSS developers* who had interacted with SASTs via CI/CD actions, *e.g.*, GitHub Workflows, in open-source repositories that (a) had at least one star or watcher, (b) were not a fork, and (c) used one of the top ten pro-

programming languages reported in GitHub Octoverse [5]. We developed scripts that used GitHub Search APIs and crawled public repositories in Coverity Scan [1] to find qualifying repositories. Next, we extracted email addresses from commits specific to CI/CD files (*e.g.*, `.github/workflows/*.yaml`) that contained SAST names. Finally, we excluded any Github-assigned private emails and those that indicated no-reply. In the end, we contacted 1,918 OSS developers exactly once via email. We discuss the ethical considerations in recruiting OSS developers in Appendix A.5.1.

Survey Results Overview: We received 89 responses from the survey, of which 39 (18/39 responses from OSS developers) were complete and valid. Of these, 35/39 worked for organizations, whereas 2 were freelancers, and 2 chose not to indicate organizational status. We made several observations that guided the design of our interview protocol based on these valid responses. *First*, over 83% (29/35) participants (from organizations) stated that security was of “extreme” importance to them, with an additional 17% (6/35) stating it as “very” important. In contrast, 63% (22/35) participants identified security as of “extreme” importance from their organization’s perspective, with several (8/35 or 23%) considering it “very” important. This tells us that security may be prioritized differently by the organization and the individual, and more importantly, that individuals may be willing to talk about these differences, which is critical for our interviews. *Second*, participants mostly relied on a combination of automated and manual security analyses, with some exceptions. *Finally*, almost all practitioners expressed that even in the case of flaws of SASTs, their applications would be moderately impacted at most, as they rely on multiple tools and/or manual reviews. These observations guided our protocol design for the interview phase, which we describe next.

5.1.2 Interview Protocol

We drafted an interview protocol consisting of a semi-structured interview-guide [11, 149] structured by “laddering” questions [74], a questioning strategy that is used to understand

the relations between concepts in a domain and to explore the concepts in-depth. The interview guide is designed to help understand the processes used to choose SASTs, how practitioners depend on SASTs for security assurance, their expectations about limitations of SASTs, and how they work around such limitations. An abridged version of the interview guide is in Table A.7 in the Appendix.

5.1.2.1 Interview Recruitment

We recruited 20 interview participants through multiple recruitment channels, aiming for diversity in project, cultural background, experience and industry contexts. Particularly, we recruited (i) 10 participants from the survey and (ii) 10 separately through our professional network. When recruiting from the survey-pool, we only invited participants who submitted reasonably valid responses and expressed interest in interview participation. To recruit from our professional networks, we relied on snowball sampling [135], i.e., emailed invitations to software engineers within our network, with details of the study as per protocol, and requested them to forward the invitation to colleagues experienced with SASTs.

Overall, we recruited 20 interview participants with diverse cultural background (*e.g.*, participants were from Asia, Europe, United Kingdom and North America, working in either local or international projects), industry contexts (*e.g.*, safety-critical, business-critical, research & development, open-source *etc.*), experience ranging from entry level engineers to project managers, and security-context (*e.g.*, working towards compliance). The anonymized details of participants are shown in Table 5.1.

5.1.2.2 Interview Protocol and Ethics

Similar to the survey, the final version of our interview guide and protocol was approved by our Institutional Review Boards (IRBs). Our consent form emphasized that no personally identifiable information would be collected, and responses will be anonymized even in the case of willfully shared private information, such as a participant’s (or colleagues) name,

associated previous or current organization, and product/client name(s). Each participant would be interviewed for approximately an hour, and would receive a \$50.00 gift card or voucher in local currency.

Furthermore, as our interview-guide contained *sensitive* questions *e.g.*, security enhancement vs meeting deadlines, or site-incidents due to flawed SAST, we followed the Menlo Report Guidelines [170, 88] to refine our protocol and interview guide to avoid any potential harm, and reminded participants that they could withdraw/redact at any time, as further detailed in Section 5.1.2.4.

5.1.2.3 Interview Pilot & Refinement

We conducted pilot interviews, followed by an in-depth discussion, with three participants within our professional network to improve the interview guide. Among these, one held a doctoral degree in computer science, with a focus in CyberSecurity, while the other two were pursuing a Ph.D. in Computer Science.

5.1.2.4 Interviewing Procedure

We conducted the interviews using either the lead-interviewer or lead-and-backup approaches, while following the semi-structured interview-guide. The lead-and-backup approach ensured that each interviewer experienced conducting the interview with the guide. While relevant questions from the guide were raised, it also allowed the lead interviewer to focus on listening and asking laddering/follow-up questions to the interviewee. All the interviews were conducted virtually via Zoom. We emailed the Informed Consent Form (and survey response when applicable) with IRB protocol references a day before the interview. After the participant joined us in the online interview session, we reminded the participant *before starting the interview* that (a) the interview will be recorded, (b) we will anonymize any sensitive information while transcribing, (c) recorded audio will be destroyed after transcribing, (d) they can redact anything they said at any point of the interview and/or can email us about it, and (e) they have the option to stop the interview

at any point. The median, effective duration of the interviews, *i.e.*, excluding the intro, briefing and verbal consent, was 1 hour 52 seconds.

5.1.3 Structure of the Interview Guide

We designed the semi-structured interview guide in a way that facilitates understanding how practitioners at organizations choose and depend on SASTs with the context of their business and security needs (**RQ1**), what practitioners with different needs and priorities know and assume about limitations, such as soundness issues, in SASTs (**RQ2**), how they address the limitations of SASTs (**RQ3**). As shown in Figure 5.1, the interview guide consists of questions arranged in six segments, ordered by increasing-depth as applicable.

5.1.3.1 Participants, Projects, and Organizations

At the start of the interview, we asked several “*warm-up questions*” to understand the products/services the participants contribute to, their organization, their experience with developer tools for software security and how they define security in terms of their work. Through these questions we developed the initial context to ask more in-depth follow-up questions. More specifically, we asked the participants about their domain of work, their target clients, how they learned about software security that’s relevant to their work, the security aspects that are important in their work, as well as the relevant threat models they consider.

5.1.3.2 Organization and Security

Next, to gain a deeper understanding of the organizational context of security in their practice, we asked the participants about how they address security in their organizations product development life-cycle. For example, we asked whether there are conflicts between feature deadlines and the security of a given feature, what the conflict resolution process is in general, and whether they have experienced any external factors that constrained

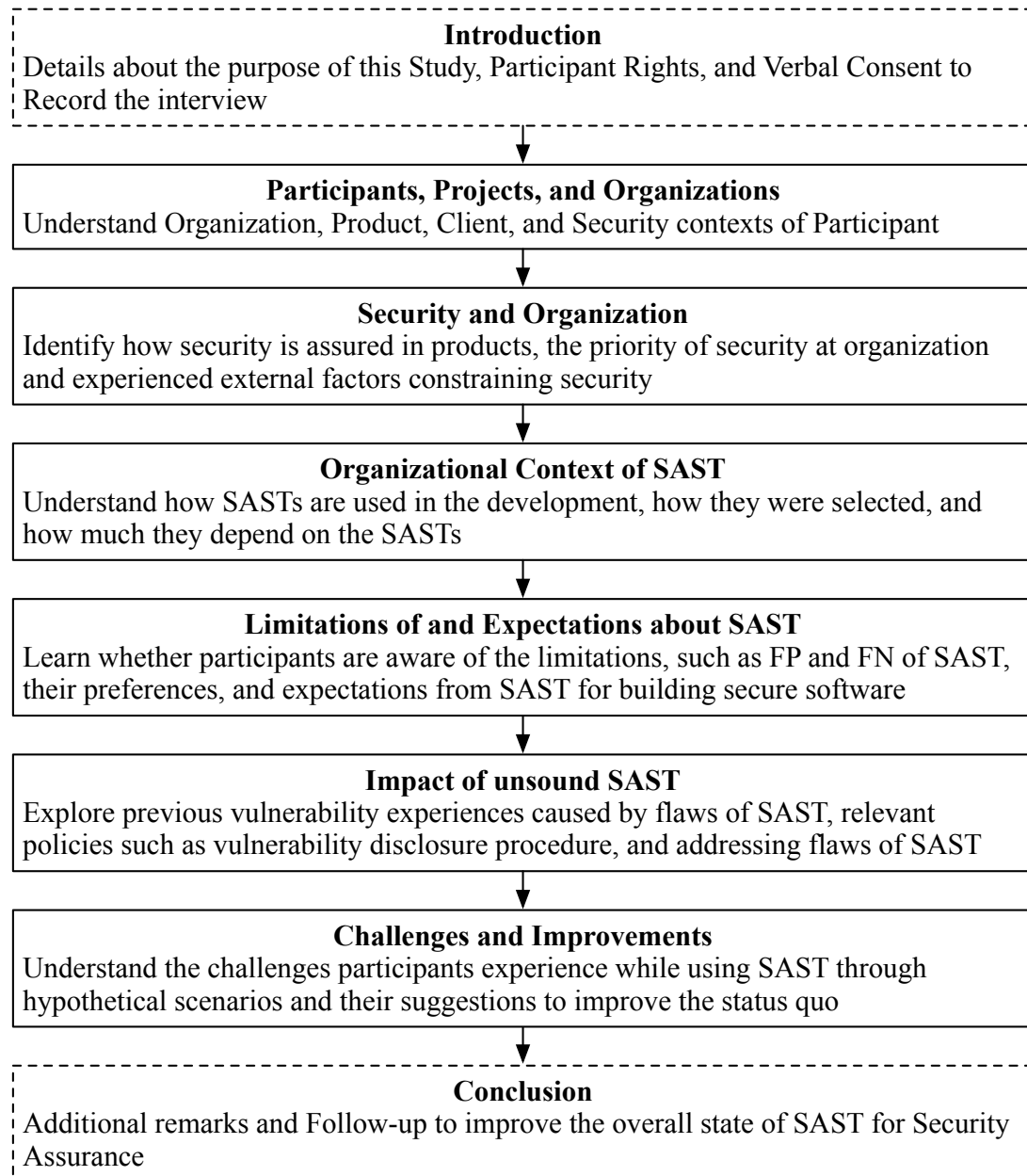


Figure 5.1: Overview of Semi-structured Interview Guide. We used Laddering technique to delve deeper in each topic, while the semi-structured approach helped us to freely deviate as necessary based on participant response.

security. By raising such questions, we developed a better understanding of the trade-offs an organization makes when it comes to security.

5.1.3.3 Organizational Context of SAST

We then asked questions about how one or more SASTs are being used in organizational and team contexts. From the survey, we observed that organizations and their developers may have different priorities and perceptions when it comes to security, which motivated us to distinguish between these two contexts. To elaborate, we asked the participants about their team structure, whether the team(s) address security requirements collaboratively or separately and how, and what happens when such requirements are not met. Moreover, to understand the role of SASTs in the organization and team, we asked questions to understand how they decided to use SASTs in the first place, how they selected SASTs, and to explain why and to what degree they rely on SASTs.

5.1.3.4 Expectations from SAST

We asked questions related to participant's expectations from SASTs and their limitations, using both hypothetical scenarios and also leveraging opinions the participant expressed throughout the interview. For example, we asked about preferences regarding false positives vs. false negatives, explaining the concepts as necessary. Furthermore we asked whether their preference of SASTs is tied to their work, values shared within the community, or something else.

5.1.3.5 Impact of Unsound/Flawed SAST

To understand the impact of a flawed/unsound SAST, we asked the participants about their experiences and organizational processes. For example, we requested that participants share specific experiences related to security vulnerabilities resulting from a SAST that did not work as intended. If that participant did not have such experiences, we asked whether a process exists that helps them address such potential flaws. In addition, we

asked how and why the participants generally attempt to address problems encountered while using SASTs.

5.1.3.6 Challenges and Improvements

Finally, we concluded the interview by raising several “*creative*” questions. For example, if a participant is given unlimited resources to solve one particular problem of the SAST they use, what problem would they prioritize before anything else, and why would they want to solve it. By raising such open-ended questions that *remove limitations* tied to organization and product-context, we aimed to understand what participants want or need in the SASTs they use.

5.1.4 Transcribing, Coding and Analysis

One of the authors systematically transcribed the audio records while anonymizing the text. This required significant amount of time as the median, effective interview duration was one hour, consisting of approximately 9,000 words, with a total word count of over 187,000 words across all interviews. We chose reflexive thematic analysis combined with inductive coding for our analytical approach [56] as it offered us the flexibility of capturing both latent and semantic meaning based on the complex interactions between the participant’s perceptions and contexts, such as assurances offered by automated security analysis techniques, organizational priorities, limitations of security resources, and the nature of products. Furthermore, it considers researcher subjectivity, *i.e.*, experiences and skills of researchers in analysis. We chose a single-coder approach, which is considered “good practice for reflexive TA”, as it helps interpretation, or “*meaning-making*”, from data [56]. While transcribing provides an initial idea about the data and internal patterns, we had to iterate through the steps of thematic analysis (familiarization, coding, identifying potential themes, refining) to finalize the themes.

Table 5.1: Overview of interviewed participants, position(s), product area(s), security priority from the perspectives of participants and project metadata. Fine-level details are binned to ensure the anonymity of the participants.

ID	Time ¹	Chl ²	Role(s) ³	Product Area ⁴	Security Priority		
					Org ⁵	Dev ⁶	Prod ⁷
Interview Participants Recruited from Survey							
P01	01:00:12	OSS	Senior Engineer	Program Analysis for Security	✓	✓	✓
P02	01:04:08	OSS	Developer	OSS - Java Application Server	✓	✓	✓
P03	00:51:04	OSS	Developer	OSS - Internet Anonymity Network	✓	✓	✓
P04	01:00:44	OSS	Embedded Engineer	Automobile Sensors	✓	✓	✓
P05	00:57:46	PN	Developer	Web Applications	✓	✓	✓
P06	00:53:53	PN	Developer	Software Service	✗	✗	✓
P07	01:04:23	PN	Engineering Manager	B2B, SaaS	✓	✓	✗
P08	00:55:03	PN	Full stack Developer	Media, Web and Back-end services	✓	✓	✓
P09	01:05:25	PN	Senior Engineer	Fintech, Business Critical	✓	✓	✓
P10	01:01:40	PN	Developer	Healthcare	✓	✓	✓
Interview Participants Recruited through Snowball Sampling from Professional Network							
P11	01:01:48	SS	Developer	Website Backend of Program Analysis for Security	✓	✓	✓
P12	01:13:48	SS	Developer	Finance of International Online Marketplace	✓	✓	✓
P13	00:51:14	SS	Research Engineer	Research Institute with Industry ties (EU)	✓	✗	✗
P14	01:02:09	SS	Principal Configuration & Dev-Ops Engineer	Law Enforcement	✓	✓	✓
P15	01:00:57	SS	Senior Developer	Service Company	✓	✓	✗
P16	01:03:43	SS	AI Developer, Project Manager	AI Products	✓	✗	✗
P17	00:49:15	SS	Entrepreneur	Enterprise Resource Planning, Education platform	✗	✗	✗
P18	00:50:53	SS	Software Infrastructure Engineer	Fortune 500 Global R&D Center	✓	✗	✓
P19	01:05:13	SS	Senior Software Engineer	Software Solution Provider	✓	✓	✗
P20	00:36:14	SS	Backend Senior Software Engineer	Telematics	✓	✓	✓

¹Effective duration, *i.e.*, timed after introduction, briefing and verbal consent for starting to record the interview, ²Recruitment channel; OSS = Open Source Software developers, PN = Survey participants recruited from Professional Network, SS = Snowball Sampling within Professional Network, ³Self-reported by the participants, multiple roles separated by commas, ⁴Product area binned, none of the interviewees work in small (e.g., 2-person) organizations, ⁵Invests in security in terms of the security team, tools, infrastructure, and/or training for developers (✓), ⁶Participant explicitly expressed that developers in organization are concerned about programmatic security, and/or is directly related with security tool development/setup (✓), ⁷Product is required to be compliant to privacy and/or security standards (*e.g.*, HIPAA, GDPR, PCI DSS, OWASP) or is critical in terms of safety (✓).

5.2 Interview Results

This section describes the results from our interpretation and analysis of 20 semi-structured interviews of practitioners, substantiated by transcribed quotes (omissions highlighted by ... and anonymization in `<angle brackets>`). Also, when quoting participants (except when inline), we also include the product area, to provide further context behind the quote, *e.g.*, P20 `Telematics`.

5.2.1 Participants, Projects, and Organizations

We succeeded in recruiting participants involved with a diverse range of products or services, such as web applications, anonymity networks, research software, safety-critical embedded systems, business-critical financial tech systems, and more, as illustrated in Table 5.1.

All the participants in our study work on multiple projects, with a number of them working in multiple organizations. Further, all the participants work in team(s), although the structure varies. For instance, P01 worked in a team that maintains an Interactive Application Security Testing (IAST) product, but also used SASTs in their work. On the other hand, P03 worked with a collection of people responsible for maintaining a popular open source anonymity protocol (*“It’s a community run project...no corporation in control of any particular aspect of it”*), and is responsible for configuring the SAST tools used by their team.

In summary, and as seen in Table 5.1, the recruited participants possessed valuable, diverse experience by working in different types of projects with varying levels of security needs at their own, unique organizations. This positioned us to further understand how productivity, in terms of feature implementation/completion, and security, in terms of ensuring that features implementations are not vulnerable, are balanced at these diverse organizations.

5.2.2 Organization and Security

Unsurprisingly, all participants agreed that delivering secure software is important. However, we were also interested to learn about the prioritization of security in organizations. Therefore, we queried participants about the potential tension within their organizations related to prioritizing software security at the expense of features and vice versa. Given that introducing and using SASTs in a development workflow requires nontrivial effort from individuals and potential financial investment from an organization, we expected most participants and organizations had a vested interest in prioritizing security. We found that this expectation to generally hold true, with a few exceptions.

Prioritizing Security vs Functionality Deadlines: Most participants indicated a prioritization of security over deadlines, *e.g.*, as P08 states, “*Security gets the highest priority. Always... Even if we are not meeting the deadline, we cannot break this.*” We found that various factors can be responsible for necessitating this prioritization, *e.g.*, the need to be compliant with existing laws and standards:

“We serve the government ... we need to have some certifications that we are complying ... (If) we have a release tomorrow and the security team found a vulnerability today, we have to block that release, and we have to fix it. Then we will release that. ... We cannot compromise that.” — P20 Telematics

It can also be due to safety-critical and/or business-critical nature of the product being built, since a bug can be costly, both in terms of lives and financial measures:

“So our security and safety and usage of the static analysis tools is mostly to prevent bugs, which could be life-threatening, of course, but also they could cost us millions”
— P04_{Automobile Sensors}

For open-source collaborations, the concept of deadlines may not be applicable at all. As P02 described, security is always of priority, and there is “*No such thing as deadlines. It’s ready when it’s ready*”.

Finding 1 (\mathcal{F}_1) – Participants generally said that they err on the side of security, fixing any known vulnerabilities before releasing a feature, regardless of deadlines.

However, some participants expressed that prioritizing security is not always possible, even when security is generally a high priority from the organization’s perspective. This can be due to the management prioritizing bug-fixing for the sake of users, as shared by P07,

“... Our user was facing a lot of issues. So, there was a deadline pressure on us to deliver the product very quickly” — P07 B2B, SAAS

This is true even for a security-testing product, albeit rarely:

“So in most cases, we try to be really strict because it’s a security testing product ... Then it’s kind of a business trade off. ... a new feature, we can usually just delay it ... If it’s an existing feature that we now uncover the vulnerability, you can’t usually switch off the feature because you have customers relying on it... Eventually you get that fixed and then responsibly disclose it. ... I think we had to do one of those in the five years I’ve been with the company.” — P01 Program Analysis for Security

The “overriding” of security to meet deadlines for existing features may incur heavy cost, however. P07 expressed the following after further conversation, *“That had a certain impact. We found .3 to .4 million <currency> of fraudulent activity after that release”*.

Finding 2 (\mathcal{F}_2) – Select situations can lead to the de-prioritization of software security, including maintaining support for *existing* features, or fixing bugs that being experienced by prominent users.

This finding echoes similar observations in prior work, that in some cases security is forgone for functionality bugs or releasing other features [112, 233, 34, 301]. Further, contrary to our initial intuition, P06 shared that an organization may not prioritize security unless it is required by its clients.

“Security is a great concern ... So if the client is strict enough to focus on the security aspects, then we follow it. Other than that, actually our <previous org> do not

care (about security) ... ” — P06_{Software Service}

Unsurprisingly, several participants shared that an organization may not afford to miss functionality deadlines if it is still in startup or growth stage:

“When I worked on a startup environment, it is always expected that we ship the features to production as soon as possible... There is a little room to explore the security options ... ” — P10_{Healthcare}

Finding 3 (\mathcal{F}_3) – Participants expressed that in certain circumstances, organizations may entirely forego security considerations and prioritize releasing features as rapidly as possible, particularly when the client does not care, or when the organization is in its early growth stage.

5.2.3 Organizational Context of SAST

With the understanding of how security is viewed from a participant’s perspective in the context of their organization, we aimed to understand their organizational context of SASTs. Particularly, we asked participants about reasons for using SASTs, the selection process of SASTs, and how they generally use those SASTs in their workflows. Although participants named SASTs, *e.g.*, Coverity, Find-Sec-Bugs, SonarQube, Semmlle/CodeQL, WhiteSource/Mend, CryptoGuard, Fortify, and VeraCode, we anonymize such details to reduce the chance of profiling developers and to avoid creating any impressions specific to any particular SAST.

Selecting SASTs: To start, we asked participants about the events that led to choosing a given SAST and to walk us through the selection process at their organizations. Interestingly, we did not find any pattern in the selection processes that would hold true for a majority of the participants.

Particularly, only P04 shared that they performed a multi-stage evaluation, *i.e.*, they started with a preliminary list of 10 – 15 SASTs, and filtered to four SASTs based on their own product-specific needs. Next, they evaluated those four SASTs using a *custom*

benchmark, and settled on a SAST that was the most usable.

Further, six practitioners shared that they chose SASTs solely based on popularity, developer friendly documentation and/or ease of use.

“We didn’t evaluate that many tools in terms of static analysis tools. We take what is the industry standard across different companies. Like <tool> is pretty popular, so that is our first choice.” — P08_{Media}, Web and Back-end services

P09 additionally mentioned that <SAST_A> was chosen due to regulatory reasons, *“I believe it was either PCI DSS requirement or a regulatory requirement”*, admitting that they do not remember the exact standard. On the other hand, four participants reported using previous experience or familiarity to select a SAST.

“Because I actually inherited some of that. The person who, actually set a lot of it up ... I think that it was what was available and what he was familiar with at the time.” — P03_{OSS} - Internet Anonymity Network

Several developers justified that they prefer freely available SASTs because it helps cut costs, *e.g.*, P07 said *“...As of now, we are looking at a free solution. If we find benefits, then we’ll go for the paid solution ...”*

Corporate influence is an additional factor for selecting a particular SAST, particularly when it comes down to cost, *e.g.*, as P14 said, *“A lot of it comes down from management, ... because they’re the ones that are paying for it”*. In a similar vein, P08 shared:

“We have different teams and different teams have different requirements. In my team, we use <SAST_A>, and it is enforced by the team leader or the team owner to use <SAST_A> as a code analysis tool.” — P08_{Media}, Web and Back-end Services

Finding 4 (\mathcal{F}_4) – Participants generally recall selecting SASTs due to factors such as recommendations/reputation, ease of use/integration, corporate pressure, cost, or compliance requirements. Only *one* participant selected a SAST for their product via exhaustive testing of 10-15 tools using a (custom) benchmark.

