Human Factors in Secure Software Development

Dissertation
zur Erlangung des Doktorgrades der
Naturwissenschaften
(Dr. rer. nat.)

dem Fachbereich Mathematik und Informatik
der Philipps-Universität Marburg
(Hochschulkennziffer 1180)
vorgelegt von

YASEMIN ACAR
geboren in Hannover

Hannover, 2021

Vom Fachbereich Mathematik und Informatik der
Philipps-Universität Marburg als Dissertation am
29.03.2021 angenommen.

Erstgutachter: Prof. Dr. Bernd Freisleben

Zweitgutachterin: Prof. Michelle L. Mazurek, PhD

Tag der mündlichen Prüfung: 29.03.2021

Erklärung

Ich versichere, dass ich meine Dissertation

„Human Factors in Secure Software Development“

selbstständig und ohne fremde Hilfe angefertigt, mich dabei keinen anderen als den von mir ausdrücklich bezeichneten Quellen und Hilfen bedient und alle vollständig oder sinngemäß übernommenen Zitate als solche gekennzeichnet habe. Die Dissertation wurde in der vorliegenden oder einer ähnlichen Form noch bei keiner anderen in- oder ausländischen Hochschule anlässlich eines Promotionsgesuchs eingereicht und hat noch keinen anderen Prüfungszwecken gedient.

(Ort/Datum) (Unterschrift mit Vor- und Zuname)

Summary

While security research has made significant progress in the development of theoretically secure methods, software and algorithms, software still comes with many possible exploits, many of those using the human factor. The human factor is often called "the weakest link" in software security. To solve this, human factors research in security and privacy focus on the users of technology and consider their security needs. The research then asks how technology can serve users while minimizing risks and empowering them to retain control over their own data. However, these concepts have to be implemented by developers whose security errors may proliferate to all of their software's users. For example, software that stores data in an insecure way, does not secure network traffic correctly, or otherwise fails to adhere to secure programming best practices puts all of the software's users at risk. It is therefore critical that software developers implement security correctly. However, in addition to security rarely being a primary concern while producing software, developers may also not have extensive awareness, knowledge, training or experience in secure development. A lack of focus on usability in libraries, documentation, and tools that they have to use for security-critical components may exacerbate the problem by blowing up the investment of time and effort needed to "get security right". This dissertation's focus is how to support developers throughout the process of implementing software securely.

This research aims to understand developers' use of resources, their mindsets as they develop, and how their background impacts code security outcomes. Qualitative, quantitative and mixed methods were employed online and in the laboratory, and large scale datasets were analyzed to conduct this research.

This research found that the information sources developers use can contribute to code (in)security: copying and pasting code from online forums leads to achieving functional code quickly compared to using official documentation resources, but may introduce vulnerable code. We also compared the usability of cryptographic APIs, finding that poor usability, unsafe (possibly obsolete) defaults and unhelpful documentation also lead to insecure code. On the flip side, well-thought out documentation and abstraction levels can help improve an API's usability and may contribute to secure API usage. We found that developer experience can contribute to better security outcomes, and that studying students in lieu of professional developers can produce meaningful insights into developers' experiences with secure programming. We found that there is a multitude of online secure development advice, but that these advice sources are incomplete and may be insufficient for developers to retrieve help, which may cause them to choose un-vetted and potentially insecure resources.

This dissertation supports that (a) secure development is subject to human factor challenges and (b) security can be improved by addressing these challenges and supporting developers. The work presented in this dissertation has been seminal in establishing human factors in secure development research within the security and privacy community and has advanced the dialogue about the rigorous use of empirical methods in security and privacy research. In these research projects, we

repeatedly found that usability issues of security and privacy mechanisms, development practices, and operation routines are what leads to the majority of security and privacy failures that affect millions of end users.

Zusammenfassung

Obwohl in der IT-Sicherheitsforschung signifikante Fortschritte in der Erforschung und Entwicklung theoretisch sicherer Methoden, Software und Algorithmen gemacht wurden, ist Software in der Praxis oft von Sicherheitsschwachstellen betroffen. Darunter ist der „Faktor Mensch“ ein häufiger Angriffspunkt, der oftmals auch als „schwächstes Glied“ in der Softwaresicherheit bezeichnet wird. Um diese Problematik anzugehen, beschäftigt sich die Forschung im Bereich „Faktor Mensch in der Sicherheits- und Privatssphäreforschung“ mit Nutzer:innen von Technologien, sowie mit ihren Ansprüchen an IT-Sicherheit. Die Forschung erörtert, auf welche Weise Technologien Nutzer:innen unterstützen können, während Risiken minimiert werden und Nutzer:innen die Kontrolle über ihre Daten behalten.