Furthermore, we asked participants about whether they considered using benchmarks,

such as the OWASP benchmark while selecting SAST. Most participants said that they were not familiar with any benchmarks, with the rest sharing that benchmarks are not representative of their specific application context, *e.g.*, “*The thing is, OWASP is something that only covers your basics. It doesn’t go beyond*” (P09). Furthermore, P01 shared that while community-based benchmarks such as OWASP are usually neutral, many others are biased.

“Quite a few of these benchmarks are created by tool vendors where their tool finds some specific edge case. No one in the right mind would write an application like this, but their tool finds a specific edge case, so they put it in the benchmark.” —

*P01*_{Program Analysis for Security}

Finding 5 (\mathcal{F}_5) – Participants who are aware of benchmarks generally do not trust them for evaluating/selecting SASTs, viewing benchmarks as either too basic to model real problems, or biased towards specific SASTs, given that vendors often contribute to their construction.

Preference between Manual Techniques and SAST: As expected, participants who use SASTs stated that they found them useful, regardless of the selection process. Several participants shared that they use SASTs because they help focus manual analysis efforts on non-trivial issues by *automatically* finding the trivial issues, *e.g.*, “*...helps find all the stupid stuff for you. Then you can concentrate on the actual logic (P01)*” and makes it easier to analyze a large code base, *e.g.*, “*Is it possible to go through each of the code change by a human being? (P20)*” and “*I think they’re absolutely useful. It kind of reduces the number of mistakes you can make*” (P09). Furthermore, several shared that it is helpful for applying a rigorous quality control to the whole code base without being affected by subjective analysis, *e.g.*,

“Lot of reasons to be paranoid about it. None of us really, totally trust ourselves. And so, we need to have these tools to make the job of finding our own mistakes easier. If only one person is working on a thing, you’re stuck with only that person’s blind

spots. ” — P03_{OSS} - Internet Anonymity Network

Finding 6 (\mathcal{F}_6) – Participants consider SASTs highly useful for both reducing developer effort and helping to cover what subjective manual analysis may miss.

Reasons for not relying on SAST: Finally, we had two participants in our study who do not rely on SASTs. P13 stated that while their product needs to be secure, it is not public-facing, *i.e.*, “*Even if there is a problem in some projects, so one can access those deployed or the application from outside of our internet*”. Interestingly, P02 shared that while they have tried premium SASTs, they did not find them useful in their particular application niche, *i.e.*, web servers, stating that

“*The primary issue with the <generic SAST> tools, every time we’ve looked at these tools, is it’s all false positives and no genuine issues at all, which is somewhat demoralizing if you try to wade through large amounts of these reports.* ” — P02_{OSS} - Java App Server

That is, as P02 further elaborated, since their product is a web server, it is required to handle “vulnerable” requests, such as “HTTP” headers, in code based on existing standards. These code components, however, trigger SASTs built to target web-applications, resulting in high false positives.

Finding 7 (\mathcal{F}_7) – The few participants who do not use SASTs cite the lack of a “*fit*” for their product: *i.e.*, as the product does not need extensive testing (echoing similar observations in prior work [298]), or because generic SASTs flag features (*e.g.*, handling standard-mandated vulnerable HTTP requests) as vulnerabilities.

5.2.4 Expectations from SAST

Developers shared that while they expect SAST tools to detect all vulnerabilities as long as they are within scope, they generally do not expect SASTs to detect all types of security vulnerabilities, *e.g.*, “*If it is in the scope, then it can detect, but my expectation is not like static analysis is the final solution...(P10)*”. When asked whether this assertion was based

on “belief” or “evidence”, P18 explained that it was *“based on belief”*, further explaining that *“We have the user ratings of our tools and there are many stars in the repos. So we think that it is reliable, and many developers use that, so it must be good”*.

Finding 8 (\mathcal{F}_8) – Although expressing that no tool can find everything, participants believe that SASTs (should or do) detect *all vulnerabilities considered within scope* (i.e., which a SAST tool claims to detect).

When

asked to give examples of vulnerabilities that developers do not expect SASTs to detect, runtime (Input/Output), external component, and software goal based issues frequently came up, e.g., *“These tools are pretty agnostic of the goals that we have put forward in the first place. They can only really seem to process errors in code and not errors in software taken as a whole (P03)”*.

Interestingly, when we asked developers if they consider a SAST to be acceptable to use even if it misses some more difficult issues, they generally expressed that they do, e.g., *“I don’t expect that there will ever be a tool that will look at a piece of code as complex as <product> and find all the security issues. ... But any issue fixed is an issue fixed and that’s a good thing”* (P02). When asked to elaborate, participants shared different reasons for finding such SASTs acceptable, such as lack of alternatives, *“If there is no other accessible alternative, then I would go and accept whatever it offers”* (P10) and additional techniques being used to cover for (issues in) SASTs e.g., *“... for our team, the manual review part is actually the biggest deal for us. ... So, for our team, I think that should not be a big issue”* (P07).

Finding 9 (\mathcal{F}_9) – Participants consider SASTs valuable even if they miss certain vulnerabilities, as *finding something would be better than nothing*.

Reducing False Positives vs False Negatives: In the context of program analysis, increasing analysis sensitivity decreases false negatives, while increasing false positives, and vice versa. Contemporary literature asserts that false positives are a major reason for practitioners to avoid using SASTs [167, 156] since *“Developer Happiness is Key”* [248],

and argues that it is necessary to reduce false positives, *in general*. That is, conventional wisdom dictates that developers want lower false positives even at the cost of false negatives, which has led to a significant focus on increasing the precision of SASTs in academia and industry in pursuit of practicality [243, 293, 31, 111, 173, 153, 30].

However, we found considerable evidence that contradicts this understanding of the developers' perspective on the soundness-precision tradeoff, with participants strongly favoring lower false negatives, even at the cost of increased false positives. As P04 and P06 state,

“False negative for sure. I just told you the amount of the price of the bug (in millions), so I don’t care if there are 10 false positives. False negative - that one is going to kill you.” — P04_{Automobile Sensors}

“From my understanding it is actually more threatening that we aren’t even aware of the vulnerability...So to me, false negative should be bigger concern...it (false positives) wastes time of developers, but it is not harmful in the whole picture” — P06_{Software Service}

P14 even argued that false positives indicate a working SAST, and when it comes to security, no stone should be left unturned,

“If you’re getting a bunch of false positives, then that typically means your static code analysis tool is doing its job. ...I’d rather my security tool be annoying and tell me about every single possible issue over it not telling me anything and just letting security things slide through.” — P14_{Law Enforcement}

Finding 10 (\mathcal{F}_{10}) – Nearly all the practitioners expressed a preference for *fewer false negatives*, *i.e.*, as long as the SAST is able to find valid security vulnerabilities, they would tolerate and even prefer few false negatives at the cost of many false positives.

Since existing literature argues that false positive rate for program analysis should not exceed 20% [167, 256, 67, 47], we requested our participants to approximately quantify

their preference (or experience) regarding the acceptable proportion of false positives to true positives. For most participants, this preference was far higher than 20% as long as the tool detected some valid vulnerabilities (i.e., had true positives), as indicated in \mathcal{F}_{10} as well. For instance, P02 admitted to dropping a tool in favor of manual analysis due to overwhelming false positives without a single valid vulnerability:

“I wouldn’t mind wading through 100 false positives, if I thought there were actually going to be genuine issues there” — P02_{OSS} - Java App Server

Some participants expressed tolerance for 80% or more false positives, although not to the extreme extent as P02.

“The acceptable range is for (reducing) one false negative, that there could be five false positive” — P10_{Healthcare}

Further, some, e.g., P09, stated that 80% false positives were common in a tool they were currently using; although they were dismayed by the low number of serious, real vulnerabilities found:

“(At present) 80% of them are actually false positives and 20% of them are actually something we can fix. Even those 20, you don’t generally find serious problems.” — P09_{Fintech}

Finally, P01 expressed a lower tolerance for FPs than most other practitioners, stating that

“We ended up with 20% real issues. 80% just false positives. And one of my last actions in that company before leaving was saying, ‘Hey, look, this tool is a waste of time’” — P01_{Program Analysis for Security}

Finding 11 (\mathcal{F}_{11}) – Practitioners are generally more tolerant of false positives than the 20% upper bound proposed in literature, given their preferences and the tools they currently use, with some finding even 80% or more false positives practical.

Effective False Positives and SAST: Due to the perceived notoriety of false positives

affecting adoption of general static analysis tools, the notion of *effective false positives*, defined as “any report from the tool where a user chooses not to take action to resolve the report” [249], or in other words - letting the developer determine whether *any* reported defect is a false positive, is gaining attention. Effective false positives have further been contextualized in SASTs in the form of letting a developer determine whether a reported vulnerability should be considered as within the scope of security context [296]. However, several participants cautioned that in their experience, developers may not make the right call when it comes to identifying an effective false positive issue *e.g.*, when an insecure code segment is considered “inactive”.

P14 and P15 expressed something similar to “*The Developer is the enemy*” threat model [299]. P15 argues that “*Junior developers don’t understand what is the impact*”, and P14 states (on effective false positives): “*From a security standpoint, you can’t really trust, you shouldn’t trust other devs, and users to always know that something could be potentially insecure. So you need to make sure that it’s not possible for it to happen or reduce the possibility of it happening as much as possible (by removing insecure code)*”. Furthermore, P14 cautioned that developers may habitually mark an actual issue as false positives erroneously,

“It seems familiar, but it may be new. And then you’re just going to ignore it because it’s close enough to something you’ve seen in the past, and you just say that it’s OK. So we do need to be vigilant on those false positives to make sure that they are truly false positives” — P14_{Law Enforcement}

P04 made a similar remark about making mistakes in deciding whether to run SAST or not on code patches, sharing that they had a vulnerability that could’ve been detected using static analysis, but was not due to the deliberate decision of not using SAST, costing millions:

“The undisclosed amount is in a couple of millions. ... We had two static analysis tools which should be used, but the decision from the management was because it was

a minor fix that they did not use them” — P04_{Automobile Sensors}

Finding 12 (\mathcal{F}_{12}) – Practitioners are generally against letting developers define “effective” false positives, or letting them decide when to run SASTs. This reservation stems from their prior experience of the adverse cost of leaving a vulnerability in the code, and/or from their knowledge of developers (1) lacking an understanding of the impact of vulnerabilities, (2) being prone to incorrectly marking actual issues as false positives, (3) being untrustworthy/biased towards marking issues as effective false positives.

5.2.5 Impact of Unsound/Flawed SAST

After learning about what participants expect from SASTs, we aimed to understand if and how participants were impacted by flaws in SASTs, *i.e.*, their inability to detect what they claim as “in scope”, how participants generally addressed the flaws, and their experiences reporting the flaws to SASTs.

Impact of Unsound SAST: All developers across survey and interviews, save for a few, shared that while they had experienced false negatives, they had not experienced any adverse impact due to flaws/unsoundness in SASTs. The practitioners explained that while false negatives are not observable since they are not reported by the SAST, they expect manual/code reviews to detect vulnerabilities missed by SASTs. Therefore, as any false negatives resulting from even unknown flaws in SASTs are addressed by their manual reviews, unsound SASTs do not impact their software.

Finding 13 (\mathcal{F}_{13}) – Practitioners are not overly concerned about the impact of unknown unsoundness issues in SASTs, as they expect subsequent manual reviews to find what the SAST missed.

P18, who works with an internal static analysis team, offered an alternate explanation as to why developers may overlook false negatives of SASTs, or their impact, because the assumption is that SASTs *just work*

“If the tools miss something, we can not detect that issue, and we just overlook the

issues... because no one ever reports about false negatives, and we don't check if the tool ever miss the vulnerabilities" — P18 Fortune 500 Global R&D Center

Finding 14 (\mathcal{F}_{14}) – Developers may use SASTs in a state of denial, i.e., assume that SASTs just work, and hence, simply overlook any evidence of false negatives, or flaws in the SASTs that lead to false negatives.

Among the exceptions, P02's organization tried and stopped using SAST because of false negatives, thus effectively negating any potential impact, as previously described in 5.2.4. On the other hand, P01 shared that while their own SAST product unintentionally introduced a vulnerability, which could've impacted their clients, *"never public, no customer ever suffered"*, as it was detected during development.

Addressing/Reporting flaws to SASTs: Participants expressed that security is important, but shared challenges associated with reporting flaws to SASTs.

Generally, flaw reports consist of either a minimal code example that demonstrates the flaw, or actual code snippet from software. However, P04 and P09 shared that going for either is problematic for two very different reasons. First, sharing actual code snippet may require going against company or client's confidentiality policy. P04 circumvents this because of a pre-existing NDA between their organization and the SAST, *"we have an NDA signed, so in case I cannot get a small example, they can also check our source code"*, whereas P09 is unable to do so.

"For certain external communications, it's a little bit difficult to do. What we can share with third party or other party is very strictly regulated by the state bank... If we want something from <tool>, we have to justify why we are sharing this particular code snippet. In particular, I think if you don't share a large amount of code with them, they won't even be able to tell why this is problematic" — P09_{Fintech}

On the other hand, several participants stated that sometimes, developers are not willing to report flaws since it is *"additional work"* (i.e., reporting the flaw, following up):

"We were asked to not do things on our own, because they will maybe increase more

pressure ... I would actually report it to my team lead, but I don't think they would actually report it to back to them" — P05_{Web Applications}

"That might not happen as well because inherently developers are lazy. If you want to share this, you have to go through with certain things" — P11_{Website Backend of Program Analysis for Security}

"That might not happen as well because inherently developers are lazy. If you want to share this, you have to go through with certain things" — P11_{Website Backend of Program Analysis for Security}

Finally, some participants shared that while they have reported flaws to SASTs, the lack of response, or lack of addressing flaws discouraged them from reporting flaws later on. P02 said, *"Nothing as far as I recall"* when asked about whether anything happened after reporting false negatives to SAST, whereas P04 said that some SAST developers might be unwilling to accept a flaw as an issue.

"So, $\langle \text{SAST}_A \rangle$, we have a worse experience. They are mostly evasive, so they are not really progressing as $\langle \text{SAST}_B \rangle$. It takes a lot of time to convince them that they are bugs. Even though you have a small example, they still ask you to try different configurations and all that stuff, but we were aware of that before we came to this part, before we selected them. Because simply they ($\langle \text{SAST}_A \rangle$) are, I wouldn't say confident, but they are confident that their solution works." — P04_{Automobile Sensors}

Finding 15 (\mathcal{F}_{15}) – Participants may hesitate to report flaws/false negatives in SASTs for several reasons, ranging from prior negative experiences with SASTs (including inaction on reported flaws), or issues internal to the organization, such as the need to maintain product confidentiality (without an explicit NDA), red tape, and the lack of incentive to perform the additional effort.

P01 shared some insight to decisions related to fixing flaws in SAST, sharing that while severity and likeliness ("correlates to presence in open source libraries") are motivating factors, so is what the business-competitors are detecting. To understand this in-depth, we presented a hypothetical scenario to P01 where a class of vulnerability is ignored by the rest of the SAST building industry and asked how is it decided whether to address it

in their SAST. P01’s response was *“It depends on the effort and depends on how critical it is”*.

Exploiting Flaws and Evasive Developers: We adopted the concept of evasive developers from [19, 299], defined as a developer who actively attempts to bypass a SAST’s checks. The motives vary, such as malice, lack of stake (third-party contractor), and/or simply being lazy. A majority of the participants stated that while they consider evasive developers realistic, such developers are unlikely to cause serious harm in their organizational context due to several factors, such as company policies *e.g.*, *“It is strictly prohibited, and it is communicated in that way that it is not acceptable to bypass those checks (P08)”*, and manual code reviews.

“The process that we have is designed that, first, it needs to pass the review of the initial reviewer which allows it to get it on the main branch. So if we, put another hurdle here and we say that there are two friends which decide that this is okay, it still needs to come through the third guy who is gonna test, the test will kill. So that’s already three guys that would need to accept the issue in the whole team.” —

P04_{Automobile Sensors}

On the other hand, some participants shared that they have observed their colleagues being evasive, or they themselves attempted to be evasive due to stressed work environment.

“We had six people and one would actually do something like that.” — P05_{Web Applications}

“There was an extreme pressure because we needed to bypass the SAST tests, otherwise we would not receive green flag from the security team. So it actually happened once.

We used to work late night to resolve all those conflicts and red flags.” — P06_{Software Service}

In contrast, P01 expressed that in an organization a developer being evasive is unlikely due to ownership at their organization, *“I want to believe that our developers are responsible ... I don’t believe anyone will try to game our system like that”*.

Finding 16 (\mathcal{F}_{16}) – The risk of evasive developers is real. That is, while some participants consider the scenario of “evasive developers” as adequately prevented by existing code reviews, this optimism is not universal: others have prior experience of evasive developers in their teams, or have evaded SASTs themselves.

5.2.6 Challenges and Improvements

Finally, we wanted to learn about the pain-points of participants related to SASTs. Our approach was to present hypothetical (but ideal) scenarios, such as unlimited resources to fix or address just one issue of SAST, with the goal of getting the participants to focus on the most severe SAST-specific issues in their perspective.

A few participants wanted to invest their resources on improving analysis techniques, both for reducing false negatives *e.g.*, “*I guess the first thing would be I would try to make it so that we’re covering all of the most obvious*” (P14), and providing meaningful alert messages *e.g.*,

“So the static analysis tool should be able to detect all the security issues within its scope and within possibilities. It should show meaningful messages . . . it should expose enough information about the issue so that the respective developer can address the issue easily” — P10_{Healthcare}

Alternately, P02 (who was generally unimpressed by SASTs throughout the study) wanted unlimited human resources for manual analysis:

“If I’ve got unlimited time and resources, then some poor, unfortunate soul is going . . . going to have to go through all of the false positives in SAST and just confirm that they are actually false positives because there’s just so many of them. . . . If those unlimited resources included some experienced security researchers, I get them doing some manual analysis. Because to be perfectly honest, the best vulnerability reports we get, which generally tends to be the more serious issues, they’re not found by tools, they’re found by people” — P02_{OSS - Server}

Several participants focused on SAST CI/CD integration issues explaining that often configuration is a major pain-point for them *e.g.*, “*I would definitely say integrations would be the top. ... I think the best example would just be for all major CI/CDs to have an open source example of how to implement and integrate with various things.*”. Other responses covered niches, such as better language-specific support, concurrency and abstraction support.

Finally, participants generally agreed that actionable reports that explain what can be done to address an issue, or provide more context, would be useful *e.g.*, “*if you write this code like this, this issue should be resolved*” (P12), and “*An explanation of why the tool flagged that particular code is very helpful. It saves us having to second guess on why is the tool reporting that*” (P02).

Finding 17 (\mathcal{F}_{17}) – The key pain points for developers when it comes to SAST tools include: false negatives, lack of meaningful alert messages/reports, and configuration/integration into product CI/CD pipelines.

5.3 Threats to Validity

This study seeks to understand the diverse perspectives of practitioners with different types of business and security needs, and is affected by the following threats to validity:

Internal Validity: Practitioners with different experiences and roles at organizations may provide responses influenced by over/under-reporting, self-censorship, recall, and sampling bias. We mitigated these factors by asking participants to share organization-specific incidents and experiences, with follow-up questions to understand their context, and reassuring that the responses would remain anonymous and untraceable (Section 5.1.3). Moreover, some participants may have experienced loss of agency in selecting SASTs (*e.g.*, P08, \mathcal{F}_4). However, all our participants have played key roles in selecting *or* using SASTs in their organizations (see Section 5.2.3), leading to useful experiences and observations that reveal meaningful patterns in SAST selection.

External Validity: Due to the nature of interview-based qualitative research focusing on a specific experience (here: with SAST), *generalizability* is considered an issue for recruitment through snowball/convenience sampling. Findings from such studies are considered “*softly generalisable*” [56]. Prior research demonstrates that such studies are reliable for identifying salient trends [36, 90]; indeed, given the diverse organizational and product contexts of our participants, their responses provide key insight into how SASTs are used in practice in complex organizations.

In other words, given the number of our participants (n=20), and the recruitment process, we do not claim that the participants are representative of the broader developer population, or that the findings are *generalizable*. That said, this study captures and analyzes the experiences of participants from diverse organizational and security contexts and offers salient insights related to the use of SASTs in practice.

5.4 Discussion

The findings from our study reveal salient aspects of how developers use SASTs, what they expect from them, and how they react when SASTs do not fulfill those expectations. We now distill the findings into four themes related to the problems inherent in the use and perceptions of SASTs as well as a path forward for researchers and practitioners.

5.4.1 Mind the Gap: The Dichotomy of Perceived Developer Needs and SAST Selection/Evaluation

A common sentiment observed throughout this study is that practitioners do care about and prioritize security. To elaborate, practitioners stated that they would generally fix vulnerabilities regardless of release deadlines (\mathcal{F}_1), except in certain mitigating circumstances (\mathcal{F}_2 , \mathcal{F}_3), and use SASTs to cover the blind spots and subjectivity pertinent to manual code analysis (\mathcal{F}_6). Moreover, nearly all practitioners favored lower false negatives (*i.e.*, not “letting security things slide through”) (\mathcal{F}_{10}), expressing a surprising tolerance for false positives as long as SASTs found vulnerabilities (\mathcal{F}_{11}).

However, we found that this strong preference for security, and particularly SASTs that find real vulnerabilities, is not reflected in how practitioners select SASTs. To elaborate, practitioners select SASTs based on cost, corporate pressure, ease of integration/use, and particularly, recommendations from peers and general reputation of the tool (\mathcal{F}_5). This generally ad-hoc and subjective criteria does not provide objective evidence of a SAST’s performance in detecting vulnerabilities. Hence, there is a clear *gap* between the criteria that practitioners use for selecting SASTs, and what they want most from SASTs (evidence of real vulnerability detection abilities).

5.4.2 The Power of Reputation and the Lack of Reliable Objective Criteria

The key question is, *why does this gap exist? That is, why don’t practitioners evaluate the security properties of SASTs?* Our findings point to two key reasons:

First, we find that practitioners may not have any motivation to evaluate SASTs. That is, practitioners seem to be unreasonably optimistic about the SASTs’ abilities, assuming that SASTs must detect everything they claim to (*i.e.*, define as within scope) (\mathcal{F}_8), and assume that SASTs “just work” (\mathcal{F}_{14}). This optimism, coupled with their reliance on reputation as a valid metric for selecting SASTs (\mathcal{F}_4), may be sufficient to dissuade practitioners from any additional effort required to evaluate SASTs. Thus, the observed lack of motivation to evaluate SASTs is concerning, particularly as the blind belief practitioners express in SASTs and their reputation does not hold up to scrutiny: *e.g.*, a recent evaluation of reputed crypto-API vulnerability detectors showed serious, previously unknown flaws, which prevent the detectors from finding vulnerabilities they consider “in scope” [19].

Second, even when practitioners want to evaluate SASTs, the existing means to do so, *i.e.*, benchmarks, are perceived as insufficient. As we found, while some practitioners may be unaware of benchmarks for evaluating SASTs, most are not. In fact, most practitioners *do not trust existing benchmarks*, viewing them as either too basic (and not representative

of real, complex, vulnerabilities), or biased (\mathcal{F}_5). These findings indicate a significant gap in the research on evaluating SASTs, and motivate the development of high-quality, comprehensive, real-world benchmarks vetted by both researchers and practitioners, if we intend to help practitioners objectively evaluate SASTs for what they most desire: the ability to detect vulnerabilities.

5.4.3 Giving Developers What They Want

We observe that practitioners repeatedly expressed that they want two things from SASTs: ease of configuration (\mathcal{F}_4 , \mathcal{F}_{17}), and for tools to detect real vulnerabilities (\mathcal{F}_8).

Fortunately, the ease of configuration is being addressed by the recent, additional support for SASTs through integration into CI/CD pipelines of open source projects, such as via Github Actions [3, 2], as well as standardized output formats, such as SARIF [216]. However, the latter is harder to achieve at present. That is, while our practitioners repeatedly expressed that they want SASTs to be first and foremost able to find critical vulnerabilities (and all those considered within scope, \mathcal{F}_8), even at the cost of higher number of false positives (\mathcal{F}_{10} , \mathcal{F}_{11}), the research community continues to show preference towards improving precision instead, *i.e.*, decreasing false positives, for SAST tools throughout the last decade [293, 31, 153, 243, 47, 256, 248]. Thus, for SASTs to actually be useful, the research and industry communities need to refocus their efforts towards finding critical vulnerabilities (and all that is deemed within scope), with improved precision being an additional, desired, property.

5.4.4 Industry is not prepared for the flaws of SASTs

Our findings expose a *critical paradox* in the assumptions industry practitioners make about their approach towards SASTs: While practitioners do expect SASTs to detect all vulnerabilities within scope (\mathcal{F}_8), they are not overly concerned with SASTs missing such vulnerabilities due to undocumented flaws, because their subsequent manual analysis to find what the SASTs missed (\mathcal{F}_{13}). However, practitioners also emphasized that their key reason for using SASTs is to account for knowledge gaps, blind spots, and subjectivity

inherent in manual analysis (\mathcal{F}_6 , \mathcal{F}_{12}). To summarize the paradox, practitioners use SASTs to account for gaps in manual analysis, but then, in turn, are confident that manual analysis will account for (unknown) SAST flaws.

This paradox suggests several undesirable aspects of the status quo. First, that developers may be overly confident in guarantees offered by their process of combining SASTs (or other tools) and manual analysis, or may simply take the reports of SASTs at face value (\mathcal{F}_{14}), which may result in undetected vulnerabilities in code that are missed by both SASTs and manual analysis; *e.g.*, previous work has shown that the same undocumented flaws can manifest in any number of SASTs, and lead to vulnerabilities in programs analyzed by the SASTs [19, 21]. Second, given that practitioners generally hesitate to report flaws in SASTs (\mathcal{F}_{15}), the flaws in a SAST would persist and harm most software using the SAST, even if a few practitioners do uncover false negatives/flaws during manual analysis. That is, if the status quo observed in this study continues, SASTs will likely never improve in their ability to detect vulnerabilities, but instead, will continue to be used in a manner that inspires a false sense of security among practitioners.

To summarize, we conclude that the industry is ill-equipped to find or address any flaws in SASTs, particularly given the state of current reporting processes that are mired in confidentiality issues, an evasive attitude, and lack of response from SASTs (\mathcal{F}_{15}). Thus, practitioners are stuck with repurposing common issue submission processes that does not cater to their confidentiality needs, does not elicit a response, and does not facilitate discussion.

5.4.5 Moving Forward: New Directions and Ideas

To improve this status quo, researchers and practitioners need to establish a dedicated process for reporting false negatives, as well as expectations from SASTs upon receiving such reports, in a manner similar to bug reporting expectations for typical software products. This might involve the development of automated methods for creating minimal examples of vulnerabilities missed by SASTs or even “self-healing” SASTs that leverage advance-

ments in automated program repair to address missed vulnerabilities. Moreover, future work may also explore streamlining the automated evaluation of SASTs (*e.g.*, developing web-based services that allow practitioners to “test” SASTs with realistic vulnerabilities), so that developers may be able to evaluate SASTs before using them, instead of leveraging subjective criteria for the same. Beyond evaluation techniques, researchers should also consider orienting future work on SAST development toward the high preference of practitioners in finding important/critical vulnerabilities, even at the expense of a high number of false positives.

To summarize, only by raising awareness about the flaws in SASTs, aligning their goals with the goals of developers, designing protocols for evaluating them, and streamlining bug reporting, particularly for false negatives, can we move towards a more desired state where practitioners are able to leverage SASTs to their true potential, resulting in a holistic reduction in hard-to-find vulnerabilities.

5.5 Chapter Summary

This chapter provides a comprehensive understanding of how practitioners with diverse business and security needs choose SASTs, and their perspectives and assumptions about limitations of SASTs. By qualitatively analyzing the responses from 20 in-depth interviews, we uncover 17 key findings that demonstrate that contrary to existing literature, practitioners have a higher level of tolerance for false positives, and prioritize avoiding false negatives. Moreover, we find that practitioners, regardless of their strong preference for security, rely on reputation to choose SASTs, as they *do not trust benchmarks* or find them reliable. Finally, practitioners may be overconfident in assuming their ability to address a SAST’s flaw with manual analysis, and are generally hesitant to report such flaws. We conclude with research directions towards automated evaluation of SASTs, aligning SASTs with what developers desire, and creating dedicated protocols for reporting flaws in SASTs.

Chapter 6

Identifying the Factors in the Lifecycle of Bugs in vulnerability detectors

Software security continues to gain attention from academia, the private, and public sectors alike. This is due to recent string of cyber attacks that are affecting private homes [151, 258], safety-critical systems such as healthcare with near fatal incidents [215], and the SolarWinds cyberattack [107]. As a result, fortifying the security of software and services that we use today through our mobile devices, smart devices, and personal computers has become of significant importance. For example, the U.S. Government Accountability Office requested for response from both public and private sectors about possible ways to increase the effectiveness of security practices [222]. As part of this fortification, new approaches are being introduced (*e.g.*, Software Bill of Materials (SBOM) [118]) and studied [278], certification of end-user software / applications is gaining attention [194], and existing approaches of automated security testing approaches are being improved [19]. Indeed, an Executive Order of the US Govt. explicitly asks publishing guidelines that includes *regularly* employing automated tools to check for known vulnerabilities [147]. Additionally, newly introduced standardized output formats of such automated security

testing tools, such as Static Analysis Result Interchange Format (SARIF) [216], are now enabling widespread integration in nearly every stage of software development lifecycle through continuous integration/continuous development (CI/CD) pipelines, *e.g.*, GitHub CodeScan Initiative [120].

However, such automated vulnerability detection (and remediation) tools are not free from flaws. As shown in studies (*e.g.*, [19, 21]), these tools, similar to any other software, may suffer from design and implementation bugs. As a result, these tools may not detect vulnerabilities that are well within their scope. However, unlike traditional bugs, the false-negative inducing bugs often remain invisible because of their non-functional nature. Moreover, as we discussed in Chapter 5, we found that submitting false-negative related issues and addressing those is not straightforward (Chapter 5.2.5). Fixing such an issue may require providing access to sensitive code or creating minimal, bug-inducing examples. Furthermore, such cases must be aligned with the existing design goals and threat model defined by the vulnerability detector developers (Chapter 4.9), which are often implicit or not communicated with the vulnerability detector users (Chapter 5.2). As a result, a soundness compromising bug that is still within the technical scope (*e.g.*, statically analyzable) of a vulnerability detector may still be considered “out-of-scope” by vulnerability detector developers.

Because bugs in vulnerability detectors directly compromise the security guarantees expected from software services, it is, therefore, essential to learn about the lifecycle of vulnerability detection compromising bugs if we want to address **RQ2** *comprehensively*. In other words, it is necessary to learn, analyze, and understand the implicit assumptions made by the designers of the vulnerability detectors, who are also practitioners. More specifically, it is important to learn:

- The factors that influence the decision-making process of addressing (or not addressing) such bugs,
- the evolution or change in the decision-making process in the wild by vulnerability detector developers, and

- patterns in triaging vulnerability detector bugs.

While it is possible to learn about these implicit assumptions made by the designers of the vulnerability detectors through qualitative interviews (Similar to the study discussed in Chapter 5), it is also possible to learn about the implicit assumptions, design choices, and preferences made explicit by the designers through their publicly accessible bug management systems used over the years. Only by identifying and learning about these choices can we hope to understand the limitations and advantages of such choices made independently by each of these tools, and we can help develop existing and future tools by addressing those limitations.

6.1 Research Methodology

To address **RQ2** from the tool designers' end, *i.e.*, learning the factors that influence the addressing of bugs in vulnerability detectors, we performed a study that consisted of (a) mining software repositories to identify relevant bug-reports/issues, and (b) qualitatively analyzing those issues (*i.e.*, title, content, interactions in the forms of comments, and reactions). We now detail these two steps.

6.1.1 Identifying and Collecting Bug Reports

For this step, we need to select vulnerability detectors that are representative of the real world, *i.e.*, has a prominent presence in online and the industry, and have publicly available discussion boards of issues (*e.g.*, GitHub). Furthermore, the vulnerability detectors need to be

- (a) in active development and exist for at least two years. This is necessary, as a vulnerability detector needs to have a history as a software repository that we can extract information from, with developed practices refined throughout years.
- (b) in active use by practitioners, *i.e.*, it is used in industry, listed by communities (*e.g.*, analysis-tools.dev) and/or OWASP, public service sites, such as NIST.

Next, we collect issues of two categories to create a statistically significant, representative sample of the issues discussed per tool. In the context of this study, a straightforward approach would be to only analyze issues that mention keywords such as false positive and false negative, along with their different formats (*e.g.*, FP, false-positive) in the title, description, label, and/or description of the issues. To prioritize selecting samples of issues, a large number of interactions (*e.g.*, comments and/or emoji based reactions) can be used to filter and/or sort those issues.

However, as we explored the repositories to understand their structure and practices, we noticed a few factors that need to be considered while considering such a straightforward approach.

- (a) **Use of Alternative Terms** A software practitioner may choose to submit an issue using alternative words, such as “*X tool not detecting Y*”, “*false alarm*”, and “*triggers warning*” instead of using specific keywords, *e.g.*, “*false negative*” or “*false positive*”.
- (b) **Absence of Specific Terms** The maintainers of the tool may not explicitly allocate labels for “false-negative” and/or “false-positive” available. For example, while GitHub/CodeQL has “False-positive” label for issues, it does not use any label for specifically marking “false-negative” as of April 2025 [117], even though it positions itself as a tool for “*automating security checks*” by developers and “*identifying variants of vulnerability*” by security researchers.
- (c) **Urgent Issues can get addressed and closed without Interaction** We noticed that critical, urgent issues can and will get addressed and closed without any significant number of interactions or no interactions at all. *e.g.*, a false positive related issue was addressed and closed without any interaction at all in less than a month [116] in Github CodeQL repository.

Considering these factors, we adjusted our approach to collect issues, both open and closed types, based on both (a) the straightforward approach and (b) a randomized approach that accounts for these factors. That is, in addition to including issues that are

interacted most, or least, (comments, reactions, and both comments and reactions) and contain the keywords (or their different forms, *e.g.*, short, abbreviated), we additionally analyze a portion of the issues randomly sampled from the repositories.

We now provide detailed steps with respect to an example repository, X. For convenience, we assume it has $N = 134$ issues, and the statistically significant number of sample issues for 95% confidence for this repository is $n = 100$. We split the sample size to several categories, with each category having specific search criteria based on the factors mentioned earlier.

- (a) sorted by the number of comments, 10% of the sample size issues (which equals to 10 for this example repository) with the highest, or lowest number of comments. In this case, this would be selecting 5 issues each from the top and the bottom from the list of issues that is sorted by the number of comments. This is done so that we get an overview of issues that are most interacted/least interacted within that particular tool's community. Similarly,
- (b) sorted by the number of interactions, 10 issues with highest or lowest number of emoji based interactions,
- (c) sorted by the number of comments, 30 issues with highest and lowest number of comments that contain specific, *case-insensitive keywords* related to false-positive (fp, false positive, false-positive, and FPs) and false-negative (fn, false negative, false-negative, FNs) in the titles or description of the reports,
- (d) sorted by the number of comments, 40 issues with highest and lowest number of comments, while additionally having *labels* related to bugs *e.g.*, as wontfix, bug, error, error-reporting. Since each tool come with their own labels, we consider labels for each repository individually.
- (e) randomly chosen 10 issues.

Note that it may not always be possible to find the calculated number of issues for a particular category, *e.g.*, issues that are labeled as “false-positive”. In such cases, we

collected all issues that satisfy the search criteria of that particular category. In addition to this, we ensured that each of the issues collected in each category are *unique*, *i.e.*, an issue that satisfies both criterion (a) highest number of comments, and (b) highest number of interactions will only be collected once to prevent double-counting.

Note that we chose to analyze statistically significant samples per repository, instead of samples across all repositories (*i.e.*, issues from all repositories), because each tool repository has its own labels, issues, design considerations, and priorities. This is because statistically significant sample size increases diminishingly with population size. That is, for two repositories, each with 100 issues, the statistically significant sample size (95% confidence) would be 132 if both repositories are considered together, whereas the statistically significant sample size would be 160 per repository.

Table 6.1: Breakdown of Repositories, the total number of issues, and statistically significant sample size for 95% confidence as of February 2024

Repository Name	Total Issues	Labels Used	Expected Sample Size/ Collected Size*
CodeQL	1,969	acknowledged, bug, questions	322/274
FindSecBugs	425	false-positive, false-negative, bug, wontfix	202/187
MobSF	1,423	bug, wontfix	303/257
Pyre	380	bug, wontfix	192/119
Semgrep	2,817	error-reporting:parse-error-msg, bug, wontfix, error-reporting	339/296
Total	7,014		2,709

* It may not always be possible to find the expected number of issues under a particular category, *e.g.*, issues that are labeled as “false-positive”. In such cases, we collected all issues that satisfy a particular category criteria.

We applied this approach on the bug repositories/issue discussion boards of five vulnerability detectors with a total of 7,014 issues, resulting in 2,709 sample size. The details of each repository, total issues, and extracted issues are provided in Table 6.1.

6.1.2 Analyzing Data

After collecting unique issues from our target repositories, we systematically labeled and qualitatively analyzed the contents of the issues. We chose reflexive thematic analysis com-

bined with inductive coding for our analytical approach [56], as this approach is suitable for capturing both latent and semantic meaning of data through the subjective interpretation of the researchers, *i.e.*, experience and skills of researchers in analysis. Further, we chose single-coder approach, but with two researchers for “*meaning-making*” from data. This is necessary, considering the large quantity of data. Specifically, while the number of issues is already large (2,709), interactions, such as all comments under an issue are also within the scope of our analysis. For example, the highest number of comments in an issue for the GitHub repository is 62 in the CodeQL repository (Issue#10132). Considering the large number of issues, our collected sample contained 6,432 comments in total. While not all comments are equally complex, elaborate, or significant, each comment still needs to be analyzed to determine their relevance nevertheless, easily adding up the effort. Considering these, we adapted the single-coder approach in the following way to ensure consistency across both *coders* and different *repositories*.

Our data labeling consists of two distinct stages. In the first stage, we labeled the statistically significant sample size of the GitHub CodeQL tool (274 issues) using MAXQDA, a qualitative data analysis software. Two researchers individually labeled the data after an initial exploration to create a code book. After labeling each category of CodeQL issues, both researchers met for the agreement-disagreement meeting and resolved the conflicts to get a Kappa score greater than 80% at sentence level. For each category, the code book was updated to remove or merge redundant codes, and create new codes. This process, consisting of five agreement-disagreement meetings, and additional meetings to discuss, helped us to create a consistent labeling approach for this particular repository.

Afterwards, both authors individually analyzed and labeled issues from the remaining four repositories, namely Semgrep, Find-sec-bugs, MobSF, and Pyre. After completing each tool, both researchers met to discuss analyzed data, interesting issues, introduced new codes, and modified existing codes. This was necessary to adapt the approach to the different repositories. Furthermore, each of the researchers individually maintained a separate document that consisted of *Emerging Thoughts* related to observed patterns, a

step that is considered valuable in thematic analysis of data. Finally, we iterated through the later steps of thematic analysis (identifying potential themes, refining) to finalize the themes.

6.2 Analysis Results

This section describes the results from our analysis, and interpretation of the contents collected from the issues posted in the open-source vulnerability detectors. During the course of our analysis, we noticed that most of the tools have their dedicated, internal communication channels, *e.g.*, Slack Workspace. As a result, processes, such as introducing or modifying key design decisions, prioritizing specific types of bugs, and determining internal policies are not visible publicly. However, we can still gain an understanding of such processes by analyzing the interactions in between the developers and software developers in the publicly available discussion boards, as those internal processes dictate the nature of the visible responses.

For example, we found an instance of user investigating a vulnerability detector as a Proof of Concept (PoC) for future adoption and requesting a feature. The tool designers asked the timeline for PoC, and if they need to “*rush*” it, only to be responded by the user, “*Don’t rush out a release on my account!*”. This particular interaction indicates prioritizing features or bug requests that has business potential, or large scale impact from the vulnerability detectors perspective.

Note that we such interpretations and deduce processes only when these are implied in the interactions. To provide a counter example, we noticed that regardless of being security focused vulnerability detectors, none of the tools except one (Find Security Bugs) used a specific label for false-negative in the issue management board. Furthermore, only two tools used specific label for false positive. While it is certainly possible to interpret these observations as a lack of focus on security specific issues across tools in most cases, it is also possible that such issues are still prioritized nevertheless in an internal board with

the help of issue management “bots”. Therefore, we only report about such ambiguous observations, and refrain from offering possible interpretive insights.

6.2.1 The Balancing act of Vulnerability Alert and Alert Fatigue

From our analysis, we found that while vulnerability detectors aim to detect vulnerabilities, most of these tools also act as multi-purpose tools, *i.e.*, reporting code quality issues, general bugs and performance optimization issues. While these vulnerability detectors are built to serve multiple use-cases, *i.e.*, security auditing, and helping developers write quality code and avoid mistakes that may lead to vulnerabilities, the expectations of software developers tend to overlap across these use cases, which we discuss next.

Tool Effectiveness and User Efficiency: Across the vulnerability detectors, we observe that the tension between software developers and developers towards balancing the effectiveness of vulnerability detectors, *i.e.*, detecting real vulnerabilities, vs user efficiency, *i.e.*, preferring reduced number of false positives, depends on their own use cases. For example, an user argued that they prefer a “*High Signal/Low Noise*” approach in a taint analysis feature request (*Semgrep*#2787). However, the same user elaborated that implementing the requested feature would result in more detections of vulnerabilities (“*good number of TPs*”), and potentially “*many false-negatives*”. In other words, the user expressed preference towards at least partial coverage of vulnerabilities, compared to no coverage, while also expressing that less number of false positives is preferable as long as it works. This preference towards vulnerability detectors that *work*, *i.e.*, finds vulnerabilities that matter, while having less number of false positives is also independently reported by Ami et al [22].

We see similar preference from the developers of Find Security Bugs, where one of the “*main developers*” explicitly expressed that for security, the goal is detect real vulnerabilities, while balancing false positives, *i.e.*, “*Everybody wants no false positives to be reported, the problem is that the analysis is quite complex and I cannot guarantee, that*

no true positives are missed. *FindSecurityBugs* is intended for security code review and should not miss any real problem” (*Find Security Bugs#76*). The user further suggested that reported items should contain high and low confidence scores to help tool software developers prioritize, and address security bugs.

Finding 1 (\mathcal{F}_1) – While users prefer not having false false positives, for security, they prefer having no false negatives over having no false positives.

Furthermore, developers of vulnerability detectors expressed a similar opinion when it comes to detecting vulnerabilities and addressing false-negatives while acknowledging that a sound analysis may not be feasible. *“Fundamentally, the core analysis is neither sound or complete, so it is not surprising that corner cases can be found where improvements are desirable. As such, we do not consider this a “bug”, but rather a pragmatic limitation”*(*GitHub/CodeQL#7106*).

However, in addition to technical feasibility, the “*legitimacy*” of code is considered as an additional factor when it comes to determining the scope of detection by some tool developers, e.g., *“We are implementing heuristic to detect true positive and to eliminate false positive, But in both case we are scoping to code that are written by human or a legitimate developer”*(*Find Security Bugs#559*) and *“... more focus on finding vulnerabilities from developers with good intention”*(*Find Security Bugs#560*).

Finding 2 (\mathcal{F}_2) – Apart from technical/pragmatic limitations, tool designers consider additional factors, such as intention and legitimacy of developer when it comes to addressing a false-negative inducing bug in vulnerability detectors

Prioritizing Bugs based on Visibility: Because of limited resources, developers tend to focus on addressing bugs that are more visible. That is, developers accept that the analysis may not be “sound or complete”, and therefore, it is not surprising to find corner cases of vulnerabilities. Such cases are considered less of a bug, and more of a “*pragmatic limitation*”. However, developers may still consider such corner cases if *“it is both relatively simple and likely to be common in practice”*(*GitHub/CodeQL#7106*).

Finding 3 (\mathcal{F}_3) – Visibility of a bug and simplicity of the solution makes it more likely to be implemented.

Addressing Bugs need Deliberation: Developers of vulnerability detectors prefer designing generic solutions that follow the existing architecture, instead of creating complicated, scenario-specific solutions. *“Improving taint seems like a more generalized solution, and doesn’t further complicate the pattern syntax”*(Semgrep#3085). Moreover, developers were also wary of introducing changes that may result in overall increase of false-negatives because of potential silent errors, *“I do, however, worry that such a change could silently break things in a number of places”*(GitHub/CodeQL#5672). Thus, even if they want to prioritize addressing a particular type of bug, both design considerations and avoiding invisible consequences may delay creating, and deploying a solution.

Finding 4 (\mathcal{F}_4) – Vulnerability detector developers prefer solutions that are generic and do not have invisible consequences.

Vulnerability detectors and Compiler Optimization/Bytecode Conversion: Some vulnerability detector designers reported that external factors, such as compiler optimization and/or bytecode conversion may result in affecting vulnerability analysis. For example, for a bug report that reported that some vulnerabilities are being randomly detected across multiple runs, the tool designers responded that it is possibly due to different optimization techniques being used across multiple versions of compilers, *“It seems worse than just reported..Current theory is the strings in the constant pool are known to be optimized differently among compiler versions.”*(Find Security Bugs#456).

Finding 5 (\mathcal{F}_5) – External factors, such as as compiler optimizations and bytecode conversion techniques, may affect the performance of vulnerability detectors.

Software Developers do not Desire Temporary Workarounds: Sometimes, vulnerability detector designers provide a temporary workaround, such as downgrading the compiler version, when a fix is not immediately available. However, software developers can be wary of such temporary workarounds. For example, *“Unfortunately, I cannot down-*

grade go, but if the fix is poised to be released soon I can wait...”(GitHub/CodeQL#14373).

This is because software developers are wary of technical debts, *i.e.*, switching to the stable release once the update is made available after reverting the temporary workaround may require manual intervention from the software developers’ side.

Finding 6 (\mathcal{F}_6) – While temporary workarounds are offered by tool designers, software developers are wary about such solutions due to the possible technical debt and increased workload that comes with such workarounds.

Version Fragmentation and Ineffective Bug-fixes: A similar problem occurs when the same vulnerability detector is maintained across multiple channels, resulting in *version fragmentation*. That is, while software developers expect bug fixes to be applied for a vulnerability detector available across different distribution channel, tool designers may find it challenging to ensure this across internal and external channels, and various package managers. For example, a tool designer responded that they were unable to reproduce the false-negative related bug, as it is likely that a bug fix was not applied across the internal and external versions of the tool: “...we haven’t updated open source Pyre in a while, so it’s possible it is a bug that we fixed a while ago that newer versions won’t have.”(Pyre#790). Such bugs can also be caused by the different mechanisms of various package managers, *e.g.*, “actually it is homebrew side’s pypi mapping issue (fixed it now)” (Semgrep#9168).

Finding 7 (\mathcal{F}_7) – Maintaining the availability of the same vulnerability detector across multiple distribution channels and versions may result in version fragmentation, *i.e.*, the same bug-fixes may not be applied across all channels.

Security Specific and Generic False Positives: We found that developers of vulnerability detectors that offer additional features, such as code quality checking, may differentiate between security-specific false-positive issues, and non-security issues, *e.g.*, GitHub/-CodeQL developers mentioned that they **prioritize solving security-specific false-positives** “current focus is on improving our security analysis”. Additionally, they also mentioned that they may prioritize solving generic, non-security false-positives if it is re-

ported by a sufficient number of users, “prioritize it if we get enough reports of the same underlying issue in other projects”(GitHub/CodeQL#5813).