Die in der Forschung entwickelten Sicherheitslösungen sollen von Entwickler:innen umgesetzt werden, deren sicherheitskritische Programmierfehler sich auf alle Nutzer:innen der entwickelten Software übertragen. Beispielsweise setzt Software, in welcher Daten unsicher gespeichert, Netzwerkverbindungen unsicher aufgebaut oder andere Standards der sicheren Softwareentwicklung nicht eingehalten werden, alle ihre Nutzer:innen Sicherheitsrisiken aus. Daher ist es von entscheidender Wichtigkeit, dass Entwickler:innen Softwaresicherheit korrekt implementieren. Die Sicherheit steht bei Entwickler:innen jedoch selten im primären Fokus. Außerdem fehlen ihnen häufig Bewusstsein, Wissen, Ausbildung und Erfahrung im Bereich der sicheren Softwareentwicklung. Mangelnde Benutzbarkeit von Softwarebibliotheken, -dokumentationen und unterstützenden Tools, die für sicherheitskritische Entwicklung genutzt werden, kann das Problem verschlimmern, da sie die für sichere Softwareentwicklung notwendige Zeit und Arbeit vervielfachen.

Diese Dissertation befasst sich mit der Frage, wie Entwickler:innen im Prozess der sicheren Softwareentwicklung unterstützt werden können.

Ziel dieser Forschung ist es, zu verstehen, wie Entwickler:innen Ressourcen nutzen, um (sicher) zu entwickeln, und wie ihre Haltungen und Denkweisen sowie ihr Hintergrund und ihre Erfahrungen die Softwaresicherheit beeinflussen. Um diese Forschung durchzuführen, wurden qualitative, quantitative und gemischte Methoden online und im Labor angewandt sowie große Datensätze untersucht.

Ergebnis dieser Forschung ist, dass die Informationsquellen, die Entwickler:innen während des Programmierens benutzen, zu Sicherheit und Schwachstellen im Programmcode beitragen können. Verglichen mit der Nutzung offizieller Dokumentation, begünstigt das Kopieren und Einfügen von Codefragmenten aus Internetforen zwar die schnellere Programmierung funktionierenden Codes, führt jedoch häufig zu Schwachstellen. Ebenso zeigen wir, dass schlechte Benutzbarkeit, unsichere, möglicherweise veraltete Standardwerte sowie wenig hilfreiche Dokumentationen kryptographischer Bibliotheken zu unsicherem Programmcode beitragen. Erfreulicherweise unterstützen gut durchdachte Dokumentationen und Abstraktionslevel die Benutzbarkeit von Softwarebibliotheken, und somit auch ihre sichere Benutzung. Außerdem zeigen wir, dass Studien, die Informatikstudent:innen anstelle von Entwickler:innen als Proband:innen nutzen, aussagekräftige Forschungsergebnisse

über das Verhalten von Entwickler:innen im Prozess der sicheren Softwareentwicklung liefern können. Weiterhin zeigten unsere Untersuchungen, dass es online eine breite Fächerung an Ratschlägen für die sichere Softwareentwicklung zu finden gibt, dass diese Informationsquellen jedoch häufig unvollständig sind und als effektive Hilfestellung für die sichere Softwareentwicklung ungeeignet sein können. Dieses kann Entwicker:innen dazu verleiten, inoffizielle, nicht überprüfte und daher potentiell unsichere Informationsquellen zur Unterstützung heranzuziehen.

Diese Dissertation belegt, dass (a) sichere Softwareentwicklung den Herausforderungen des „Faktors Mensch“ unterliegt und dass (b) Softwaresicherheit verbessert werden kann, indem diese Herausforderungen adressiert und Entwickler:innen unterstützt werden. Die Forschungsarbeit, die in dieser Dissertation vorgestellt wird, war maßgeblich daran beteiligt, den Forschungsbereich „Faktor Mensch in der Softwaresicherheit“ in der Sicherheits- und Privatsphäreforschung zu etablieren und hat den wissenschaftlichen Diskurs über den rigorosen Einsatz empirischer Methoden in der Sicherheits- und Privatsphäreforschung vorangetrieben. In diesen Forschungsprojekten wurde wiederholt festgestellt, dass Benutzbarkeitsprobleme von Sicherheits- und Privatsphäremechanismen sowie von Entwicklungsprozessen und -verhaltensweisen zum Großteil der Sicherheits- und Privatssphäreprobleme führen, die Millionen von Nutzer:innen betreffen.