Finding 8 (\mathcal{F}_8) – While reducing security specific false positives is prioritized, generic false positives are prioritized only when frequently observed.

6.2.2 Balancing Security-First Features and Developer Happiness

vulnerability detectors strive to be effective from a security-centric perspective, *i.e.*, to be used for security auditing, compliance, or finding vulnerabilities and their variation, and developer-friendly in terms of use, configuration, and alert fatigue. However, by analyzing the issues across the vulnerability detectors in this study, we found that these two desirable goals are often at odds with each other. Furthermore, both tool software developers and developers can take opposing roles when it comes to preferring security assurance over developer happiness and vice versa, which we detail next.

Transparency and Deviation from Expected Behavior: To achieve the goal of developer happiness, *i.e.*, reducing alert fatigue, vulnerability detectors may silently fail to detect issues or skip analyzing files without notifying the user. software developers consider such issues are “worth reporting” as “it is easy to get frustrated by this kind of issue”(Pyre#93).

Finding 9 (\mathcal{F}_9) – software developers consider silent suppression of errors, *e.g.*, failed analysis, a frustrating issue

The tension between security-first and developer happiness also extends towards customizability of vulnerability scanning rules. For example, a user reported that his custom rules related to statements were being matched as both statements and expressions, which he considers a bug. To elaborate, from an analysis perspective, statements (*i.e.*, code segments that end with `;`, *e.g.*, `f○○();`) and expressions (*e.g.*, `f○○()`) are different, and the user’s expectation was that both of these should be distinct. However, the developers of the relevant vulnerability detector pushed back, stating that novice software devel-

opers may not know the difference between statements and expressions, “*since many of our software developers don’t fully understand the difference between statements and expressions, making the engine try to follow a user’s intuitive mental model is a design decision that (I think) benefits the novice user at the expense of some annoyance to the expert.*”(Semgrep#2137). Based on the user’s feedback, however, the developers agreed that this should be explicitly mentioned in the documentation for Transparency.

Finding 10 (\mathcal{F}_{10}) – While most tools additionally offer customization for developers, some tools silently create additional rules based on the custom rule to be ‘novice-friendly’. However, developers may not consider such automatic creation helpful as this potentially results in unpredictable behavior of the vulnerability detector in future

However, even if a behavior is documented, it can be *unexpected* according to the same developers of the vulnerability detector, *e.g.*, “*Per our documentation, this is the “expected” behavior (although perhaps not expected by us!)*”(Semgrep#1414).

Finding 11 (\mathcal{F}_{11}) – software developers expect documentation to detail the limitations of a tool to ensure transparency and for avoiding resubmission of bug reports.

Additionally, some tool designers prefer writing documentation as a guideline, instead of explaining internal processes of a vulnerability detector(*Find Security Bugs#346*).

Finding 12 (\mathcal{F}_{12}) – Some tool designers prefer to exclusively use the documentation for guidelines and examples only.

Caching and False-Negatives: Because of the common tendency of vulnerability detectors to take too much time when it comes to analyzing large software projects, different strategies are being adopted to reduce the runtime, such as optimization by caching results of analysis. However, sometimes, these optimizations are buggy, resulting in silently ignoring vulnerabilities that are well within the analysis scope of the detectors. For example, “*if you were working on a local machine ... perhaps some scans were run using*

cached data, and any uncommitted code changes were not picked up.” (Bearer#1114).

Finding 13 (\mathcal{F}_{13}) – The default behavior of vulnerability detectors to cache analysis may result in missed vulnerability detections.

Alerting Security and Non-Security Issues: We observed a similar tension between the tool designers and software developers about the scope of detection, and reporting issues. For example, one user was hesitant about requesting a feature of introducing coding standard related issue, as those are not related to security. While the developers encouraged such feature requests, *“Semgrep is not intended only for security issues. Its target user are also regular developers.”*, the user pushed back, stating that they have always differentiated between security and non-security issues, since *“security and testing findings are often blockers; but other types are not”*(Semgrep#8074). On the other hand, Find Security Bugs uses a combination of labels (*rank* and *confidence*) to help developers prioritize addressing bugs. However, the lack of a common meta-data based labeling of detection rules may result in compatibility issues when a vulnerability detector is used as a component/plugin, e.g., *“if confidence levels are not supported by plugins, we should change it of course”*. For example, Find Security Bugs can be used as a plugin with SonarQube, even though SonarQube uses different label (*severity*) to help developers prioritize both security and non-security related bugs.

Finding 14 (\mathcal{F}_{14}) – Tool designers may or may not differentiate between security and non-security issues, whereas additional labels accompanying vulnerability report, such as severity, and rank are considered helpful for software developers to prioritize addressing bugs.

Furthermore, tool designers are wary about providing features that make ignoring false positive *“too easy”*; such as inline comments because *“Blanket ignores seem dangerous or at least not ideal”*, as *“AppSec engineers at multiple companies ... the only ones who can ignore...developers can propose ignores, but ... AppSec team for approval.”*(Semgrep#3521).

Finding 15 (\mathcal{F}_{15}) – Tool designers distinguish between Security-specific and development-specific roles when designing suppression mechanisms of vulnerability detectors

Effective False Positive and Bypass Mechanisms: Letting the developers determine whether a reported vulnerability is practically a false positive due to additional factors, such as context, dead code, and use case, is gaining attention as the concept of effective false positive. Therefore, the ease of marking alerts as false positives is an important factor to the users. However, when it comes to security, making this too easy can *introduce confusion* discussed in (*GitHub#11427*). While the software developers pushed for easier suppression of false positives through inline annotations “*I am honestly shocked that there is no inlining mechanism to suppress false positives*”, the developers pushed back stating that while “*it’s a topic that is often debated in the team*”, offering both UI-based and inline suppression can “*lead to significant confusion*” as security teams may want to take a look at suppressed alerts, and by collecting information related to false-positives from the UI, it becomes possible to improve GitHub/CodeQL.

Finding 16 (\mathcal{F}_{16}) – The duality of vulnerability detectors as security-evaluation tools and developer-helping tools may result in conflicting expectations from software developers and tool designers.

6.3 Threats to Validity

In this study, our understanding of the factors that influence the lifecycle of bugs in vulnerability detectors is affected by the following threats to validity:

Internal Validity: The discussions in between developers and users of vulnerability detectors as visible in issue management boards provides us a partial picture, as often these are influenced by internal, privately accessible discussions in between the maintainers of the detectors. We attempted to mitigate it by being as thorough as possible while analyzing the visible contents, and analyzing referred chain of issues/report as relevant.

External Validity: Due to the nature of this study, generalizability is considered an issue due to our sampling of issues from a limited number of vulnerability detectors. While findings from such studies are considered “*softly generalisable*” [56], prior research demonstrates that such studies are still *reliable* for identifying salient trends [36, 90]. Because of the diverse background of users, the selection of different vulnerability detectors, and our systematic way of selecting and extracting information of issues, we believe this study provides valuable insights about the factors that influence the lifecycle of bugs in vulnerability detectors.

6.4 Chapter Summary

In this chapter, we discussed the different roles played by vulnerability detectors, and presented the first, qualitative study of factors that influence the lifecycle of bugs in vulnerability detectors. We systematically selected 2,709 issues, consisting of 6,432 comments from five prominent vulnerability detectors used in the industry and qualitatively analyzed them to identify insights that directly influence how developers and users recognize, prioritize and address the bugs that affect the effective and efficient application of vulnerability detectors. These insights are also helpful for identifying the overlapping and conflicting design and use case assumptions made by both designers and users of vulnerability detectors, enabling researchers to identify future research opportunities.

Chapter 7

Future Work

Because bugs in SASTs directly compromise the security guarantees expected from software services, it is, therefore, essential to learn about the lifecycle of bugs if we want to address *comprehensively*. In other words, it is necessary to learn, analyze, and understand the implicit assumptions made by the designers of the vulnerability detectors, who are also practitioners. More specifically, it is important to learn:

- The factors that results in introducing a soundness-compromising bug, *e.g.*, other bug-fixing commits or new features,
- The factors that influence the decision-making process of addressing (or not addressing) such bugs, and
- the evolution or change in the decision-making process in the wild by vulnerability detector developer.

As this work has shown, reporting false negative issues is of great importance when it comes to improving security analysis tools. However, the effectiveness of such reports is often compromised because of several factors, such as non-disclosure agreements or red-tapes, lack of incentives, non-straightforward approach for reproducibility, and a lack of dedicated processes for reporting false negatives. As the world adopts security analysis tools, so will it become more important to streamline the process for reporting false negative, so that the end-users can participate in improving those tools, and in turn, improve the security of their software and systems.

Chapter 8

Conclusion

Throughout the dissertation, we dissected the security focused vulnerability detectors that are used in the software industry at large, regardless of whether the software is built, maintained, and/or developed by commercial organizations, open source entities, or by hobbyists. In our analysis, we looked at the claims made by prominent vulnerability detectors from the industry, academia, and the open source community, particularly data leak detectors and crypto-detectors. We created and extended novel frameworks, namely μ SE (Chapter 3) and MASC (Chapter 4), to systematically evaluate them.

Through the evaluation, we found that vulnerability detectors can and do contain flaws that make them unsound, *i.e.*, they do not detect vulnerabilities that are well within their scope of detection. Through the responsible disclosure process, and subsequent discussion, we further found that the developers of vulnerability detectors often adopt a technique-centric design stance, without focusing on a security-centric, or security-focused evaluation based approach. As a result, while vulnerability detectors make claims, such as detecting all variants of in-scope vulnerabilities, or to be used for security audits, they end up providing a false sense of security because of various factors, such as having no threat model or a threat model that does not align with the described use-case scenarios. Systematic evaluation frameworks, that are built to evolve, and can evaluate using expressive test-cases that resemble the different use-case scenarios, including hostile reviews.

Because we, as a community of software engineers and security practitioners, are depending more and more on the vulnerability detectors, it is important to understand the expectations and perceptions of practitioners about vulnerability detectors. Hence, we performed a qualitative research (Chapter 5) with industry practitioners from organizations requiring different types of business and security-critical needs. We found that practitioners rarely evaluate vulnerability detectors and depend on word of mouth referral because they lack the means to evaluate based on their own use-cases. Further, while they do consider reducing false-positives an important goal, they want vulnerability detectors that work first in the first place, and are willing to tolerate a relatively large number of false-positives if it means that they can detect important vulnerabilities. further, we found that the industry as a whole is not ready to address the flaws in vulnerability detectors, as discovered flaws by users are rarely reported because of various reasons, such as NDAs, red-tapes, associated difficulties, and lack of incentives.

Finally, to understand how the flaws, or bugs are acknowledged, addressed, prioritized, and delivered to the users, we performed a qualitative study on a statistically significant sample of issues collected from prominent, open-source vulnerability detectors (Chapter 6). We found that the duality of vulnerability detectors, *i.e.*, the use case of finding vulnerabilities in a hostile settings, and the use case of finding mistakes made by developers in a friendly environment, are both served by vulnerability detectors, and can lead to both overlapping, and conflicting factors that influence the lifecycle of bugs. Finally, we discuss the future directions that stem from this dissertation in Chapter 8.

Bibliography

- [1] Coverity Scan - Projects Using Scan. <https://scan.coverity.com/projects>.
- [2] Enabling code scanning for a repository - GitHub Docs. <https://docs.github.com/en/free-pro-team@latest/github/finding-security-vulnerabilities-and-errors-in-your-code/enabling-code-scanning-for-a-repository>. accessed Apr, 2023.
- [3] Features • GitHub Actions. <https://github.com/features/actions>. accessed Apr, 2023.
- [4] OWASP Benchmark — OWASP Foundation. <https://owasp.org/www-project-benchmark/>.
- [5] The State of the Octoverse — The State of the Octoverse explores a year of change with new deep dives into writing code faster, creating documentation and how we build sustainable communities on GitHub. <https://octoverse.github.com/2021/>.
- [6] YOUSRA AAFER, NAN ZHANG, ZHONGWEN ZHANG, XIAO ZHANG, KAI CHEN, XIAOFENG WANG, XIAOYONG ZHOU, WENLIANG DU, AND MICHAEL GRACE. Hare hunting in the wild android: A study on the threat of hanging attribute references. In *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, pages 1248–1259, New York, NY, USA, 2015. ACM.
- [7] YASEMIN ACAR, MICHAEL BACKES, SVEN BUGIEL, SASCHA FAHL, PATRICK MC-

- DANIEL, AND MATTHEW SMITH. SoK: Lessons Learned From Android Security Research For Appified Software Platforms. In *37th IEEE Symposium on Security and Privacy (SP'16)*, 2016.
- [8] YASEMIN ACAR, CHRISTIAN STRANSKY, DOMINIK WERMKE, MICHELLE L. MAZUREK, AND SASCHA FAHL. Security developer studies with GitHub users: Exploring a convenience sample. In *Thirteenth Symposium on Usable Privacy and Security (SOUPS 2017)*, pages 81–95, Santa Clara, CA, July 2017. USENIX Association.
- [9] YASEMIN ACAR, CHRISTIAN STRANSKY, DOMINIK WERMKE, MICHELLE L. MAZUREK, AND SASCHA FAHL. Vulnerability discovery for all: Experiences of marginalization in vulnerability discovery. In *2023 IEEE Symposium on Security and Privacy (SP)*, Santa Clara, CA, May 2023. USENIX Association.
- [10] YASEMIN ACAR, CHRISTIAN STRANSKY, DOMINIK WERMKE, CHARLES WEIR, MICHELLE L. MAZUREK, AND SASCHA FAHL. Developers Need Support, Too: A Survey of Security Advice for Software Developers. In *2017 IEEE Cybersecurity Development (SecDev)*, pages 22–26, September 2017.
- [11] WILLIAM C. ADAMS. Conducting Semi-Structured Interviews. In *Handbook of Practical Program Evaluation*, Kathryn E. Newcomer, Harry P. Hatry, and Joseph S. Wholey, editors, pages 492–505. John Wiley & Sons, Inc., 2015.
- [12] S. AFROSE, S. RAHAMAN, AND D. YAO. CryptoAPI-Bench: A comprehensive benchmark on java cryptographic API misuses. In *2019 IEEE Cybersecurity Development (SecDev)*, pages 49–61, September 2019.
- [13] SHARMIN AFROSE, YA XIAO, SAZZADUR RAHAMAN, BARTON MILLER, AND DAN-FENG DAPHNE YAO. Evaluation of Static Vulnerability Detection Tools with Java Cryptographic API Benchmarks. *IEEE Transactions on Software Engineering*, 2022.

- [14] NOURA ALOMAR, PRIMAL WIJESEKERA, EDWARD QIU, AND SERGE EGELMAN. “You’ve got your nice list of bugs, now what?” vulnerability discovery and management processes in the wild. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, pages 319–339, 2020.
- [15] AMIT SEAL AMI. MASC Artifact, April 2021. <https://github.com/Secure-Platforms-Lab-W-M/MASC>.
- [16] AMIT SEAL AMI. MASC Online Appendix, July 2024. <https://github.com/Secure-Platforms-Lab-W-M/masc-journal-artifact>.
- [17] AMIT SEAL AMI, SYED YUSUF AHMED, RADOWAN MAHMUD REDOY, NATHAN COOPER, KAUSHAL KAFLE, KEVIN MORAN, ADWAIT NADKARNI, AND DENYS POSHYVANYK. MASC: A Tool for Mutation-based Evaluation of Static Crypto-API Misuse Detectors. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023*, New York, NY, USA, December 2023. Association for Computing Machinery.
- [18] AMIT SEAL AMI, SYED YUSUF AHMED, RADOWAN MAHMUD REDOY, NATHAN COOPER, KAUSHAL KAFLE, KEVIN MORAN, DENYS POSHYVANYK, AND ADWAIT NADKARNI. MASC: A Tool for Mutation-Based Evaluation of Static Crypto-API Misuse Detectors. In *Proceedings of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE’23), Demonstration Track*, San Francisco, December 2023.
- [19] AMIT SEAL AMI, NATHAN COOPER, KAUSHAL KAFLE, KEVIN MORAN, DENYS POSHYVANYK, AND ADWAIT NADKARNI. Why Crypto-detectors Fail: A Systematic Evaluation of Cryptographic Misuse Detection Techniques. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 397–414, San Francisco, CA, USA, May 2022. IEEE Computer Society.

- [20] AMIT SEAL AMI, KAUSHAL KAFLE, KEVIN MORAN, ADWAIT NADKARNI, AND DENYS POSHYVANYK. Demo: Mutation-based Evaluation of Security-focused Static Analysis Tools for Android. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21), Formal Tool Demonstration Track*, May 2021.
- [21] AMIT SEAL AMI, KAUSHAL KAFLE, KEVIN MORAN, ADWAIT NADKARNI, AND DENYS POSHYVANYK. Systematic Mutation-Based Evaluation of the Soundness of Security-Focused Android Static Analysis Techniques. *ACM Transactions on Privacy and Security*, 24(3):15:1–15:37, February 2021.
- [22] AMIT SEAL AMI, KEVIN MORAN, DENYS POSHYVANYK, AND ADWAIT NADKARNI. “False negative - that one is going to kill you” - Understanding Industry Perspectives of Static Analysis based Security Testing. In *Proceedings of the 2024 IEEE Symposium on Security and Privacy (S&P)*, San Francisco, CA, USA, May 2024. IEEE Computer Society.
- [23] ROSS ANDERSON. Why cryptosystems fail. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 215–227, December 1993.
- [24] ROSS ANDERSON. *Security Engineering: A Guide to Building Dependable Distributed Systems*, chapter 28. Wiley, New York, 3rd edition, 2020.
- [25] Security with HTTPS and SSL - Android Developers, November 2020. <https://developer.android.com/training/articles/security-ssl>.
- [26] ANDROID DEVELOPERS. Fragments. <https://developer.android.com/guide/components/fragments.html>.
- [27] DENNIS APPELT, CU DUY NGUYEN, LIONEL C. BRIAND, AND NADIA ALSHAH-WAN. Automated testing for SQL injection vulnerabilities: an input mutation ap-

- proach. In *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, pages 259–269, 2014.
- [28] JORGE ARANDA AND GINA VENOLIA. The secret life of bugs: Going past the errors and omissions in software repositories. In *2009 IEEE 31st International Conference on Software Engineering*, pages 298–308, May 2009.
- [29] DANIEL ARP, MICHAEL SPREITZENBARTH, MALTE HÜBNER, HUGO GASCON, AND KONRAD RIECK. DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, February 2014.
- [30] STEVEN ARZT AND ERIC BODDEN. StubDroid: Automatic inference of precise data-flow summaries for the android framework. In *International Conference for Software Engineering (ICSE)*, May 2016.
- [31] STEVEN ARZT, SIEGFRIED RASTHOFER, CHRISTIAN FRITZ, ERIC BODDEN, ALEXANDRE BARTEL, JACQUES KLEIN, YVES LE TRAON, DAMIEN OCTEAU, AND PATRICK MCDANIEL. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI '14*, pages 259–269, Edinburgh, United Kingdom, 2013. ACM Press.
- [32] STEVEN ARZT, SIEGFRIED RASTHOFER, CHRISTIAN FRITZ, ERIC BODDEN, ALEXANDRE BARTEL, JACQUES KLEIN, YVES LE TRAON, DAMIEN OCTEAU, AND PATRICK MCDANIEL. FlowDroid: Precise Context, Flow, Field, Object-sensitive and Lifecycle-aware Taint Analysis for Android Apps. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.
- [33] STEVEN ARZT, SIEGFRIED RASTHOFER, ENRICO LOVAT, AND ERIC BODDEN. DroidForce: Enforcing complex, data-centric, system-wide policies in android. In *In-*

- ternational Conference on Availability, Reliability and Security (ARES 2014)*, pages 40–49. IEEE, September 2014.
- [34] HALA ASSAL AND SONIA CHIASSON. 'Think secure from the beginning': A Survey with Software Developers. In *Proceedings of the 2019 CHI Conference on Human Factors in Computing Systems*, pages 1–13, Glasgow Scotland Uk, May 2019. ACM.
- [35] KATHY WAIN YEE AU, YI FAN ZHOU, ZHEN HUANG, AND DAVID LIE. PScout: Analyzing the Android Permission Specification. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 217–228, 2012.
- [36] DAVID AUSTEN-SMITH AND JEFFREY S. BANKS. Information Aggregation, Rationality, and the Condorcet Jury Theorem. *The American Political Science Review*, 90(1):34–45, 1996.
- [37] VITALII AVDIHENKO, KONSTANTIN KUZNETSOV, ALESSANDRA GORLA, ANDREAS ZELLER, STEVEN ARZT, SIEGFRIED RASTHOFER, AND ERIC BODDEN. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436, May 2015.
- [38] NATHANIEL AYEWAH AND WILLIAM PUGH. A Report on a Survey and Study of Static Analysis Users. In *Proceedings of the 2008 Workshop on Defects in Large Software Systems - DEFECTS '08*, page 1, Seattle, Washington, 2008. ACM Press.
- [39] NATHANIEL AYEWAH, WILLIAM PUGH, DAVID HOVEMEYER, J. DAVID MORGENTHALER, AND JOHN PENIX. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, September 2008.
- [40] NATHANIEL AYEWAH, WILLIAM PUGH, J. DAVID MORGENTHALER, JOHN PENIX, AND YUQIAN ZHOU. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program*

- Analysis for Software Tools and Engineering - PASTE '07*, pages 1–8, San Diego, California, USA, 2007. ACM Press.
- [41] MICHAEL BACKES, SVEN BUGIEL, ERIK DERR, SEBASTIAN GERLING, AND CHRISTIAN HAMMER. R-Droid: Leveraging Android App Analysis with Static Slice Optimization. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '16, pages 129–140, New York, NY, USA, 2016. ACM.
- [42] MICHAEL BACKES, SVEN BUGIEL, CHRISTIAN HAMMER, OLIVER SCHRANZ, AND PHILIPP VON STYP-REKOWSKY. Boxify: Full-fledged App Sandboxing for Stock Android. In *24th USENIX Security Symposium (USENIX Security 15)*, August 2015.
- [43] MICHAEL BACKES, SEBASTIAN GERLING, CHRISTIAN HAMMER, MATTEO MAFFEI, AND PHILIPP VON STYP-REKOWSKY. AppGuard: Enforcing User Requirements on Android Apps. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2013.
- [44] GABRIELE BAVOTA, BERNARDINO DE CARLUCCIO, ANDREA DE LUCIA, MASSIMILIANO DI PENTA, ROCCO OLIVETO, AND ORAZIO STROLLO. When Does a Refactoring Induce Bugs? An Empirical Study. In *Proceedings of the 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, SCAM '12, pages 104–113, Washington, DC, USA, 2012. IEEE Computer Society.
- [45] BCX509ExtendedTrustManager (Bouncy Castle Library 1.66 API Specification), June 2020. <https://www.bouncycastle.org/docs/tlsdocs1.5on/org/bouncycastle/jsse/BCX509ExtendedTrustManager.html>.
- [46] JGDMS/FilterX509TrustManager.java at pfirmstone/JGDMS, June 2020. <https://github.com/pfirmstone/JGDMS/blob/>

a44b96809783199b5fd69ffb803e0c4ceb9fad67/JGDMS/jgdms-jeri/
src/main/java/net/jini/jeri/ssl/FilterX509TrustManager.java.

- [47] AL BESSEY, KEN BLOCK, BEN CHELF, ANDY CHOU, BRYAN FULTON, SETH HALLEM, CHARLES HENRI-GROS, ASYA KAMSKY, SCOTT MCPeAK, AND DAWSON ENGLER. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World. *Communications of the ACM*, 53(2):66–75, February 2010.
- [48] FARZANA AHAMED BHUIYAN, AKOND RAHMAN, AND PATRICK MORRISON. Vulnerability discovery strategies used in software projects. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, ASE '20*, page 13–18, New York, NY, USA, 2021. Association for Computing Machinery.
- [49] TEGAWENDE F. BISSYANDE, DAVID LO, LINGXIAO JIANG, LAURENT REVEILLERE, JACQUES KLEIN, AND YVES LE TRAON. Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 188–197, Pasadena, CA, USA, November 2013. IEEE.
- [50] RICHARD BONETT, KAUSHAL KAFLE, KEVIN MORAN, ADWAIT NADKARNI, AND DENYS POSHYVANYK. Discovering flaws in security-focused static analysis tools for android using systematic mutation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1263–1280, Baltimore, MD, 2018. USENIX Association.
- [51] RICHARD BONETT, KAUSHAL KAFLE, KEVIN MORAN, ADWAIT NADKARNI, AND DENYS POSHYVANYK. Discovering flaws in security-focused static analysis tools for android using systematic mutation. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1263–1280, Baltimore, MD, August 2018. USENIX Association.
- [52] HUDSON BORGES, RODRIGO BRITO, AND MARCO TULIO VALENTE. Beyond Textual Issues: Understanding the Usage and Impact of GitHub Reactions. In *Proceed-*

- ings of the XXXIII Brazilian Symposium on Software Engineering*, pages 397–406, Salvador Brazil, September 2019. ACM.
- [53] A. BRAGA AND R. DAHAB. Mining Cryptography Misuse in Online Forums. In *2016 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 143–150. IEEE, August 2016.
- [54] ALEXANDRE BRAGA, RICARDO DAHAB, NUNO ANTUNES, NUNO LARANJEIRO, AND MARCO VIEIRA. Practical Evaluation of Static Analysis Tools for Cryptography: Benchmarking Method and Case Study. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 170–181. IEEE, October 2017.
- [55] ALEXANDRE BRAGA, RICARDO DAHAB, NUNO ANTUNES, NUNO LARANJEIRO, AND MARCO VIEIRA. Understanding How to Use Static Analysis Tools for Detecting Cryptography Misuse in Software. *IEEE Transactions on Reliability*, 68(4):1384–1403, December 2019.
- [56] V. BRAUN AND V. CLARKE. *Thematic Analysis: A Practical Guide*. SAGE Publications, 2021.
- [57] SVEN BUGIEL, LUCAS DAVI, ALEXANDRA DMITRIENKO, THOMAS FISCHER, AHMAD-REZA SADEGHI, AND BHARGAVA SHASTRY. Toward Taming Privilege-Escalation Attacks on Android. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, 2012.
- [58] SVEN BUGIEL, LUCAS DAVI, ALEXANDRA DMITRIENKO, STEPHAN HEUSER, AHMAD-REZA SADEGHI, AND BHARGAVA SHASTRY. Practical and Lightweight Domain Isolation on Android. In *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM)*, 2011.
- [59] S. CALZAVARA, I. GRISHCHENKO, AND M. MAFFEI. HornDroid: Practical and

- Sound Static Analysis of Android Applications by SMT Solving. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 47–62, March 2016.
- [60] YINZHI CAO, YANICK FRATANTONIO, ANTONIO BIANCHI, MANUEL EGELE, CHRISTOPHER KRUEGEL, GIOVANNI VIGNA, AND YAN CHEN. EdgeMiner: Automatically Detecting Implicit Control Flow Transitions through the Android Framework. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, February 2015.
- [61] OSCAR CHAPARRO, CARLOS BERNAL-CÁRDENAS, JING LU, KEVIN MORAN, ANDRIAN MARCUS, MASSIMILIANO DI PENTA, DENYS POSHYVANYK, AND VINCENT NG. Assessing the quality of the steps to reproduce in bug reports. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 86–96, Tallinn Estonia, August 2019. ACM.
- [62] OSCAR CHAPARRO, JING LU, FIORELLA ZAMPETTI, LAURA MORENO, MASSIMILIANO DI PENTA, ANDRIAN MARCUS, GABRIELE BAVOTA, AND VINCENT NG. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 396–407, Paderborn Germany, August 2017. ACM.
- [63] TING-HAN CHEN, CARLOTTA TAGLIARO, MARTINA LINDORFER, KEVIN BORGOLTE, AND JEROEN VAN DER HAM-DE VOS. Are You Sure You Want To Do Coordinated Vulnerability Disclosure? In *2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 307–314, Vienna, Austria, July 2024. IEEE.
- [64] YIKANG CHEN, YIBO LIU, KA LOK WU, DUC V LE, AND SZE YIU CHAU. Towards Precise Reporting of Cryptographic Misuses. In *Proceedings 2024 Network*

- and Distributed System Security Symposium*, San Diego, CA, USA, 2024. Internet Society.
- [65] B. CHES AND G. MCGRAW. Static analysis for security. *IEEE Security Privacy*, 2(6):76–79, November 2004.
- [66] ERIKA CHIN, ADRIENNE PORTER FELT, KATE GREENWOOD, AND DAVID WAGNER. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2011.
- [67] MARIA CHRISTAKIS AND CHRISTIAN BIRD. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 332–343, Singapore Singapore, August 2016. ACM.
- [68] cloc - Count Lines of Code, June 2020. <https://github.com/AlDanial/cloc>.
- [69] Code Review Tool - Amazon CodeGuru Security - AWS, June 2024. <https://aws.amazon.com/codeguru/>.
- [70] Codiga: Static Code Analysis in Real-Time, June 2024. <https://www.codiga.io/static-code-analysis/>.
- [71] CogniCrypt - Secure Integration of Cryptographic Software — CogniCrypt, June 2020. <https://www.eclipse.org/cognicrypt/>.
- [72] MAURO CONTI, VU THIEN NGA NGUYEN, AND BRUNO CRISPO. CRePE: Context-Related Policy Enforcement for Android. In *Proceedings of the 13th Information Security Conference (ISC)*, October 2010.
- [73] JULIET M. CORBIN AND ANSELM STRAUSS. Grounded theory research: Procedures, canons, and evaluative criteria. *Qualitative Sociology*, 13(1):3–21, 1990.

- [74] C. CORBRIDGE, G. RUGG, N. P. MAJOR, N. R. SHADBOLT, AND A. M. BURTON. Laddering: Technique and tool use in knowledge acquisition. *Knowledge Acquisition*, 6(3):315–341, September 1994.
- [75] NSF Award Search: Award#1929701 - SaTC: TTP: Medium: Collaborative: Deployment-quality and Accessible Solutions for Cryptography Code Development, May 2020. https://www.nsf.gov/awardsearch/showAward?AWD_ID=1929701&HistoricalAwards=false.
- [76] Oracle - Industrial Experience of Finding Cryptographic Vulnerabilities in Large-scale Codebases, July 2020. https://labs.oracle.com/pls/apex/f?p=94065:40150:0::::P40150_PUBLICATION_ID:6629.
- [77] DeepSource: The Code Health Platform, June 2024. <https://deepsource.com/>.
- [78] R. A. DEMILLO, R. J. LIPTON, AND F. G. SAYWARD. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, April 1978.
- [79] LIN DENG, N. MIRZAEI, P. AMMANN, AND J. OFFUTT. Towards mutation analysis of android apps. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–10, April 2015.
- [80] ANNA DEREZIŃSKA AND KONRAD HALAS. *Analysis of Mutation Operators for the Python Language*, pages 155–164. Springer International Publishing, Cham, 2014.
- [81] ANDROID DEVELOPERS. Android developer documentation - broadcasts, mar 2019.
- [82] ANDROID DEVELOPERS. Android developer documentation - intents and intent filters, mar 2019.
- [83] ANDROID DEVELOPERS. Android developer documentation - the activity lifecycle, mar 2019.

- [84] ANDROID DEVELOPERS. Enable multidex for apps with over 64K methods, May 2020. <https://developer.android.com/studio/build/multidex>.
- [85] DANIEL DI NARDO, FABRIZIO PASTORE, AND LIONEL C. BRIAND. Generating complex and faulty test data through model-based mutation analysis. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–10, 2015.
- [86] MICHAEL DIETZ, SHASHI SHEKHAR, YULIY PISETSKY, ANHEI SHU, AND DAN S. WALLACH. Quire: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the USENIX Security Symposium*, August 2011.
- [87] DINO DISTEFANO, MANUEL FÄHNDRICH, FRANCESCO LOGOZZO, AND PETER W. O’HEARN. Scaling static analyses at Facebook. *Communications of the ACM*, 62(8):62–70, July 2019.
- [88] DAVID DITTRICH, ERIN KENNEALLY, AND MICHAEL BAILEY. Applying Ethical Principles to Information and Communication Technology Research: A Companion to the Menlo Report. *SSRN Electronic Journal*, 2013.
- [89] BRENDAN DOLAN-GAVITT, PATRICK HULIN, ENGIN KIRDA, TIM LEEK, ANDREA MAMBRETTI, WIL ROBERTSON, FREDERICK ULRICH, AND RYAN WHELAN. Lava: Large-scale automated vulnerability addition. In *Proceedings of the 37th IEEE Symposium on Security and Privacy (S&P)*, may 2016.
- [90] HAN DORUSSEN, HARTMUT LENZ, AND SPYROS BLAVOUKOS. Assessing the Reliability and Validity of Expert Interviews. *European Union Politics*, 6(3):315–337, September 2005.
- [91] DroidBench 2.0. <https://github.com/secure-software-engineering/DroidBench>. Last accessed on June 27, 2020.

- [92] THE ECONOMIST. Planet of the phones. <http://www.economist.com/news/leaders/21645180-smartphone-ubiquitous-addictive-and-transformative-plan> February 2015.
- [93] 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 - CCS '13*, pages 73–84, Berlin, Germany, 2013. ACM Press.
- [94] MANUEL EGELE, CHRISTOPHER KRUEGEL, ENGIN KIRDA, AND GIOVANNI VIGNA. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, February 2011.
- [95] PÄR EMANUELSSON AND ULF NILSSON. A Comparative Study of Industrial Static Analysis Tools. *Electronic Notes in Theoretical Computer Science*, 217:5–21, July 2008.
- [96] WILLIAM ENCK, PETER GILBERT, BYUNG-GON CHUN, LANDON P. COX, JAEYEON JUNG, PATRICK MCDANIEL, AND ANMOL N. SHETH. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.
- [97] WILLIAM ENCK, MACHIGAR ONGTANG, AND PATRICK MCDANIEL. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, November 2009.
- [98] MADELINE ENDRES, KEVIN BOEHNKE, AND WESTLEY WEIMER. Hashing it out: A survey of programmers’ cannabis usage, perception, and motivation. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1107–1119, Pittsburgh Pennsylvania, May 2022. ACM.

- [99] ÇAĞRI EREN, KEREM ŞAHİN, AND ERAY TÜZÜN. Analyzing Bug Life Cycles to Derive Practical Insights. In *Proceedings of the 27th International Conference on Evaluation and Assessment in Software Engineering*, pages 162–171, Oulu Finland, June 2023. ACM.
- [100] 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, CCS '12*, page 50–61, New York, NY, USA, 2012. Association for Computing Machinery.
- [101] SASCHA FAHL, MARIAN HARBACH, HENNING PERL, MARKUS KOETTER, AND MATTHEW SMITH. Rethinking ssl development in an appified world. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, CCS '13*, pages 49–60, New York, NY, USA, 2013. ACM.
- [102] F-Droid - Free and Open Source Android App Repository, June 2020. <https://f-droid.org/>.
- [103] ADRIENNE PORTER FELT, ERIKA CHIN, STEVE HANNA, DAWN SONG, AND DAVID WAGNER. Android Permissions Demystified. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2011.
- [104] ADRIENNE PORTER FELT, HELEN J. WANG, ALEXANDER MOSHCHUK, STEVEN HANNA, AND ERIKA CHIN. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the USENIX Security Symposium*, August 2011.
- [105] XINMING OU FENGGUO WEI, SANKARDAS ROY AND ROBBY. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2014.

- [106] FELIX FISCHER, KONSTANTIN BOTTINGER, 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, San Jose, CA, USA, May 2017. IEEE.
- [107] FORTINET. SolarWinds Supply Chain Attack — Fortinet. <https://www.fortinet.com/resources/cyberglossary/solarwinds-cyber-attack>.
- [108] THE APACHE SOFTWARE FOUNDATION. Apache ant build system, 2019.
- [109] ELLI FRAGKAKI, LUJO BAUER, LIMIN JIA, AND DAVID SWASEY. Modeling and enhancing android’s permission system. In *Computer Security – ESORICS 2012*, Sara Foresti, Moti Yung, and Fabio Martinelli, editors, pages 1–18, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
- [110] JASON FRANKLIN, SAGAR CHAKI, ANUPAM DATTA, AND ARVIND SESHADRI. Scalable parametric verification of secure systems: How to verify reference monitors without worrying about data structure size. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 365–379, 2010.
- [111] CHRISTIAN FRITZ, STEVEN ARZT, SIEGFRIED RASTHOFER, ERIC BODDEN, ALEXANDRE BARTEL, JACQUES KLEIN, YVES LE TRAON, DAMIEN OCTEAU, AND PATRICK MCDANIEL. Highly precise taint analysis for android applications. Technical Report TUD-CS-2013-0113, EC SPRIDE, May 2013.
- [112] KELSEY R. FULTON, ANNA CHAN, DANIEL VOTIPKA, MICHAEL HICKS, AND MICHELLE L. MAZUREK. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, pages 597–616. USENIX Association, August 2021.
- [113] TOM GANZ, ERIK IMGRUND, MARTIN HÄRTERICH, AND KONRAD RIECK. Pavudi:

- Patch-based vulnerability discovery using machine learning. In *Proceedings of the 39th Annual Computer Security Applications Conference, ACSAC '23*, page 704–717, New York, NY, USA, 2023. Association for Computing Machinery.
- [114] 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, Montreal, QC, Canada, May 2019. IEEE.
- [115] CLINT GIBLER, JON CRUSSELL, JEREMY ERICKSON, AND HAO CHEN. Androi-dLeaks: Automatically Detecting Potential Privacy Leaks In Android Applications on a Large Scale. In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST)*, June 2012.
- [116] GITHUB. Java: FP in java/integer-multiplication-cast-to-long with integer of max size - Issue #1062 - github/codeql. <https://github.com/github/codeql/issues/1062>.
- [117] GITHUB. Labels - github/codeql. <https://github.com/github/codeql/labels?page=1&sort=name-asc>.
- [118] GITHUB. Software Bill of Materials (SBOM) — CISA . <https://www.cisa.gov/sbom>.
- [119] About code scanning - github docs, nov 2020. <https://docs.github.com/en/free-pro-team@latest/github/finding-security-vulnerabilities-and-errors-in-your-code/about-code-scanning>.
- [120] Announcing third-party code scanning tools: static analysis & developer security training - The GitHub Blog, November 2020. <https://github.blog/2020-10-05-announcing-third-party-code-scanning-tools-static-analysis-an>

- [121] Apache/ignite. The Apache Software Foundation, November 2020. <https://github.com/apache/ignite>.
- [122] Azure/azure-sdk-for-java: This repository is for active development of the azure sdk for java, November 2020. <https://github.com/Azure/azure-sdk-for-java>.
- [123] Build Software Better, Together - Github, June 2020. <https://github.com>.
- [124] codeql/Encryption.qll at github/codeql, November 2020. <https://github.com/github/codeql/blob/768e5190a1c9d40a4acc7143c461c3b114e7fd59/java/ql/src/semmler/code/java/security/Encryption.qll#L142>.
- [125] codeql/UnsafeCertTrust.qll at github/codeql, November 2020. <https://github.com/github/codeql/blob/768e5190a1c9d40a4acc7143c461c3b114e7fd59/java/ql/src/experimental/Security/CWE/CWE-273/UnsafeCertTrust.qll>.
- [126] encryption-machine/asymencrptmachine.java at mrdrivingduck/encryption-machine, nov 2020. <https://github.com/mrdrivingduck/encryption-machine/blob/74c5679c86cf11f74409cfc63e1385b906734099/src/iot/zjt/encrypt/machine/AsymEncrptMachine.java#L95>.
- [127] ExoPlayer/CachedContentIndex.java at google/ExoPlayer, nov 2020. <https://github.com/google/ExoPlayer/blob/f182c0c1169cba7c22280058368127c24609054f/library/core/src/main/java/com/google/android/exoplayer2/upstream/cache/CachedContentIndex.java#L332>.
- [128] GitHub - Where software is built, November 2020. <https://github.com/search/advanced>.
- [129] GitHub Security Lab, November 2020. <https://securitylab.github.com/>.

- [130] hive/genericudfaesbase.java at master - apache/hive, November 2020. <https://github.com/apache/hive/blob/526bd87e9103375f0ddb8064dcfd8c31342b4c08/ql/src/java/org/apache/hadoop/hive/ql/udf/generic/GenericUDFAesBase.java#L108>.
- [131] ignite/igniteutils.java at apache/ignite, November 2020. <https://github.com/apache/ignite/blob/1a3fd112b02133892c7c95d4be607079ffa83211/modules/core/src/main/java/org/apache/ignite/internal/util/IgnoreUtils.java#L11714>.
- [132] Java: CWE-273 Unsafe certificate trust by luchua-bc · Pull Request #3550 - github/codeql, November 2020. <https://github.com/github/codeql/pull/3550>.
- [133] jmeter/trustallsslsocketfactory.java at apache/jmeter, nov 2020. <https://github.com/apache/jmeter/blob/704adb91f7f967402b9b709e89f5b73f0a466283/src/core/src/main/java/org/apache/jmeter/util/TrustAllSSLSocketFactory.java>.
- [134] pdf-service/md5util.java at elainrd/pdf-servic, nov 2020. <https://github.com/elainrd/pdf-service/blob/243588e446ed875e13a99ea08e2baa3e0806c346/src/main/java/com/hhd/pdf/util/Md5Util.java#L76>.
- [135] LEO A. GOODMAN. Snowball Sampling. *The Annals of Mathematical Statistics*, 32(1):148–170, 1961.
- [136] PETER LEO GORSKI, YASEMIN ACAR, LUIGI LO IACONO, AND SASCHA FAHL. Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–13, Honolulu HI USA, April 2020. ACM.

- [137] PETER LEO GORSKI, LUIGI LO IACONO, DOMINIK WERMKE, CHRISTIAN STRANSKY, SEBASTIAN MÖLLER, YASEMIN ACAR, AND SASCHA FAHL. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018, Baltimore, MD, USA, August 12-14, 2018*, Mary Ellen Zurko and Heather Richter Lipford, editors, pages 265–281. USENIX Association, 2018.
- [138] MICHAEL GRACE, YAJIN ZHOU, QIANG ZHANG, SHIHONG ZOU, AND XUXIAN JIANG. RiskRanker: Scalable and Accurate Zero-day Android Malware Detection. In *Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys)*, 2012.
- [139] MATTHEW GREEN AND MATTHEW SMITH. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Security & Privacy*, 14(5):40–46, September 2016.
- [140] MARCO GUTFLEISCH, JAN H. KLEMMER, NIKLAS BUSCH, YASEMIN ACAR, M. ANGELA SASSE, AND SASCHA FAHL. How Does Usable Security (Not) End Up in Software Products? Results From a Qualitative Interview Study. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 893–910, May 2022.
- [141] WOLF HALTON, BO WEAVER, JUNED ANSARI, SRINIVASA KOTIPALLI, AND MOHAMMED A IMRAN. Penetration testing: A survival guide. In *Penetration Testing: A Survival Guide*. Packt Publishing, first edition, 2017. 4. Overview of Attacking Android Apps - QARK (Quick Android Review Kit).
- [142] R. G. HAMLET. Testing programs with the aid of a compiler. *IEEE Trans. Software Eng.*, 3(4):279–290, July 1977.
- [143] MOHAMMADREZA HAZHIRPASAND, MOHAMMAD GHAFARI, STEFAN KRÜGER, ERIC BODDEN, AND OSCAR NIERSTRASZ. The Impact of Developer Experience

- in Using Java Cryptography. In *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, September 2019.
- [144] STEPHAN HEUSER, ADWAIT NADKARNI, WILLIAM ENCK, AND AHMAD-REZA SADEGHI. ASM: A Programmable Interface for Extending Android Security. In *Proceedings of the USENIX Security Symposium*, August 2014.
- [145] TSUNG-HSUAN HO, DANIEL DEAN, XIAOHUI GU, AND WILLIAM ENCK. PREC: Practical Root Exploit Containment for Android Devices. In *Proceedings of the Fourth ACM Conference on Data and Application Security and Privacy (CODASPY)*, March 2014.
- [146] S. HOLAVANALLI, D. MANUEL, V. NANJUNDASWAMY, B. ROSENBERG, F. SHEN, S. Y. KO, AND L. ZIAREK. Flow permissions for android. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 652–657, Nov 2013.
- [147] THE WHITE HOUSE. Executive Order on Improving the Nation’s Cybersecurity. <https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/>, May 2021.
- [148] ALLEN D HOUSEHOLDER, GARRET WASSERMANN, ART MANION, AND CHRIS KING. The CERT Guide to Coordinated Vulnerability Disclosure.
- [149] S.E. HOVE AND B. ANDA. Experiences from Conducting Semi-Structured Interviews in Empirical Software Engineering Research. In *11th IEEE International Software Metrics Symposium (METRICS’05)*, pages 10 pp.–23, 2005.
- [150] TONY HSIANG-CHIH HSU. Practical security automation and testing: Tools and techniques for automated security scanning and testing in DevSecOps. In *Practical*

- Security Automation and Testing*. Packt Publishing, Limited, Birmingham, 2019.
- Android Security Testing - Static secure code scanning with QARK.
- [151] HU, JANE C. How one lightbulb could allow hackers to burgle your home. <https://qz.com/1493748/how-one-lightbulb-could-allow-hackers-to-burgle-your-home/> amp/, Accessed June 2019.
- [152] WEI HUANG, YAO DONG, ANA MILANOVA, AND JULIAN DOLBY. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 106–117, Baltimore MD USA, July 2015. ACM.
- [153] WEI HUANG, YAO DONG, ANA MILANOVA, AND JULIAN DOLBY. Scalable and precise taint analysis for Android. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis - ISSA 2015*, pages 106–117, Baltimore, MD, USA, 2015. ACM Press.
- [154] ICC-Bench. <https://github.com/fgwei/ICC-Bench>. Last accessed on June 27, 2020.
- [155] ERIK IMGRUND, TOM GANZ, MARTIN HÄRTERICH, LUKAS PIRCH, NIKLAS RISSE, AND KONRAD RIECK. Broken promises: Measuring confounding effects in learning-based vulnerability discovery. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security, AISec '23*, page 149–160, New York, NY, USA, 2023. Association for Computing Machinery.
- [156] NASIF IMTIAZ, AKOND RAHMAN, EFFAT FARHANA, AND LAURIE WILLIAMS. Challenges with Responding to Static Analysis Tool Alerts. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 245–249, May 2019.