Foreword

The research in this dissertation was made possible by the support of my advisors, mentors, co-authors, colleagues, friends and family.
I am grateful to my family for getting me excited about reading and math early, setting me on a hopefully lifelong path of learning. I dedicate this dissertation to my kind and brilliant sister. Anything I can do, you can do, too.
I would like to thank Bernd Freisleben for shepherding this dissertation across the finish line; Michelle L. Mazurek for advising, collaborating, mentoring, and sharing R scripts, coffee, sushi and salads with me; Sascha Fahl, for everything; Matthew Smith, for introducing me to this research; Simson L. Garfinkel, for mentoring, collaborating, lunch, coffee, team workouts, and urging me to finish this dissertation *today*; Angela Sasse for encouragement, many great discussions, and paving the way; Christian Stransky, for sharing the joys and horrors of many late nights and deadlines, as well as for latex formatting and tech support, both within and outside of the scope of this dissertation (I guess you can always re-use our bibliography?); my colleagues and collaborators Dominik Wermke, Nicolas Huaman, Marten Oltrogge, Sabrina Amft, Niklas Busch, Harjot Kaur, Johanna Schrader, Lea Gröber, Alex Krause, Peter Leo Gorski, Karoline Busse, Sven Bugiel, and others for many fun and productive hours; Andreas Fahl, Lara Acar, Elisabeth Stelling, Sarah Schrader and Jan Brockmann for proofreading (all remaining typos and incorrectly-placed commas are mine alone); my gracious and kind hosts at NIST for a lovely research visit, including Julie Haney and Mary Theofanos; friends and awesome researchers in the SOUPS community, who emailed me and cheered me up at the best and worst times, including Mary Ellen Zurko; awesome people in the Security and Human Behavior community, who helped expand my horizon; collaborators and friends, including Brad Reaves and Adam Aviv, who have supported, discussed and encouraged in many ways.

List of Abbreviations

ACM	**A**ssociation for **C**omputing **M**achinery
AIC	**A**kaike **I**nformation **C**riterion
AOSP	**A**ndroid **O**pen **S**ource **P**roject
API	**A**pplication **P**rogramming **I**nterface
App	**App**lication
CBC	**C**ipher **B**lock **C**haining
CD	**C**ompact **D**isk
CFB	**C**ipher **F**eed**b**ack
CFG	**C**ontrol **F**low **G**raph
CTR	**Coun**t**er** Mode
CLMM	**C**umulative **L**inked **M**ixed **M**odels
CPU	**C**entral **P**rocessing **U**nit
DOM	**D**ocument **O**bject **M**odel
DRM	**D**igital **R**ights **M**anagement
ECB	**E**lectronic **C**ode **B**ook
HCI	**H**uman **C**omputer **I**nteraction
HTML	**H**yper **T**ext **M**arkup **L**anguage
HTTP	**H**yper **T**ext **T**ransfer **P**rotocol
ICC	**I**nter **C**omponent **C**ommunication
IDE	**I**ntegrated **D**evelopment **E**nvironment
IEEE	**I**nstitute of **E**lectrical and **E**lectronics **E**ngineers
IPC	**I**nter **P**rocess **C**ommunication
IRM	**I**nlined **R**eference **M**onitoring
JSON	**J**ava**S**cript **O**bject **N**otation
MITMA	**M**an **I**n **T**he **M**iddle **A**ttack
NDK	**N**ative **D**evelopment **K**it
NIST	**National** **I**nstitute of **S**tandards and **T**echnology
OS	**O**perating **S**ystem
OWASP	**O**pen **W**eb **A**pplication **S**ecurity **P**roject
PGP	**P**retty **G**ood **P**rivacy
Q&A	**Q**uestion **&** **A**nswer
ROM	**R**ead **O**nly **M**emory
SDK	**S**oftware **D**evelopment **K**it
SELinux	**S**ecurity **E**nhanced **Linux**
SO	**S**tack **O**verflow
SQL	**S**tructured **Q**uery **L**anguage
SSL	**S**secure **S**ocket **L**ayer
SUS	**S**ystem **U**sability **S**cale
TLS	**T**transport **L**ayer **S**ecurity
UI	**U**ser **I**nterface
UID	**U**nique **Id**entifier
URL	**U**niform **R**esource **L**ocator
USB	**U**niversal **S**erial **B**us