- [157] GARTNER INC. Application Security Testing Reviews 2023 — Gartner Peer Insights. <https://www.gartner.com/market/application-security-testing>, 2023.
- [158] GRADLE INC. Gradle build system, 2019.
- [159] H.r.1668 - 116th congress (2019-2020): Internet of things cybersecurity improvement act of 2020 — congress.gov — library of congress, dec 2020. (Accessed on 07/21/2021).
- [160] IoT Security Rating — Identity Management & Security, April 2021. <https://ims.ul.com/IoT-security-rating>.
- [161] ioXt - The Global Standard for IoT Security. <https://www.ioxtalliance.org>, March 2023.
- [162] REYHANEH JABBARVAND AND SAM MALEK. An energy-aware mutation testing framework for android. In *FSE'17*, 2017.
- [163] KONRAD JAMROZIK, PHILIPP VON STYP-REKOWSKY, AND ANDREAS ZELLER. Mining sandboxes. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 37–48, May 2016.
- [164] JAN JANCAR, MARCEL FOURNÉ, DANIEL DE ALMEIDA BRAGA, MOHAMED SABT, PETER SCHWABE, GILLES BARTHE, PIERRE-ALAIN FOUQUE, AND YASEMIN ACAR. “They’re not that hard to mitigate”: What Cryptographic Library Developers Think About Timing Attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 632–649, May 2022.
- [165] JINSEONG JEON, KRISTOPHER K. MICINSKI, JEFFREY A. VAUGHAN, ARI FOGEL, NIKHILESH REDDY, JEFFREY S. FOSTER, AND TODD MILLSTEIN. Dr. Android and Mr. Hide: Fine-grained Permissions in Android Applications. In *Proceedings of the ACM Workshop on Security and Privacy in Mobile Devices (SPSM)*, 2012.

- [166] LIMIN JIA, JASSIM ALJURAIDAN, ELLI FRAGKAKI, LUJO BAUER, MICHAEL STROUCKEN, KAZUhide FUKUSHIMA, SHINSAKU KIYOMOTO, AND YUTAKA MIYAKE. Run-Time Enforcement of Information-Flow Properties on Android (Extended Abstract). In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, September 2013.
- [167] BRITTANY JOHNSON, YOONKI SONG, EMERSON MURPHY-HILL, AND ROBERT BOWDIDGE. Why Don't Software Developers Use Static Analysis Tools to Find Bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681, San Francisco, CA, USA, May 2013. IEEE.
- [168] JAEYEON JUNG, ANMOL SHETH, BEN GREENSTEIN, DAVID WETHERALL, GABRIEL MAGANIS, AND TADAYOSHI KOHNO. Privacy Oracle: A System for Finding Application Leaks with Black Box Differential Testing. In *Proceedings of the 15th ACM conference on Computer and Communications Security*, pages 279–288. ACM New York, NY, USA, 2008.
- [169] KAUSHAL KAFLE, KEVIN MORAN, SUNIL MANANDHAR, ADWAIT NADKARNI, AND DENYS POSHYVANYK. A Study of Data Store-based Home Automation. In *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY)*, March 2019.
- [170] ERIN KENNEALLY AND DAVID DITTRICH. The Menlo Report: Ethical Principles Guiding Information and Communication Technology Research. *SSRN Electronic Journal*, 2012.
- [171] BARBARA KITCHENHAM, O. PEARL BRERETON, DAVID BUDGEN, MARK TURNER, JOHN BAILEY, AND STEPHEN LINKMAN. Systematic literature reviews in software engineering – A systematic literature review. *Information and Software Technology*, 51(1):7–15, 2009.

- [172] WILLIAM KLIEBER, LORI FLYNN, AMAR BHOSALE, LIMIN JIA, AND LUJO BAUER. Android Taint Flow Analysis for App Sets. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, 2014.
- [173] WILLIAM KLIEBER, LORI FLYNN, WILL SNAVELY, AND MICHAEL ZHENG. Practical Precise Taint-flow Static Analysis for Android App Sets. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–7, Hamburg Germany, August 2018. ACM.
- [174] SRINIVASA RAO KOTIPALLI AND MOHAMMED A IMRAN. Hacking android. In *Hacking Android*. Packt Publishing, Limited, Birmingham, first edition, 2016. 4. Overview of Attacking Android Apps - QARK(Quick Android Review Kit).
- [175] STEFAN KRÜGER, SARAH NADI, MICHAEL REIF, KARIM ALI, MIRA MEZINI, ERIC BODDEN, FLORIAN GÖPFERT, FELIX GÜNTHER, CHRISTIAN WEINERT, DANIEL DEMMLER, AND RAM KAMATH. CogniCrypt: Supporting Developers in Using Cryptography. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017*, pages 931–936, Piscataway, NJ, USA, 2017. IEEE Press.
- [176] 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)*, Todd Millstein, editor, volume 109 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 10:1–10:27. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2018.
- [177] YOUN KYU LEE, JAE YOUNG BANG, GHOLAMREZA SAFI, ARMAN SHAHBAZIAN, YIXUE ZHAO, AND NENAD MEDVIDOVIC. A sealant for inter-app security holes in

- android. In *Proceedings of the 39th International Conference on Software Engineering*, pages 312–323, May 2017.
- [178] LGTM - Continuous security analysis, November 2020. <https://lgtm.com/>.
- [179] L. LI, A. BARTEL, J. KLEIN, AND Y. L. TRAON. Automatically exploiting potential component leaks in android applications. In *2014 IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 388–397, Sept 2014.
- [180] LI LI, ALEXANDRE BARTEL, TEGAWENDÉ F. BISSYANDÉ, JACQUES KLEIN, YVES LE TRAON, STEVEN ARZT, SIEGFRIED RASTHOFER, ERIC BODDEN, DAMIEN OCTEAU, AND PATRICK MCDANIEL. Iccta: Detecting inter-component privacy leaks in android apps. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, pages 280–291, 2015.
- [181] LI LI, ALEXANDRE BARTEL, JACQUES KLEIN, YVES LE TRAON, STEVEN ARZT, SIEGFRIED RASTHOFER, ERIC BODDEN, DAMIEN OCTEAU, AND PATRICK MCDANIEL. I Know What Leaked in Your Pocket: Uncovering Privacy Leaks on Android Apps with Static Taint Analysis. In *CoRR*, 2014.
- [182] M. LILLACK, C. KASTNER, AND E. BODDEN. Tracking load-time configuration options. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2017.
- [183] GUANJUN LIN, JUN ZHANG, WEI LUO, LEI PAN, AND YANG XIANG. Poster: Vulnerability discovery with function representation learning from unlabeled projects. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, page 2539–2541, New York, NY, USA, 2017. Association for Computing Machinery.
- [184] MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, MICHELE TUFANO, KEVIN MORAN, MASSIMILIANO DI PENTA, CHRISTOPHER VENDOME, CARLOS BERNAL-

- CÁRDENAS, AND DENYS POSHYVANYK. Enabling mutation testing for android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 233–244, New York, NY, USA, 2017. ACM.
- [185] MARTINA LINDORFER, MATTHIAS NEUGSCHWANDTNER, LUKAS WEICHSELBAUM, YANICK FRATANTONIO, VICTOR VAN DER VEEN, AND CHRISTIAN PLATZER. Andrubiś—1,000,000 apps later: A view on current Android malware behaviors. In *Third International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.
- [186] Linkedin/Qark, May 2020. <https://engineering.linkedin.com/blog/2015/08/introducing-qark>.
- [187] ANDREA LISI, PRATEETI MUKHERJEE, LAURA DE SANTIS, LEI WU, DMITRIJ LAGUTIN, AND YKI KORTESNIEMI. Automated responsible disclosure of security vulnerabilities. *IEEE Access*, 10:10472–10489, 2022.
- [188] BIN LIU, BIN LIU, HONGXIA JIN, AND RAMESH GOVINDAN. Efficient privilege de-escalation for ad libraries in mobile apps. In *Proceedings of the 13th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys ’15, pages 89–103, New York, NY, USA, 2015. ACM.
- [189] SHUHAN LIU, JIAYUAN ZHOU, XING HU, FILIPE ROSEIRO COGO, XIN XIA, AND XIAOHU YANG. An empirical study on vulnerability disclosure management of open source software systems. *ACM Transactions on Software Engineering and Methodology*, page 3716822, March 2025.
- [190] BENJAMIN LIVSHITS, DIMITRIOS VARDOLAKIS, MANU SRIDHARAN, YANNIS SMARAGDAKIS, ONDŘEJ LHOTÁK, J. NELSON AMARAL, BOR-YUH EVAN CHANG, SAMUEL Z. GUYER, UDAY P. KHEDKER, AND ANDERS MØLLER. In defense of soundness: A manifesto. *Communications of the ACM*, 58(2):44–46, January 2015.

- [191] LONG LU, ZHICHUN LI, ZHENYU WU, WENKE LEE, AND GUOFEI JIANG. CHEX: Statically Vetting Android Apps for Component Hijacking Vulnerabilities. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 229–240, 2012.
- [192] LINGHUI LUO, FELIX PAUCK, GORAN PISKACHEV, MANUEL BENZ, IVAN PASHCHENKO, MARTIN MORY, ERIC BODDEN, BEN HERMANN, AND FABIO MASSACCI. TaintBench: Automatic real-world malware benchmarking of Android taint analyses. *Empirical Software Engineering*, 27(1):16, January 2022.
- [193] YU-SEUNG MA, YONG RAE KWON, AND JEFF OFFUTT. Inter-class mutation operators for java. In *13th International Symposium on Software Reliability Engineering (ISSRE 2002), 12-15 November 2002, Annapolis, MD, USA*, pages 352–366, 2002.
- [194] PRIANKA MANDAL, AMIT SEAL AMI, VICTOR OLAIYA, SAYYED HADI RAZMJO, AND ADWAIT NADKARNI. “Belt and suspenders” or “just red tape”? Investigating Early Artifacts and User Perceptions of IoT App Security Certification. In *Proceedings of the 2024 USENIX Security Symposium (USENIX)*, August 2024.
- [195] PRIANKA MANDAL, AMIT SEAL AMI, VICTOR OLAIYA, SAYYED HADI RAZMJO, AND ADWAIT NADKARNI. “Belt and suspenders” or “just red tape”? Investigating Early Outcomes and Perceptions of IoT Security Compliance Enforcement. In *Proceedings of the 2024 USENIX Security Symposium (USENIX)*, August 2024. To appear.
- [196] KE MAO, MARK HARMAN, AND YUE JIA. Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016*, pages 94–105, New York, NY, USA, 2016. ACM.
- [197] MATTHEW B. MILES, A. M. HUBERMAN, AND JOHNNY SALDAÑA. *Qualitative*

- Data Analysis: A Methods Sourcebook*. SAGE Publications, Inc, Thousand Oaks, California, 3rd edition edition, 2014.
- [198] JOYDEEP MITRA AND VENKATESH-PRASAD RANGANATH. Ghera: A Repository of Android App Vulnerability Benchmarks. In *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, pages 43–52, Toronto Canada, November 2017. ACM.
- [199] K. MORAN, M. LINARES-VASQUEZ, C. BERNAL-CARDENAS, C. VENDOME, AND D. POSHYVANYK. Crashscope: A practical tool for automated testing of android applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*, pages 15–18, May 2017.
- [200] KEVIN MORAN, MICHELE TUFANO, CARLOS BERNAL-CÁRDENAS, MARIO LINARES-VÁSQUEZ, GABRIELE BAVOTA, CHRISTOPHER VENDOME, MASSIMILIANO DI PENTA, AND DENYS POSHYVANYK. Mdroid+: a mutation testing framework for android. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, ICSE '18*, page 33–36, New York, NY, USA, 2018. Association for Computing Machinery.
- [201] KEVIN MORAN, MARIO LINARES VÁSQUEZ, CARLOS BERNAL-CÁRDENAS, CHRISTOPHER VENDOME, AND DENYS POSHYVANYK. Automatically discovering, reporting and reproducing android application crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*, pages 33–44, 2016.
- [202] GIOVANE C. M. MOURA AND JOHN HEIDEMANN. Vulnerability Disclosure Considered Stressful. *ACM SIGCOMM Computer Communication Review*, 53(2):2–10, April 2023.
- [203] MSE DEVELOPERS. mse sources and data., mar 2019.

- [204] ANDREW C. MYERS. JFlow: Practical Mostly-Static Information Flow Control. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 1999.
- [205] ANDREW C. MYERS AND BARBARA LISKOV. Protecting Privacy Using the Decentralized Label Model. *ACM Transactions on Software Engineering and Methodology*, 9(4):410–442, October 2000.
- [206] MARCUS NACHTIGALL, MICHAEL SCHLICHTIG, AND ERIC BODDEN. A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 532–543, Virtual South Korea, July 2022. ACM.
- [207] SARAH NADI, STEFAN KRÜGER, MIRA MEZINI, AND ERIC BODDEN. Jumping Through Hoops: Why Do Java Developers Struggle with Cryptography APIs? In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 935–946, New York, NY, USA, 2016. ACM.
- [208] ADWAIT NADKARNI, BENJAMIN ANDOW, WILLIAM ENCK, AND SOMESH JHA. Practical DIFC Enforcement on Android. In *Proceedings of the 25th USENIX Security Symposium*, August 2016.
- [209] ADWAIT NADKARNI AND WILLIAM ENCK. Preventing Accidental Data Disclosure in Modern Operating Systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, November 2013.
- [210] YUHONG NAN, MIN YANG, ZHEMIN YANG, SHUNFAN ZHOU, GUOFEI GU, AND XIAOFENG WANG. Uipicker: User-input privacy identification in mobile applications. In *USENIX Security Symposium*, pages 993–1008, 2015.
- [211] NATIONAL INSTITUTE OF TECHNOLOGY (NIT) PUDUCHERRY, KARAİKAL, INDIA, KEERTHI VASAN K., AND ARUN RAJ KUMAR P. Taxonomy of SSL/TLS Attacks.

- International Journal of Computer Network and Information Security*, 8(2):15–24, February 2016.
- [212] MOHAMMAD NAUMAN, SOHAIL KHAN, AND XINWEN ZHANG. Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2010.
- [213] DUC CUONG NGUYEN, DOMINIK WERMKE, YASEMIN ACAR, MICHAEL BACKES, CHARLES WEIR, AND SASCHA FAHL. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17*, pages 1065–1077, New York, NY, USA, October 2017. Association for Computing Machinery.
- [214] LISA NGUYEN QUANG DO, JAMES WRIGHT, AND KARIM ALI. Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations. *IEEE Transactions on Software Engineering*, 48(3):835–847, 2020.
- [215] NPR. Ransomware attack led to harrowing lapses at Ascension hospitals, staffers say : NPR. <https://www.npr.org/2024/06/19/nx-s1-5010219/ascension-hospital-ransomware-attack-care-lapses>.
- [216] OASIS. The Static Analysis Results Interchange Format (SARIF). <https://sarifweb.azurewebsites.net/>.
- [217] D. OCTEAU, S. JHA, AND P. MCDANIEL. Retargeting Android applications to java bytecode. In *ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012.
- [218] DAMIEN OCTEAU, DANIEL LUCHAUP, MATTHEW DERING, SOMESH JHA, AND PATRICK MCDANIEL. Composite constant propagation: Application to android inter-component communication analysis. In *Proceedings of the 37th International*

- Conference on Software Engineering - Volume 1*, ICSE '15, pages 77–88, Piscataway, NJ, USA, 2015. IEEE Press.
- [219] DAMIEN OCTEAU, PATRICK MCDANIEL, SOMESH JHA, ALEXANDRE BARTEL, ERIC BODDEN, JACQUES KLEIN, AND YVES LE TRAON. Effective Inter-component Communication Mapping in Android with EPPIC: An Essential Step Towards Holistic Security Analysis. In *Proceedings of the USENIX Security Symposium*, 2013.
- [220] DAMIEN OCTEAU, PATRICK MCDANIEL, SOMESH JHA, ALEXANDRE BARTEL, ERIC BODDEN, JACQUES KLEIN, AND YVES LE TRAON. Effective inter-component communication mapping in android: An essential step towards holistic security analysis. In *Presented as Part of the 22nd USENIX Security Symposium (USENIX Security 13)*, pages 543–558. USENIX, 2013.
- [221] DAMIEN OCTEAU, PATRICK MCDANIEL, SOMESH JHA, ALEXANDRE BARTEL, ERIC BODDEN, AND YVES LE TRAON. Effective Inter-Component Communication Mapping in Android with Epicc: An Essential Step Towards Holistic Security Analysis.
- [222] U. S. GOVERNMENT ACCOUNTABILITY OFFICE. SolarWinds Cyberattack Demands Significant Federal and Private-Sector Response (infographic) — U.S. GAO. <https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic>.
- [223] U. S. GOVERNMENT ACCOUNTABILITY OFFICE. SolarWinds Cyberattack Demands Significant Federal and Private-Sector Response (infographic) — U.S. GAO. <https://www.gao.gov/blog/solarwinds-cyberattack-demands-significant-federal-and-private-sector-response-infographic>.
- [224] A. JEFFERSON OFFUTT AND ROLAND H. UNTCH. *Mutation 2000: Uniting the Orthogonal*, pages 34–44. Springer US, Boston, MA, 2001.

- [225] R. A. P. OLIVEIRA, E. ALÉGROTH, Z. GAO, AND A. MEMON. Definition and evaluation of mutation operators for GUI-level mutation analysis. In *International Conference on Software Testing, Verification, and Validation - Workshops, ICSTW'15*, pages 1–10, 2015.
- [226] MARTEN OLTROGGE, NICOLAS HUAMAN, SABRINA AMFT, YASEMIN ACAR, MICHAEL BACKES, AND SASCHA FAHL. Why Eve and Mallory Still Love Android: Revisiting TLS (In) Security in Android Applications. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.
- [227] MACHIGAR ONGTANG, STEPHEN MCCLAUGHLIN, WILLIAM ENCK, AND PATRICK MCDANIEL. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 25th Annual Computer Security Applications Conference (ACSAC)*, pages 340–349, December 2009.
- [228] False negatives kill. <https://github.com/Secure-Platforms-Lab-W-M/false-negatives-kill>, April 2023.
- [229] Java Secure Socket Extension (JSSE) Reference Guide - Oracle Help Center, November 2020. <https://docs.oracle.com/en/java/javase/11/security/java-secure-socket-extension-jsse-reference-guide.html>.
- [230] Open Web Application Security Project, May 2020. <https://www.owasp.org/>.
- [231] OWASP Benchmark, May 2020. <https://owasp.org/www-project-benchmark/>.
- [232] Test cases for risky or broken cryptographic algorithm erroneously labeled as not vulnerable · Issue #92 · OWASP/Benchmark, November 2020. <https://github.com/OWASP/Benchmark/issues/92>.
- [233] HERNAN PALOMBO, ARMIN ZIAIE TABARI, DANIEL LENDE, JAY LIGATTI, AND XINMING OU. An ethnographic understanding of software (In)Security and a Co-

- Creation model to improve secure software development. In *Sixteenth Symposium on Usable Privacy and Security (SOUPS 2020)*, pages 205–220. USENIX Association, August 2020.
- [234] KAI PAN, SUNGHUN KIM, AND E. JAMES WHITEHEAD. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, June 2009.
- [235] SEBASTIANO PANICHELLA, GERARDO CANFORA, AND ANDREA DI SORBO. “Won’t We Fix this Issue?” Qualitative characterization and automated identification of wontfix issues on GitHub. *Information and Software Technology*, 139:106665, November 2021.
- [236] FELIX PAUCK, ERIC BODDEN, AND HEIKE WEHRHEIM. Do Android Taint Analysis Tools Keep Their Promises? In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018*, pages 331–341, New York, NY, USA, 2018. ACM.
- [237] PAUL PEARCE, ADRIENNE PORTER FELT, GABRIEL NUNEZ, AND DAVID WAGNER. AdDroid: Privilege Separation for Applications and Advertisers in Android. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, 2012.
- [238] KAI PETERSEN, SAIRAM VAKKALANKA, AND LUDWIK KUZNIARZ. Guidelines for conducting systematic mapping studies in software engineering: An update. *Information and Software Technology*, 64:1–18, August 2015.
- [239] UPSORN PRAPHAMONTRIPONG, JEFF OFFUTT, LIN DENG, AND JINGJING GU. An experimental evaluation of web mutation operators. In *International Conference on Software Testing, Verification, and Validation, ICSTW’16*, pages 102–111, 2016.

- [240] LinkedIn/qark, June 2020. https://github.com/linkedin/qark/blob/master/qark/plugins/crypto/ecb_cipher_usage.py#L31.
- [241] LINA QIU, YINGYING WANG, AND JULIA RUBIN. Analyzing the analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis - ISSTA 2018*, pages 176–186, Amsterdam, Netherlands, 2018. ACM Press.
- [242] Broker-J - Apache Qpid, May 2020. <https://qpid.apache.org/components/broker-j/>.
- [243] 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, London, United Kingdom, November 2019. ACM Press.
- [244] S. RASTHOFER, S. ARZT, E. LOVAT, AND E. BODDEN. Droidforce: Enforcing complex, data-centric, system-wide policies in android. In *2014 Ninth International Conference on Availability, Reliability and Security*, pages 40–49, Sept 2014.
- [245] VAIBHAV RASTOGI, YAN CHEN, AND WILLIAM ENCK. AppsPlayground: Automatic Large-scale Dynamic Analysis of Android Applications. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, February 2013.
- [246] BRADLEY REAVES, JASMINE BOWERS, SIGMUND ALBERT GORSKI III, OLABODE ANISE, RAHUL BOBHATE, RAYMOND CHO, HIRANAVA DAS, SHARIQUE HUSSAIN, HAMZA KARACHIWALA, NOLEN SCAIFE, BYRON WRIGHT, KEVIN BUTLER, WILLIAM ENCK, AND PATRICK TRAYNOR. * droid: Assessment and Evaluation of

- Android Application Analysis Tools. *ACM Computing Surveys (CSUR)*, 49(3):55, 2016.
- [247] GEMA RODRÍGUEZ-PÉREZ, GREGORIO ROBLES, ALEXANDER SEREBRENİK, ANDY Z Aidman, DANIEL M. GERMÁN, AND JESUS M. GONZALEZ-BARAHONA. How bugs are born: A model to identify how bugs are introduced in software components. *Empirical Software Engineering*, 25(2):1294–1340, March 2020.
- [248] CAITLIN SADOWSKI, EDWARD AFTANDILIAN, ALEX EAGLE, LIAM MILLER-CUSHON, AND CIERA JASPAN. Lessons from Building Static Analysis Tools at Google. *Commun. ACM*, 61(4):58–66, mar 2018.
- [249] CAITLIN SADOWSKI, JEFFREY VAN GOGH, CIERA JASPAN, EMMA SODERBERG, AND COLLIN WINTER. Tricorder: Building a Program Analysis Ecosystem. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, pages 598–608. IEEE, 2015.
- [250] ANTU SAHA AND OSCAR CHAPARRO. Decoding the Issue Resolution Process in Practice via Issue Report Analysis: A Case Study of Firefox. In *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, May 2025. to be published.
- [251] Sarif Viewer - Visual Studio Marketplace. <https://marketplace.visualstudio.com/items?itemName=MS-SarifVSCode.sarif-viewer>.
- [252] RAIMONDAS SASNAUSKAS AND JOHN REGEHR. Intent fuzzer: Crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, WODA+PERTEA 2014, pages 1–5, New York, NY, USA, 2014. ACM.
- [253] MICHAEL SCHLICHTIG, ANNA-KATHARINA WICKERT, STEFAN KRÜGER, ERIC

- BODDEN, AND MIRA MEZINI. CamBench – Cryptographic API Misuse Detection Tool Benchmark Suite, April 2022.
- [254] SHASHI SHEKHAR, MICHAEL DIETZ, AND DAN S. WALLACH. AdSplit: Separating Smartphone Advertising from Applications. In *Proceedings of the USENIX Security Symposium*, 2012.
- [255] FENG SHEN, NAMITA VISHNUHOTLA, CHIRAG TODARKA, MOHIT ARORA, BABU DHANDAPANI, STEVEN Y. KO, AND LUKASZ ZIAREK. Information Flows as a Permission Mechanism. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2014.
- [256] HAIHAO SHEN, JIANHONG FANG, AND JIANJUN ZHAO. EFindBugs: Effective Error Ranking for FindBugs. In *Verification and Validation 2011 Fourth IEEE International Conference on Software Testing*, pages 299–308, March 2011.
- [257] ShiftLeft Scan, November 2020. <https://shiftleft.io/scan>.
- [258] VIJAY SIVARAMAN, DOMINIC CHAN, DYLAN EARL, AND ROKSANA BORELI. Smart-phones attacking smart-homes. In *Proceedings of the 9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, pages 195–200. ACM, 2016.
- [259] ROCKY SLAVIN, XIAOYIN WANG, MITRA BOKAEI HOSSEINI, JAMES HESTER, RAM KRISHNAN, JASPREET BHATIA, TRAVIS D. BREAU, AND JIANWEI NIU. Toward a framework for detecting privacy policy violations in android application code. In *Proceedings of the 38th International Conference on Software Engineering, ICSE '16*, pages 25–36, New York, NY, USA, 2016. ACM.
- [260] STEPHEN SMALLEY AND ROBERT CRAIG. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Proceedings of the ISOC Network and Distributed Systems Symposium (NDSS)*, 2013.

- [261] JUSTIN SMITH, BRITTANY JOHNSON, EMERSON MURPHY-HILL, BILL CHU, AND HEATHER RICHTER LIPFORD. Questions developers ask while diagnosing potential security vulnerabilities with static analysis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 248–259, Bergamo Italy, August 2015. ACM.
- [262] Developer security — Snyk, June 2024. <https://snyk.io/>.
- [263] Code Quality and Security — SonarQube. <https://www.sonarqube.org/>, May 2020.
- [264] Code Quality, Security & Static Analysis Tool with SonarQube — Sonar, June 2024. <https://www.sonarsource.com/products/sonarqube/>.
- [265] Soundiness Home Page. <http://soundiness.org/>.
- [266] DAVID SOUNTHIRARAJ, JUSTIN SAHS, GARRETT GREENWOOD, ZHIQIANG LIN, AND LATIFUR KHAN. SMV-HUNTER: Large Scale, Automated Detection of SSL/TLS Man-in-the-Middle Vulnerabilities in Android Apps. In *Proceedings 2014 Network and Distributed System Security Symposium*, San Diego, CA, 2014. Internet Society.
- [267] Find Security Bugs, May 2020. <https://find-sec-bugs.github.io/>.
- [268] find-sec-bugs/find-sec-bugs: The SpotBugs plugin for security audits of Java web applications and Android applications. (Also work with Kotlin, Groovy and Scala projects), April 2021. <https://github.com/find-sec-bugs/find-sec-bugs>.
- [269] android - Google Play Warning: How to fix incorrect implementation of HostnameVerifier? - Stack Overflow, June 2020. <https://stackoverflow.com/a/41330005>.

- [270] android - unsafe implementation of the HostnameVerifier interface - Stack Overflow, June 2020. <https://stackoverflow.com/questions/47069277/unsafe-implementation-of-the-hostnameverifier-interface>.
- [271] Cryptography Stack Exchange, June 2020. <https://crypto.stackexchange.com>.
- [272] Java - an Unsafe Implementation of the Interface X509TrustManager from Google, June 2020. <https://stackoverflow.com/questions/35545126/an-unsafe-implementation-of-the-interface-x509trustmanager-from-google>.
- [273] jeesuite-libs/DES.java at vakinge/jeesuite-libs, June 2020. <https://github.com/vakinge/jeesuite-libs/blob/master/jeesuite-common/src/main/java/com/jeesuite/common/crypt/DES.java#L37>.
- [274] Stack Overflow - Where Developers Learn, Share, & Build Careers, June 2020. <https://stackoverflow.com>.
- [275] UltimateAndroid/TripleDES.java at cymcsg/UltimeAndroid, June 2020. <https://github.com/cymcsg/UltimeAndroid/blob/678afdda49d1e7c91a36830946a85e0fda541971/UltimeAndroid/ultimateandroid/src/main/java/com/marshalchen/ua/common/commonUtils/urlUtils/TripleDES.java#L144>.
- [276] Bypass SSL Warning from google play, April 2021. <https://stackoverflow.com/questions/36913633/bypass-ssl-warning-from-google-play>.
- [277] How to bypass SSL certificate validation in Android app?, April 2021. <https://stackoverflow.com/questions/35548162/how-to-bypass-ssl-certificate-validation-in-android-app>.
- [278] TREVOR STALNAKER, NATHAN WINTERSGILL, OSCAR CHAPARRO, MASSIMILIANO DI PENTA, DANIEL M. GERMAN, AND DENYS POSHYVANYK. BOMs Away! Inside

the Minds of Stakeholders: A Comprehensive Study of Bills of Materials for Software Systems, September 2023.

- [279] STEVEN ARTZ. FlowDroid 2.0. <https://github.com/secure-software-engineering/soot-infoflow/releases>.
- [280] ROCK STEVENS, FARIS BUGRA KOKULU, ADAM DOUPÉ, AND MICHELLE L. MAZUREK. Above and Beyond: Organizational Efforts to Complement U.S. Digital Security Compliance Mandates. In *Proceedings 2022 Network and Distributed System Security Symposium*. Internet Society, 2022.
- [281] STREAM101. Possible to integrate Fragment lifecycle ? <https://github.com/secure-software-engineering/soot-infoflow-android/issues/52>.
- [282] Software Assurance Marketplace, June 2020. <https://continuousassurance.org/>.
- [283] JUNWEI TANG, JINGJING LI, RUIXUAN LI, HONGMU HAN, XIWU GU, AND ZHIYONG XU. SSLDetector: Detecting SSL Security Vulnerabilities of Android Applications Based on a Novel Automatic Traversal Method. *Security and Communication Networks*, 2019:1–20, October 2019.
- [284] SYNOPSYS TECHNOLOGY. Coverity SAST Software — Synopsys, May 2020. <https://www.synopsys.com/software-integrity/security-testing/static-analysis-sast.html>.
- [285] REP. RITCHIE TORRES. H.R.4611 - DHS Software Supply Chain Risk Management Act of 2021, October 2021.
- [286] UNITED STATES SENATE. S.965 - cyber shield act of 2021. <https://www.congress.gov/bill/117th-congress/senate-bill/965>, 2021.
- [287] 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*, page 13. IBM Press, 1999.
- [288] CARMINE VASSALLO, SEBASTIANO PANICHELLA, FABIO PALOMBA, SEBASTIAN PROKSCH, HARALD C. GALL, AND ANDY ZAIDMAN. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering*, 25(2):1419–1457, March 2020.
- [289] AMIT VASUDEVAN, SAGAR CHAKI, LIMIN JIA, JONATHAN McCUNE, JAMES NEWSOME, AND ANUPAM DATTA. Design, implementation and verification of an extensible and modular hypervisor framework. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 430–444, 2013.
- [290] VERACODE. Veracode’s 10th state of software security report finds organizations reduce rising ‘security debt’ via devsecops, special sprints. <https://www.veracode.com/veracodes-10th-state-software-security-report-finds-organizations-reduce> 2020.
- [291] TIMOTHY VIDAS, NICOLAS CRISTIN, AND LORRIE FAITH CRANOR. Curbing Android Permission Creep. In *Proceedings of the Workshop on Web 2.0 Security and Privacy (W2SP)*, 2011.
- [292] DANIEL VOTIPKA, ROCK STEVENS, ELISSA REDMILES, JEREMY HU, AND MICHELLE MAZUREK. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391, 2018.
- [293] FENGGUO WEI, SANKARDAS ROY, XINMING OU, AND ROBBY. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Transactions on Privacy and Security*, 21(3):1–32, April 2018.

- [294] FENGGUO WEI, SANKARDAS ROY, XINMING OU, AND ROBBY. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. *ACM Transactions on Privacy and Security*, 21(3):1–32, August 2018.
- [295] MING WEN, YEPANG LIU, RONGXIN WU, XUAN XIE, SHING-CHI CHEUNG, AND ZHENDONG SU. Exposing Library API Misuses Via Mutation Analysis. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 866–877, May 2019.
- [296] ANNA-KATHARINA WICKERT, LARS BAUMGÄRTNER, MICHAEL SCHLICHTIG, KRISHNA NARASIMHAN, AND MIRA MEZINI. To Fix or Not to Fix: A Critical Study of Crypto-misuses in the Wild. In *The 21th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2022.
- [297] ANNA-KATHARINA WICKERT, MICHAEL REIF, MICHAEL EICHBERG, ANAM DODHY, AND MIRA MEZINI. A Dataset of Parametric Cryptographic Misuses. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 96–100, May 2019.
- [298] JIM WITSCHHEY, OLGA ZIELINSKA, ALLAIRE WELK, EMERSON MURPHY-HILL, CHRIS MAYHORN, AND THOMAS ZIMMERMANN. Quantifying developers’ adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 260–271, Bergamo Italy, August 2015. ACM.
- [299] GLENN WURSTER AND P. C. VAN OORSCHOT. The developer is the enemy. In *Proceedings of the 2008 New Security Paradigms Workshop*, pages 89–97, Lake Tahoe California USA, September 2008. ACM.
- [300] XANITIZER. Xanitizer by RIGS IT - Because Security Matters, May 2020. <https://www.rigs-it.com/xanitizer/>.

- [301] SHUNDAN XIAO, JIM WITSCHHEY, AND EMERSON MURPHY-HILL. Social influences on secure development tool adoption: Why security tools spread. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work & Social Computing*, pages 1095–1106, Baltimore Maryland USA, February 2014. ACM.
- [302] RUBIN XU, HASSEN SAIDI, AND ROSS ANDERSON. Aurasium: Practical Policy Enforcement for Android Applications. In *Proceedings of the USENIX Security Symposium*, 2012.
- [303] YUANZHONG XU AND EMMETT WITCHEL. Maxoid: transparently confining mobile applications with custom views of state. In *Proceedings of the Tenth European Conference on Computer Systems*, page 26, April 2015.
- [304] JEAN YANG, KUAT YESSENOV, AND ARMANDO SOLAR-LEZAMA. A Language for Automatically Enforcing Privacy Policies. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.
- [305] W. YANG, X. XIAO, B. ANDOW, S. LI, T. XIE, AND W. ENCK. Appcontext: Differentiating malicious and benign mobile app behaviors using context. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 303–313, May 2015.
- [306] FIORELLA ZAMPETTI, RITU KAPUR, MASSIMILIANO DI PENTA, AND SEBASTIANO PANICHELLA. An empirical characterization of software bugs in open-source Cyber-Physical Systems. *Journal of Systems and Software*, 192:111425, October 2022.
- [307] LI ZHANG, JIONGYI CHEN, WENRUI DIAO, SHANQING GUO, JIAN WENG, AND KEHUAN ZHANG. CryptoREX: Large-scale analysis of cryptographic misuse in IoT devices. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 151–164, Chaoyang District, Beijing, September 2019. USENIX Association.

- [308] CHIXIANG ZHOU AND PHYLLIS G. FRANKL. Mutation testing for java database applications. In *Second International Conference on Software Testing Verification and Validation, ICST 2009, Denver, Colorado, USA, April 1-4, 2009*, pages 396–405, 2009.
- [309] YAJIN ZHOU AND XUXIAN JIANG. Dissecting Android Malware: Characterization and Evolution. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2012.
- [310] YAJIN ZHOU, ZHI WANG, WU ZHOU, AND XUXIAN JIANG. Hey, You, Get off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets. In *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS)*, February 2012.
- [311] YAJIN ZHOU, XINWEN ZHANG, XUXIAN JIANG, AND VINCENT W. FREEH. Taming Information-Stealing Smartphone Applications (on Android). In *Proceedings of the International Conference on Trust and Trustworthy Computing (TRUST)*, June 2011.
- [312] THOMAS ZIMMERMANN, RAHUL PREMRAJ, NICOLAS BETTENBURG, SASCHA JUST, ADRIAN SCHROTER, AND CATHRIN WEISS. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering*, 36(5):618–643, September 2010.
- [313] WEIQIN ZOU, DAVID LO, ZHENYU CHEN, XIN XIA, YANG FENG, AND BAOWEN XU. How Practitioners Perceive Automated Bug Report Management Techniques. *IEEE Transactions on Software Engineering*, 46(8):836–862, August 2020.
- [314] CHAOSHUN ZUO, JIANLIANG WU, AND SHANQING GUO. Automatically Detecting SSL Error-Handling Vulnerabilities in Hybrid Mobile Web Apps. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15*, pages 591–596, New York, NY, USA, 2015. ACM.

Appendix A

Appendix

A.1 Appendix of Chapter 3

```
1  BroadcastReceiver receiver = new BroadcastReceiver() {
2      @Override
3      public void onReceive(Context context, Intent intent) {
4          BroadcastReceiver receiver = new BroadcastReceiver() {
5              @Override
6                  public void onReceive(Context context, Intent intent) {
7                      String dataLeak = Calendar.getInstance().getTimeZone().
8                          getDisplayName();
9                      Log.d("leak-1", dataLeak);}};
10         registerReceiver(receiver, new IntentFilter().addAction("android.intent.
11             action.SEND"));
12     };
13     registerReceiver(receiver, new IntentFilter().addAction("android.intent.action.
14         SEND"));
```

Listing A.1: A dynamically-created BroadcastReceiver, created inside another, with data leak. Whenever the onReceive() callback of the receiver object is invoked, it will create another receiver object of similar type, with a leak inside its own onReceive() callback. This can be further evolved to use anonymous object declaration that can leak information in a similar nature.

CrashScope (Execution Engine): The EE functions builds upon CrashScope [201, 199], which statically analyzes the code of a target app to identify activities implementing potential contextual features (*e.g.*, rotation, sensor usage) via API call-chain propagation.

This execution is guided by one of several exploration strategies, organized along three dimensions: (i) GUI-exploration, (ii) text-entry, and (iii) contextual features.

Note that because the goal of the EE is to explore as many screens of a target app as possible, the EE forgoes certain combinations of exploration strategies from CRASH-SCOPE [201, 199] (*e.g.*, entering unexpected text or disabling contextual features) prone to eliciting crashes from apps. The approach uses adb and Android’s uiautomator framework to interact with and extract GUI-related information from a target device or emulator. Further implementation details of exploration strategies can be found in [201, 199].

Table A.1: List of App names, URLs and IDs assigned by us for the purpose of the μ SE study

App ID	Android App name	URL
app 01	2048	https://f-droid.org/en/packages/com.uberspot.a2048/
app 02	Protect Baby Monitor	https://f-droid.org/en/packages/protect.babymonitor/
app 03	QR Scanner	https://f-droid.org/en/packages/com.secuso.privacyFriendlyCodeScanner/
app 04	Location Share	https://f-droid.org/en/packages/ca.cmetcalfe.locationshare/
app 05	Camera Roll	https://f-droid.org/en/packages/us.koller.cameraroll/
app 06	AndroidPN Client	https://f-droid.org/en/packages/org.androidpn.client/
app 07	Activity Launcher	https://f-droid.org/en/packages/de.szalkowski.activitylauncher/
app 08	Man Man	https://f-droid.org/en/packages/com.adonai.manman/
app 09	BMI Calculator	https://f-droid.org/en/packages/com.zola.bmi/
app 10	A Time Tracker	https://f-droid.org/en/packages/com.markuspage.android.atimetracker/
app 11	AFH Downloader	https://f-droid.org/en/packages/org.afhdownloader/
app 12	Android Explorer	https://f-droid.org/en/packages/com.iamtrk.androidexplorer/
app 13	Kaltura Device Info	https://f-droid.org/en/packages/com.oF2pks.kalturadeviceinfos/
app 14	Apod Classic	https://f-droid.org/en/packages/com.jvillalba.apod.classic/
app 15	Calendar Trigger	https://f-droid.org/en/packages/uk.co.yahoo.plrpp.calendartrigger/

Apps used in the study: For our study, we collected a set of 15 open-source apps from F-Droid [102], as shown in Table A.1. The apps come from different heterogeneous build configuration settings, with compile SDK API level 23 (Marshmallow) to 27 (Oreo), minimum SDK API level 9 (Gingerbread) to 16 (Jelly Bean), and target SDK API level from 17 (Jelly Bean) to 26 (Oreo), with sizes ranging from several hundred KB to a maximum of 3.5 MB.

A.2 Additional Evaluation Details

We provide additional details about evaluation, *e.g.*, rationale for choosing certain mutation operators for evaluation, the necessity of optimizing number of mutations generated, and details about confirmation of killed mutations are available in S&P’22 [19] and in the online appendix [16].

Table A.2: List of Applications mutated using MASC for the current study, CLOC = Count Lines of Code from Java Source files only [68], Source = Source Code collected from/Originated From

ID	Name	Type	Source	CLOC
T1	Fake Traveler	Android	GitHub	552
T2	Simple-Solitaire	Android	GitHub	29,303
T3	OpenTracks	Android	GitHub	27,432
T4	RevenueCat	Android	GitHub	1,294
T5	Orbot	Android	GitHub	6,231
T6	Mastodon	Android	GitHub	31,328
T7	Snapdrop	Android	GitHub	1,782
T8	OpenTasks	Android	GitHub	34,105
T9	Nextcloud Notes	Android	GitHub	24,262
T10	MIFARE Classic Tool	Android	GitHub	7,807
T11	Armadillo - Encrypted Shared Preference	Android	GitHub	3,809
T12	Authorizer	Android	GitHub	35,241
T13	Android GoldFinger	Android	GitHub	2,374
T14	OVAA	Android	GitHub	687
T15	Tink	Java	GitHub	97,703

Table A.3: Relevance of crypto-detectors evaluated using MASC for the Extended Study

Tool	Practical Relevance	Deployment Use Case
<i>DeepSource</i> (Industry)	Available in Github Code Scan integration [119]	<i>Prevent hundreds of known security vulnerabilities in your code and stay compliant with industry standards.</i> [77]
<i>SonarQube</i> (Industry)	Available in Github Code Scan integration [119]	<i>... detects security vulnerabilities in your code so they can be eliminated before you build and test your application</i> [264]
<i>Amazon CodeGuru Security</i> (Industry)	Used in the Industry (<i>e.g.</i> , by Amazon, and offered as service) [69]	<i>Detect security vulnerabilities at any stage of the development lifecycle</i> [69]
<i>Codiga</i> (Industry)	Available in Github Code Scan integration [119]	<i>...find critical application vulnerabilities, such as Mitre CWE, SANS CWE Top 25 and OWASP Top 10</i> [70]
<i>Snyk</i> (Industry)	Available in Github Code Scan integration [119]	<i>Secure your code as it's written with static application security testing built by, and for, developers.</i> [262]

*We obtained full licenses for these proprietary tools for first iteration of this study, ¹Xanitizer has been merged to another product and is not included in this extension, ²We did not obtain license for this extended study

A.3 Crypto-API Misuse Taxonomy Data

Additional details of the taxonomy data, *e.g.*, types of cases that our SLR approach may miss, is available in the S&P’22 [19] paper and the online appendix [16].

Table A.4: Selected Sources for Extracting Cryptography Misuse, from Academia from 2019-2022

ID	Title	Year	Venue
36	Ensuring correct cryptographic algorithm and provider usage at compile time	2021	FTfJP
37	Why Eve and Mallory Still Love Android: Revisiting TLS (In)Security in Android Applications	2021	USENIX
38	Towards HTTPS Everywhere on Android: We Are Not There Yet	2020	USENIX
39	CRYPTOAPI-BENCH: A Comprehensive Benchmark on Java Cryptographic API Misuses	2019	SecDev
40	CRYPTOREX: Large-scale Analysis of Cryptographic Misuse in IoT Devices	2019	USENIX
41	Negative Results on Mining Crypto-API Usage Rules in Android Apps	2019	MSR
42	Java Cryptography Uses in the Wild	2020	ESEM
43	Python Crypto Misuses in the Wild	2021	ESEM
44	Understanding How to Use Static Analysis Tools for Detecting Cryptography Misuse in Software	2019	ToR
45	A Comparative Study of Misapplied Crypto in Android and iOS Applications	2019	ICETE
46	A Dataset of Parametric Cryptographic Misuses	2019	MSR
47	CogniCryptGEN: generating code for the secure usage of crypto APIs	2020	CGO
48	CryptoTutor: Teaching Secure Coding Practices through Misuse Pattern Detection	2020	SIGITE
49	Using Graph Embeddings and Machine Learning to Detect Cryptography Misuse in Source Code	2020	ICMLA
50	Evaluation of Static Vulnerability Detection Tools with Java Cryptographic API Benchmarks	2022	TSE
51	Automatic Detection of Java Cryptographic API Misuses: Are We There Yet?	2022	TSE
52	Hotfixing Misuses of Crypto APIs in Java Programs	2021	ICCSS
53	CRYLOGGER: Detecting Crypto Misuses Dynamically	2021	SP
54	CRYScanner: Finding cryptographic libraries misuse	2021	NICS

A.4 Appendix

A.4.1 Code Snippets

```

1 Class T { String algo="AES/CBC/PKCS5Padding";
2 T mthd1(){ algo = "AES"; return this;} T mthd2(){ algo="DES"; return this;} }
3 Cipher.getInstance(new T().mthd1().mthd2());

```

```

1 val = new Date(System.currentTimeMillis()).toString();
2 new IvParameterSpec(val.getBytes(),0,8);}

```

```

1 void checkServerTrusted(X509Certificate[] x, String s)
2 throws CertificateException {
3 if (!(null != s && s.equalsIgnoreCase("RSA"))) {
4     throw new CertificateException("not RSA");}

```

```

1 public boolean verify(String host, SSLSession s) {
2     if(true || s.getCipherSuite().length()>=0){
3         return true;} return false;}

```

```

1 interface ITM extends X509TrustManager { }
2 abstract class ATM implements X509TrustManager { }

```

```

1 new HostnameVerifier(){
2     public boolean verify(String h, SSLSession s) {
3         return true; } };

```

```

1 new X509ExtendedTrustManager(){
2     public void checkClientTrusted(X509Certificate[] chain, String a) throws
        CertificateException {}
3     public void checkServerTrusted(X509Certificate[] chain, String authType)throws
        CertificateException {}
4     public X509Certificate[] getAcceptedIssuers() {return null;} ...};

```

```

1 void checkServerTrusted(X509Certificate[] certs, String s)
2     throws CertificateException {
3     if (!(null != s || s.equalsIgnoreCase("RSA") || certs.length >= 314)) {
4         throw new CertificateException("Error");}

```

```

1 abstract class AHV implements HostnameVerifier{ new AHV(){
2     public boolean verify(String h, SSLSession s)
3         return true;}};

```

Listing A.10: Anonymous Inner Class Object of An Empty Abstract Class that implements HostnameVerifier

```

1 abstract class AbstractTM implements X509TrustManager{} new AbstractTM(){
2     public void checkServerTrusted(X509Certificate[] chain, String authType) throws
      CertificateException {}
3     public X509Certificate[] getAcceptedIssuers() {return null;}}};

```

Listing A.11: Anonymous inner class object with a vulnerable checkServerTrusted

```

met
1 interface IHV extends HostnameVerifier{} new IHV(){
2     public boolean verify(String h, SSLSession s) return true;}};

```

List

```

1 KeyGenerator keyGen = KeyGenerator.getInstance("AES");
2 keyGen.init(128); SecretKey secretKey=keyGen.generateKey();

1 if (!className.contains("android."))
2     classNames.add(className.substring(1, className.length() - 1)); return classNames;

1 if (!(true || arg0==null || arg1==null)) {
2     throw new CertificateException();}

1 this.name = name == null ? "AES" : name;
2 this.mode = mode == null ? "CBC" : mode;
3 this.pad = pad == null ? "PKCS5Padding" : pad;
4 this.string = StringUtils.format("%s/%s/%s", this.name, this.mode, this.pad);

```

Listing A.16: Transformation String formation in Apache Druid similar to \mathcal{F}_2 which uses AES in

```

CBC
1 Class T {
2     int i = 0;
3     cipher = "AES/GCM/NoPadding";
4     public void A(){
5         cipher = "AES/GCM/NoPadding";
6     }
7     public void B(){
8         cipher = "AES/GCM/NoPadding";
9     }
10    public void C(){
11        cipher = "AES/GCM/NoPadding";
12    }
13    public void D(){
14        cipher = "AES";
15    }
16    public String getVal(){
17        return cipher
18    }
19 }
20 Cipher.getInstance(new T().A().B().C().D().getVal() ) ;

```

Listing A.17: Iterative Method Chaining

```

1 Class T {
2     int i = 0;
3     cipher = "AES/GCM/NoPadding";
4     public void A(){
5         if ( i == 0){
6             if (i == 0){
7                 if(i == 0){
8                     cipher = "AES";
9                 }
10                else{
11                    cipher = "AES/GCM/NoPadding";
12                }
13            }
14            else{
15                cipher = "AES/GCM/NoPadding";
16            }
17        } else{
18            cipher = "AES/GCM/NoPadding";
19        }
20    }
21    public String getVal(){
22        return cipher
23    }
24 }
25 Cipher.getInstance(new T().A().getVal() ) ;

```

```

1 Class T {
2     int i = 0;
3     cipher = "AES/GCM/NoPadding";
4     public String A(){
5         return "D";
6     }
7     public String B(){
8         return "E";
9     }
10    public String C(){
11        return "S";
12    }
13    public void add(){
14        cipher = A() + B() + C();
15    }
16    public String getVal(){
17        return cipher
18    }
19 }
20 Cipher.getInstance(new T().add().getVal() ) ;

```

```

1 T secure = new T();
2 T insecure = new T().mthd2();
3 secure = insecure;
4 Cipher.getInstance(secure.getVal());

```

```

1 String cryptoVariable = "AES";
2 char[] cryptoVariable1 = cryptoVariable.toCharArray();
3 javax.crypto.Cipher.getInstance(String.valueOf(cryptoVariable1));

1 javax.crypto.Cipher.getInstance("secureParamAES".substring(11));

1 byte[] cryptoTemp = "12345678".getBytes();
2 javax.crypto.spec.IvParameterSpec ivSpec = new javax.crypto.spec.IvParameterSpec.getInstance(
  (cryptoTemp, "AES");

```

Listing A.23: Constant IV

A.4.2 Additional Implementation and Evaluation Details

A.4.2.1 Expanded rationale for choosing certain operators

We prioritized misuse cases for inclusion in MASC that are discussed more frequently in the artifacts. For instance, when implementing restrictive operators (Sec. 4.3.1), we chose the misuse of using AES with ECB mode, or using ECB mode in general, as both misuse cases were frequently mentioned in our artifacts (*i.e.*, in 2 and 11 artifacts respectively). Additionally, we chose the misuse of using DES mode, since several crypto-detectors did not consider that not specifying a mode explicitly for encryption defaults to ECB mode. Similarly, we chose the misuse cases of using MD5 algorithm with the MessageDigest API for hashing (5 artifacts), and digital signatures (5 artifacts). When implementing flexible mutation operators (Sec. 4.3.2), we observed that the majority of the misuse cases relate to improper SSL/TLS verification and error handling, and hence chose to mutate the X509TrustManager and HostnameVerifier APIs with **OP**₇ – **OP**₁₃.