UX	**U**ser **E**xperience
VPN	**V**irtual **P**rivate **N**etwork
XML	**E**xtensible **M**arkup **L**anguage

Contents

Chapter 1

Introduction

This dissertation is based on six of my previous publications, all of which were written with me as the main author. This introduction is updated, expanded and adapted from a comprehensive overview and research agenda, previously published as the conference paper "You are not your developer, either: A research agenda for usable security and privacy research beyond end users" at 2016 IEEE Cybersecurity Development. This paper was a joint effort with my advising and supervising co-authors Sascha Fahl and Michelle L. Mazurek; we jointly reviewed the literature, developed future research directions and co-wrote the paper; therefore, this chapter uses the academic "we".

While security researchers have developed methods and tools that should improve software security, data breaches and exploits are becoming more frequent, with both personal and corporate data being breached [171]. Even though security research has advanced cryptographic algorithms, access control and memory-safe applications that have been shown to offer provably strong, maybe even perfect security, and should be able to deliver protection from most of these attacks, the rate of cyber attacks continues to increase [232]. This huge gap between strong theoretical security offered by security mechanisms and low actual security in practice can partially be explained by a lack of consideration of human factors when developing these solutions. Security mechanisms may be hard to use, may be interfering with users' priorities, or may make unrealistic assumptions about users' security knowledge. The security and privacy community has worked on improving the usability of security tools and interfaces created for end users for more than 20 years.

Encrypted messaging is one interesting example: while asymmetric encryption dates back to the 1970s [68, 209] and PGP [297] was introduced in 1991, few people encrypt their emails. This is despite strong incentives like present day nation-state surveillance in industrialized and digitalized countries. However, even with strong incentives, like present day nation-state surveillance, almost no one uses end-to-end email encryption. However, since a widely used messaging application started end-to-end encrypting their messages by default at no additional effort for users, a large number of encrypted messages are now sent daily [11].

Whitten and Tygar's seminal 1999 paper analyzed email encryption as a usability problem [272], helping to establish a new research field, the usable security and privacy community. This community has since identified human and social science factors, such as economics, cognitive biases, access and structure of information as well as mindsets, as major contributors to users failing to effectively use existing security and privacy mechanisms. The usable security and privacy community aims to understand how considering human factors can better protect users' security and privacy; this is done by improving the usability of existing security mechanisms and offering guidelines for designing new mechanisms that are usable from the start. Within the usable security and privacy community, topics that have been heavily

researched include email encryption [83, 98, 212, 213, 224, 272], passwords and alternative authentication mechanisms [33, 81, 117, 137, 157, 236], security-relevant user interactions such as warning messages and security indicators [10, 37, 92, 219, 241, 269], and privacy controld and behaviors [7, 8, 14, 58, 208].

While progress has been made in improving end users' adherence and sometimes even comprehension of security-critical issues, a key constituency has thus far been understudied: Software developers make security and privacy decisions that have a huge impact on end-user (and therefore overall ecosystem) security, and they suffer from similar comprehension and adherence problems to end users. Although usable security and privacy research focusing on developers is still in an early stage, preliminary results illustrate a common theme: Developers are regular users of security and privacy mechanisms (e.g., security APIs, protocols, and tools), but are by no means security experts [22, 85]. We argue for a systematic approach to studying developers within the security ecosystem. While developer-usability studies targeting specific security tools and APIs are becoming more common [35, 280, 285], topics are fragmented and quality research norms have not yet been firmly established. We argue for systematizing future research on usable security [1] for developers, including working to validate promising research methods and identifying key areas of focus.

1.1 Lessons Learned from Studying Usable Security for End Users

We briefly discuss key lessons learned from more than 20 years of research into usable security for end users, and how these lessons can apply in the developer space.

1.1.1 You Are Not Your User

Plentiful research has demonstrated that unusable end-user security tools and interfaces frequently arise when the developers of these tools make unfounded assumptions about what the intended users know and understand. Examples include everything from encryption tools that expect users to understand the difference between encryption and signing [272], to browser warnings and app permission descriptions that use too much security jargon [241], to expecting users to understand the importance of software updates [80, 156, 175, 254]. In each of these cases, security experts have expected end users to know and care about security, perhaps because of assumed similarity bias, in which people often assume that everyone is similar to themselves and the people they know [123].