A.4.2.2 Do we need to optimize the number of mutants generated?

MASC generates thousands of mutants to evaluate crypto-detectors, which may prompt the question: should we determine exactly *how many* mutants to generate or optimize them? The answer to this question is no, for two main reasons.

First, MASC generates mutants as per the mutation-scope applied, *i.e.*, for the exhaustive scope, it is natural for MASC to seed an instance of the same mutation at every possible/compilable entry point (including internal methods) in the mutated application. Similarly, for the similarity scope, we seed a mutant besides every similar “usage” in the mutated application. Therefore, in all of these cases, every mutant seeded is justified/necessitated by the mutation scope being instantiated. Any reduction in mutants would require MASC to sacrifice the goals of its mutation scopes, which may not be in the interest of a best-effort comprehensive evaluation. Second, in our experience, the number of mutants does not significantly affect the time to seed taken by MASC. That is, MASC took just 15 minutes to seed over 20000 mutants as a part of our evaluation (see Section 4.7). Moreover, once the target tool’s analysis is complete, we only have to analyze the unkilld mutants, which is a far smaller number than those originally seeded (Section 4.7). Therefore, in our experience, there is little to gain (and much to lose) by reducing the number of mutants seeded; *i.e.*, we want to evaluate the tools as thoroughly as we can, even if it means evaluating them with certain mutation instances/mutants that may be effectively similar.

That said, from an analysis perspective, it may be interesting to dive deeper into the relative effectiveness of individual features (*i.e.*, operators as well as scopes), even if they are all individually necessary, as each mutation operator exploits a unique API use characteristic, and scopes exploit unique code-placement opportunities, and any combination of these may appear in real programs. However, it would be premature to determine relative advantages among scopes/operators using the existing evaluation sample (*i.e.*, 9 detectors evaluated, 13 open-source apps mutated, 19 misuse cases instantiated, with 12 operators). For instance, mutating other misuse cases, or evaluating another tool, or using a different set of open source apps to mutate, may all result in additional/different success at the feature-level (although overall, MASC would still find flaws, and satisfy its claims). We defer such an evaluation to determine the relative advantages of different mutation features to future work, as described in Section 4.8.

A.4.2.3 Further details regarding confirming killed mutants

Matching the mutation log generated by MASC with the reports generated by crypto-detectors is challenging because crypto-detectors often generate reports in heterogeneous and often mutually incompatible ways; *i.e.*, GCS, LGTM, ShiftLeft, and more recently, CogniCrypt generate text files following the recently introduced Static Analysis Results Interchange Format (SARIF) [216] format. However, CryptoGuard, Toolx, SpotBugs and QARK generate reports in custom report formats, downloadable as HTML, CSV, or text, web-based services such as Amazon CodeGuru Security, Snyk, SonarQube, and DeepSource offer results through web-based user-interface, and finally, Xanitizer generates PDFs with source code annotations. We developed a semi-automated implementation that allows us to systematically identify uncaught mutants given these disparate formats. For QARK and CryptoGuard, we wrote custom scripts to parse and summarize their reports into a more manageable format, which we then manually reviewed and matched against MASC’s mutation logs. For SARIF formatted reports, we used a VSCode based SARIF viewer [251] that allows iterative searching of logs and tool reports by location. For CogniCrypt, SpotBugs, and Xanitizer, we performed the matching manually since even though they used custom Text or PDF formats, they were generated in such a way that manual checking was trivial. This process is in line with prior work that faces similar challenges [21]. As more tools move to standard formats such as SARIF (which is being promoted by analysis suites such as Github Code Scan) and being adopted by crypto-detectors (*e.g.*, Xanitizer and CogniCrypt adopted SARIF after our 2022 study concluded), we expect the methodology to be fully automated.

A.4.2.4 Why GCS, LGTM, and QARK fail to detect base cases

In our 2022 study, we observe that GCS and LGTM fail to detect base cases (*i.e.*, \emptyset in Table 4.3) for **FC3** – **FC5**, although they claim to find SSL vulnerabilities in Java, due to incomplete rulesets (*i.e.*, the absence of several SSL-related rules) [124]. However, we

Table A.5: Mutants analyzed vs detected by crypto-detectors

Tool	Input Type	Analyzed	Detected
<i>CryptoGuard</i>	apk or jar	45,763	25,299
<i>Xanitizer</i> ¹	Java Src Code & jar	17,788	17,774
<i>CogniCrypt</i>	apk or jar	23,601	4,576
<i>Toolx</i> ²	Android or Java Src Code	9,774	8,547
<i>SpotBugs</i>	jar	17,702	13,848
<i>QARK</i>	Java Src Code or apk	46,324	7
<i>LGTM</i>	Java Src Code	34,846	21,474
<i>GCS</i>	Java Src Code	34,846	21,440
<i>ShiftLeft</i>	Java Src Code	46,252	35,200
<i>Snyk</i>	Java Src Code	47,002	40,877
<i>DeepSource</i>	Java Src Code	47,002	17,028
<i>Codiga</i>	Java Src Code	26,725	0
<i>SonarQube</i>	Java Src Code	13,749	11,601
<i>Amazon CodeGuru Security</i>	Java Src Code	46,967	840

¹Xanitizer has been merged to another product and is not included in this extension,

²We did not obtain license for this extended study

noticed that there was an SSL-related experimental pull request for GCS’s ruleset [125, 132] and even upon integrating it into GCS and LGTM, we found both tools to still be vulnerable to the base cases. In our current study, even after we used the Github Code Security with both the *default*, and *security-extended* test suites as of April 2024, it was unable to detect misuse related to `X509TrustManager` and `HostnameVerifier`. Similarly, QARK fails to detect base cases for all flaws in **FC1** and **FC2**, because of its incomplete ruleset [240].

A.4.3 Types of cases that our SLR approach may miss

Our SLR approach involves manually analyzing each document in an attempt to include all misuse cases, but this extraction of misuse cases is often affected by the context in which they are expressed. For instance, CogniCrypt’s core philosophy is whitelisting, which is reflected throughout its papers and documentation. However, there are two ways in which whitelisting is expressed in the paper, one concerning functionality, and another

security, *i.e.*, cases of desired behavior expressed in the paper may not always indicate a security best-practice. For instance, the ORDER keyword in the CrySL language initially caused us to miss the PBEKeySpec misuse (now included in the taxonomy), because as defined in the paper, ORDER keyword allows defining “*usage patterns*” that will not break functionality. Thus, as the “usage” patterns were not security misuses (or desired behaviors for security), we did not include them as misuse cases in the taxonomy. However, in a later part of the paper, the ORDER keyword is used to express a security-sensitive usage, for PBEKeySpec, but the difference in connotation is not made explicit. This implicit and subtle context-switch was missed by both our annotators in the initial SLR, but fixed in a later iteration, and misuse cases related to the ORDER keyword were added to the taxonomy.

Similarly when labeling for misuse extraction (Sec. 4.2.3) we marked each misuse found in a document using common terminology (*i.e.*, labels) across all documents. Thus, if a misuse found in the current document was previously discovered and annotated with a particular label, we would simply apply the same label to the newly found instance. This standard, best-practice approach [171, 197] makes it feasible to extract a common taxonomy from a variety of documents written by different authors, who may use inconsistent terminology. However, a limitation of this generalization is that in a rare case wherein a particular example may be interpreted as two different kinds of misuse, our approach may lose context and label it as only one type of misuse. For instance, based on how the misuse of a “password stored in String” was described in most of the documents we studied, the misuse label of “using a hardcoded password” was applied to identify it across the documents. However, this results in the loss of the additional, semantically different misuse that may still be expressed in terms of a “password stored in String”, that passwords should not be stored/used in a String data construct for garbage collection-related reasons. Note that this problem would only occur in rare instances wherein (1) there are multiple contexts/interpretations of the same misuse example, and (2) only one or few document(s) use the additional context. This misuse has also been included in the

taxonomy.

A.4.4 Additional Evaluation Data

Table A.6: List of Applications mutated using MASC in previous study (S&P’22 [19]), CLOC = Count Lines of Code from Java Source files only [68], Source = Source Code collected from/O-riginated From

ID	Name	Type	Source	CLOC
A1	2048	Android	GitHub	136
A2	BMI Calculator	Android	GitHub	145
A3	Calendar Trigger	Android	GitHub	8,553
A4	LocationShare	Android	GitHub	215
A5	NasaApodCL	Android	GitHub	706
A6	AFH Downloader	Android	GitHub	1,657
A7	A Time Tracker	Android	GitHub	2,928
A8	Kaltura Device Info	Android	GitHub	1,049
A9	Protect Baby Monitor	Android	GitHub	625
A10	Activity Monitor	Android	GitHub	1,168
A11	personalDNSfilter	Android	GitHub	8,446
A12	aTalk	Android	GitHub	254,364
A13	Car Report	Android	BitBucket	16,966
Apache Qpid™ Broker-J				
J14.1	Broker-J - AMQP/JDBC	Java	Apache	597
J14.2	Broker-J - Tools	Java	Apache	1,725
J14.3	Broker-J - HTTP	Java	Apache	24,141
J14.4	Broker-J - Core	Java	Apache	127,280

A.5 Survey Protocol

To understand how practitioners perceive security tools, and whether security is prioritized by individuals and organizations similarly, we prepared an online survey questionnaire (questionnaire in the online appendix [228]) and drafted a research protocol. We piloted the initial survey with five participants. Three of the participants were graduate students, and the rest had doctoral degrees. All pilots were from computer science background, with additional experience in software engineering and/or security. By incorporating their

feedback, we improved the survey by modifications and additional descriptions as necessary. Our final survey protocol received the approval of our Institutional Review Boards (IRBs). The experimental protocol of both our survey and interviews included a consent form which emphasized that the data of the participants will remain confidential and de-identified. Furthermore, a participant could optionally submit their email address to have the chance of winning one of two \$50.00 gift cards or the equivalent value in local currency vouchers. The winners would be chosen from qualified participants who completed the survey and provided valid responses in the survey.

A.5.1 Survey Recruitment

To diversify our recruitment approach in terms of experience, culture and industry contexts, we leveraged multiple recruitment channels. We sent invitation emails describing the goal of the survey (*i.e.*, in order to learn about their professional experiences and opinions about vulnerability detectors) to our professional networks, relying on snowball sampling for recruitment, as well as to OSS developers (as previously described in Section 5.1.1).

Ethical Considerations in Recruiting OSS developers: We collected publicly available email addresses only, and explicitly stated our recruitment procedure in our initial contact, which is common in other recent studies (*e.g.*, Endres et al. [98]). We considered several *potential trade-offs* that factored into this recruitment strategy, in addition to following the guidance provided by our IRB: (a) It is *difficult* to recruit practitioners across borders who have the relevant experience, *i.e.*, configured and used automated security analysis tools, (b) we were collecting publicly available information and not amplifying the visibility of the individuals' email address, and (c) we carefully considered the Menlo Report's ethical guidelines [170, 88]. Specifically based on these guidelines, the only potential *harm* to an invited person would be receiving one unsolicited email, whereas the potential benefit of this research is in helping create more secure software, for everyone, by understanding the needs and challenges of practitioners related to security analysis

techniques.

A.5.2 Online Survey and Data Analysis

Our survey (provided in the online appendix [228]) consisted of Likert Scale based questions, with optional, open-ended response to clarify their selected choice(s). Our analysis prioritized the text-based responses since these provided additional context for the selected choice(s) in Likert scale. One of the authors open-coded the responses for analysis. The responses of the survey, which we summarize next, guided our interview protocol.

A.6 Survey Results

The results of our survey helped us refine the semi-structure guide of questions for the interview. We now describe the demographics as well as general results elicited from the survey responses.

Demographics: Of the 39 responses we received, 25 worked in a full time employment, and 12 worked as both freelancers and full-time employees. Almost all of them (85%) identified themselves as developers with 25 of them having more than five years of professional experience and six with at least three years of experience. 53% participants helped release a new version of software or service at least on a monthly basis in the past two years, with 26% on quarterly basis. All the participants ranked themselves as at least slightly knowledgeable in security, with five being extremely knowledgeable, eight very knowledgeable and 20 moderately knowledgeable. 50% of the participants entered their location as Asia, with the rest distributed equally between North America, Europe, United Kingdom and Africa.

Prioritizing Security by Organizations and Individuals: Through the survey, we asked the participants to rate the importance of privacy, security against malicious attacks, ease of use, multi-platform compatibility, multitude of features and responsiveness with respect to applications they help develop from their individual perspective. Furthermore,

we asked the participants to rate how these are prioritized by their organizations based on their personal experience.

All participants individually expressed that securing against malicious attack is very important, with 83% working in organizations expressing that it is of extreme importance. However, from their organization's perspective, only 30/35 participants shared that securing against malicious attacks is *at least* very important, with two selecting slightly important and three moderately important. The remaining two participants chose not to answer. In other words, the importance of security against malicious attacks might not be prioritized similarly by an organization and an individual of the same organization. For similar questions about protecting privacy in software and or services, 25 participants expressed that it is at least very important, with two selecting moderately important. Similar to the trend observed for securing against malicious attacks, participants expressed that they think their organizations prioritizes privacy differently compared to themselves.

To summarize, *an organization and its practitioners can have significantly different priorities on security and privacy* for their software or services.

Reliance on Automated and/or Manual Analysis Techniques: When asked how the participants relied on automated and manual techniques for finding security vulnerabilities, seven participants expressed that they rely on automated techniques for reasons such as lack of security-related expertise, manual testing being time-consuming and for automatically preventing intruders from attacking their systems. All the participants (26/39) who chose both automated analysis and manual analysis techniques expressed that they do it because of additional coverage, with the manual technique being used to cover corner cases, application specific logic, or out of scope issues. Finally, the participants (6/39) who expressed that they rely only on manual analysis techniques shared that it is due to lack of effectiveness, or lack of resources, or due to simply being more comfortable with manual analysis techniques.

To summarize, *practitioners mostly rely on a combination of automated and manual*

techniques to increase coverage, with the only exceptions being an exclusive reliance on automated techniques due to lack of security expertise, and on only manual techniques due to expertise/comfort with the same.

Impact due to Unsound vulnerability detectors: We asked participants how their software or service would get impacted in case there was a soundness issue vulnerability detectorsthey use. Interestingly, almost all practitioners expressed that *even in the case of flaws of vulnerability detectors, their applications would be moderately impacted at most*, explaining that they do not entirely depend on these tools for ensuring security and instead *rely on multiple tools and/or manual reviews*.

The few participants who shared that they would be significantly affected were either involved with tool development, or were entirely dependent on vulnerability detectors. In other words, *practitioners take the impact of flaws in security tools lightly as they use multiple tools and/or manual analysis techniques to overcome limitations*.

Table A.7: Abridged version of the Semi-structured Interview Guide

<i>Section A: Participants, Projects, and Organizations</i>
<p>To get started, can you tell us about the type or domain of software you primarily develop?</p> <p>How would you describe your target client for software? Is it general people, government, or other software firms?</p> <p>How did you get to learn about security? Does your company arrange training/workshops for you? Or self-learning?</p> <p>What is important in terms of security in terms of your product/software?</p> <p>What potential threats do you consider that may compromise the security of system?</p> <p>Are these the threat assumptions you normally consider in the domain you work on/at work?</p>
<i>Section B: Organization and Security</i>
<p>Do you remember being constrained by any factors, such as Deadline/Time, Requested Features, Dependencies, or others, when programming that may have affected/compromised security guarantees?</p> <p>How would you describe the software development process you follow?</p> <p>Do you remember your organization's existing coding standards, national regulations or any other software development process aspects having any influence on security guarantees?</p> <p>Have you implemented security functionalities through in-house development instead of relying on third-party libraries? What situation necessitated this?</p> <p>Between using third-party libraries for security-sensitive functionalities and implementing security-specific features on your own, which one do you prefer?</p> <p>Can you tell us more about the team you work with?</p> <p>Can you tell us about your team structure and security specific functions/components?</p> <p>Do you write test cases specifically for covering/testing security-related requirements? Can you give us an example?</p> <p>What are the consequences if the security requirements in your software are not met?</p>
<i>Section C: Organizational Context of vulnerability detector</i>
<p>Why do you favor using vulnerability detectors in your organization? What events led to this decision?</p> <p>Where does the vulnerability detector come in the Software Development Life Cycle (SDLC) you follow? Can you walk us through the process?</p> <p>Do all (security-related) team members know/receive training about the vulnerability detectors that you use?</p> <p>Are there generally any vulnerability detector-specific requirements from the user/customer?</p> <p>Can you please walk us through the process of selecting such a security focused tool?</p> <p>How are vulnerability detectors helpful for Agile/Scrum processes?</p> <p>Can you tell us more about the events which influenced you in becoming a vulnerability detector user instead of focusing on being a manual technique based user?</p> <p>How much is generally the cost in dollar value for licensing and/or using vulnerability detectors?</p> <p>You have mentioned that your organization relies more on the vulnerability detectors over Manual Techniques (or vice versa). Why is that?</p> <p>How do you generally handle security bugs in product?</p>
<i>Section D: Expectations from vulnerability detector</i>
<p>Consider the following statement: "When using an automated code review/scanner, a static tool should be capable of reporting all the issues in the code as far as static analysis allows". What is your opinion regarding this statement?</p> <p>"When using an automated code review/scanner, a static tool should only show results it is 100% certain about, even if it means it may miss a few potential issues", What is your opinion regarding this statement?</p> <p>Depending on the difficulty/nature of vulnerable code issues, some issues might be more difficult than others to detect by tools. Therefore, would you consider tools that are not perfect (<i>i.e.</i>, may miss some vulnerabilities) to still be acceptable to use? What is your opinion regarding this?</p> <p>You mentioned that you prefer FN/FP over FN/FP; Do you think this is purely because of the kind of software you work on, or do you think it is shared in the general developer community or in <type> developer community?</p> <p>How would you describe your overall impression when using security analysis tools for analyzing custom implemented security features?</p> <p>Does the tool clearly present detected security vulnerabilities, provide any explanations, link detected security vulnerabilities to known examples? Anything else you prefer these tools should have/report that is currently not available?</p>
<i>Section E: Impact of Unsound/Flawed vulnerability detector</i>
<p>Have you ever been in a situation where there was a vulnerability in your software, which should've been detected by a vulnerability detector but was not? How did you handle it?</p> <p>If in case your software has a security issue which was not found due to buggy vulnerability detector, how do you handle the consequences?</p> <p>"Just because a tool report states that there are no security errors does not mean the software is secure, since the tool itself may be buggy". Can you please elaborate on your opinion regarding this statement?</p> <p>Do you expect the vulnerability detector to catch everything?</p> <p>What happens if you find something that a vulnerability detector should catch, but does not? Do you report it to the vulnerability detector developers?</p> <p>Have you encountered any situation where any developer tried to evade vulnerability detector security checks by abusing flaws?</p> <p>If you ever reported a problem to vulnerability detector developers, did you ever get a response and a follow-up fix to the issue that you reported (with example)?</p> <p>(Previous context) What role do you think fuzzing tools play in comparison to vulnerability detectors here? Do you think fuzzing tools can replace vulnerability detectors?</p>
<i>Section F: Challenges and Improvements</i>
<p>Have you ever considered designing and using an in-house vulnerability detector? What limitations of existing tools motivated you to do so?</p> <p>If you were given unlimited resources to fix/create the perfect vulnerability detector, what issue would you address before anything else?</p> <p>Do you have any kind of final thoughts or anything that you would like to follow up on?</p>

VITA

Amit Seal Ami

Amit Seal Ami is a Ph.D. Candidate in the Department of Computer Science at William & Mary, advised by Dr. Adwait Nadkarni and Dr. Denys Poshyvanyk. His work has led to the discovery of over 20 critical flaws in highly popular security tools used by thousands of developers in the industry and has had an impact on the security of most consumer software used by people, be it mobile apps on our phones, or cloud services that enable them. His research has been published at highly selective venues in security and SE, such as IEEE S&P (2022, 2024), USENIX Security (2024), FSE (2023), ACM TOPS (2021), and ICSE (2021), and has been supported by the CoVA CCI Dissertation Fellowship. His work has been recognized through the Distinguished Paper Award at the IEEE S&P 2024 and the Stephen K. Park Gradu Graduate Research Award 2025. He is also a COVES'22 Policy Fellow and worked with the Joint Commission on Technology and Science, Commonwealth of Virginia, to help improve existing security policies for information technology in the state. Previously, he received his Bachelor's and Master's in Software Engineering from the Institute of Information Technology, University of Dhaka, Bangladesh, where he later developed an outreach program for students to connect with the W&M CSCI faculty.

For more information, please visit: <https://amitsealami.com>