This lesson applies even more strongly when considering security tools and APIs used by developers. Because developers by definition have some level of technical expertise, it is easy for security experts to mistakenly believe that developer-users also understand security, or that expert tools need not be designed with usability in mind. It seems likely this fallacy is at the root of unusable cryptography APIs, as well as difficult-to-interpret outputs from bug-finding tools.

In the case of end users, these problems have been mitigated somewhat by reminding tool developers to consider the different needs and attitudes of end users,

[1] For simplicity, throughout this paper we refer to *security and privacy* as *security*. We find that privacy-preserving or -enhancing behavior often requires the use of secure mechanisms, while good security practice often protects privacy. The research techniques and approaches we discuss generally apply well for both.

and by explicitly evaluating usability rather than making assumptions about what is usable [207, 269]. We believe similar solutions can be helpful when building security tools for developers; we discuss potential targets for usability evaluation in Section 1.2.3.

1.1.2 Security Is A Secondary Concern

The usable-security field has firmly established that security is a secondary concern for end users; when it gets in the way of a user's primary goal, security becomes an annoyance to be worked around or ignored. As examples, end users adopt insecure password practices when requirements become too onerous [9, 288] and ignore security icons and warnings when they are motivated to proceed to their goal [144, 219].

This concept applies equally to developers, who have priorities—functional correctness, time to market, maintainability, economics, compliance with other corporate policies—that sometimes appear to conflict with security and are often more salient [270]. For end users, the usable-security community frequently recommends taking users out of the loop as much as possible [61], such as by making updates automatic, choosing secure defaults, and forcing browsers to use HTTPS. When removing the user from the loop is infeasible, the community has often emphasized *opinionated design*, also called nudging or soft paternalism, which encourages users to make more secure choices even if they do not entirely understand the situation. For example, browser certificate warnings are designed to discourage click-through [86]. We discuss ideas for applying these approaches to developers in Section 1.2.4.

1.1.3 More is Not Always Better

A third key lesson from usable security for end users is that simply adding more and more security advice and recommendations is not a viable solution. Piling on advice can overwhelm users and lead them to give up on taking any steps to improve security; similarly, encountering too many warnings that don't lead to actual harms causes habituation and disengagement. While the usable-security community continues to struggle with this problem, recently researchers are acknowledging the overabundance of unhelpful advice and even advocating rollback of some overzealous policies, such as password expiration [52, 60, 121]. This overabundance of security advice has shifted the problem for end users to choosing which information sources they trust or rely on the most [206].

Related issues are beginning to be seen in the advocacy of secure development; for example, the proliferation of new, sometimes incompatible, encryption libraries claiming both security and usability with little or no empirical evaluation. While the end-user security community has not identified any comprehensive solution to this problem, we encourage the developer security community to bear in mind that simply asking developers to do more and more in the name of security is unlikely to help and may even exacerbate the problem.

1.2 A Research Agenda for Usable Security for Developers

We believe that thus far, usable security for developers has been a critically underinvestigated area. Recently, the topic has begun to receive more attention, and we expect that in the near future many researchers will address it. In this section, we

lay out a high-level research agenda covering what we believe are the most important needs in this area. We organize our suggestions into four areas to investigate: how best to conduct usable security research with developers; how developers think about security in the context of their needs and priorities; how usable current security tools and APIs are and where they fall short; and how to build more usable tools and paradigms in the future.

1.2.1 Methodology and Ecological Validity

One major concern with studying usable security for developers is ecological validity: whether or not the circumstances of a study accurately reflect the real world [124]. While this is a challenge for most user studies, it's especially challenging when targeting usable security for developers, for several reasons. Because security is a secondary concern, asking users about it directly may not effectively reflect realistic circumstances, in which developers may not be thinking about security or in which other priorities may outweigh security concerns. In addition, recruiting professional developers to study can be challenging: depending on the researcher's geographical area, there may not be many developers locally available, and those who are may be too busy to attend studies. The hourly rates these highly specialized people are typically paid will often exceed the researcher's available budget. Finally, real-life development tasks are complicated and may be difficult to simulate in a study environment.

To address this challenge, we need methodological research investigating *how* to study developers' security behavior. One critical question is whether and in what circumstances computer science students, who are often studied out of convenience, can effectively substitute for professional developers. In our work examining how information resources impact developers' decision making, we asked both students and professionals to complete four time-limited, security-related programming tasks. We found that professionals outperformed students in functional correctness, but were no more secure [4] (Chapter 3 in this dissertation). While this result is intriguing, further investigation is needed. Is this result reproducible with other security tasks and environments? What constitutes a professional? How do professionals from big and small companies differ, and how do they compare to graduate and undergraduate students from different universities? Are lab studies necessary, or can online studies be useful? To answer these questions, controlled comparison studies are needed; Chapter 5 includes one such study, focused on recruitment from one online open source platform [6].

Researchers should also investigate what kind of study tasks work best for evaluating security tools and behaviors; to do this, researchers should aim to compare controlled studies with field observations. We have previously applied similar methods to evaluate ecological validity for password studies [81, 157], while other researchers have addressed ecological validity, e.g., for studies of security indicators [219]. We can also learn from the software engineering community's work investigating developers and their tools and behaviors in non-security domains [44, 72, 220, 239].

Key research questions:

- Which recruitment strategies provide representative samples of real-world developers efficiently?
- Which study and task designs are most appropriate to measure developers' motivations, attitudes and knowledge?

1.2.2 Understanding Developers' Motivations, Attitudes, and Knowledge

In a landmark 1999 article, Adams and Sasse challenged the conventional wisdom that users reject security behaviors—in this case password policies—due primarily to laziness or carelessness [9]. Instead, they argued, misbehavior stemmed primarily from misunderstandings, competing priorities, and challenging interfaces. A similar consensus is starting to emerge with respect to developers' security behaviors: although historically developers have been seen as "experts" in contrast to less knowledgeable end users, many (most) developers are not experts in security, and make errors through misunderstandings and difficult-to-use interfaces. In addition, developers have priorities—such as adding functionality, optimizing the end-user experience, reducing time-to-market, and reducing development costs—that often appear to be in conflict with best security practices. Before we can develop better tools, interfaces, and educational interventions to promote secure development, we must investigate what developers understand about security and how they view secure development in the context of their overall goals.

Acquiring this understanding can be approached in several ways. We can use qualitative interviews and quantitative surveys to ask developers directly about their security knowledge, attitudes, and decision-making processes. This parallels Adams and Sasse's work [9], as well as many subsequent papers evaluating end-user security attitudes and behaviors [54, 206, 228, 254, 267]. Balebako et al. used this approach to investigate how mobile app developers make privacy-relevant decisions, finding that lack of awareness and lack of resources contribute to poor privacy decisions [22]. In the same study, the authors report on some use of third-party security tools considered more secure than homemade implementations. We expect that a similar study focused explicitly on security attitudes and behaviors would find related barriers and more in-depth analysis of why and how third-party security tools are and are not used.

While studies in which participants are asked explicitly about their attitudes and behaviors provide valuable data and important context, self-reporting is inherently limited by human recall and by well-known psychological biases [134, 252]. To get a complete picture, therefore, we must supplement these findings with measurements of actual behavior. This can be obtained via in-situ observational studies (e.g., following developers to design meetings, observing their work in progress, etc.), and by field or diary studies in which developers report on their security-relevant decisions as they make them. This might include observing decisions like which libraries to use, what security threat model is appropriate, and whether to use, e.g., bug-finding or fuzzing tools. While these studies can be complicated, expensive, and time-consuming, they provide rich data with strong validity that often cannot be obtained any other way.

Key research questions:

- What motivates developers to use secure mechanisms and concepts, and how can we use this to improve the status quo?
- What prevents developers from adhering to secure recommendations, and how can we counter this?
- Which information sources do developers turn to and trust, and how can we use this to improve security?
- Where do developers lack knowledge, and how can we either provide them with secure information sources or secure their software without requiring security education?

1.2.3 Investigating the Status Quo

In addition to understanding developers' knowledge and attitudes, we must investigate how existing APIs, documentation, and tools encourage or discourage good security behaviors. By identifying which tools work well and which fail, and why, we can improve existing tools and build new ones that are more likely to be effective.

Existing tools and APIs can be evaluated via field and measurement studies that capture security behaviors, implementations, and mistakes across a broad swathe of software. For example, several studies have examined the use of TLS and cryptography more generally in mobile apps and identified common pitfalls and errors [70, 85, 99, 205]. We propose further measurements, such as examining how insecure code propagates on GitHub, or studying how popularity of different security libraries correlates with common errors. These kinds of measurements can potentially be extended by contacting involved developers for follow-up interviews concerning how libraries were chosen or how errors were made.

While field measurements provide a valuable large-scale look at how tools and APIs are used in practice, they do not allow researchers to isolate and test specific hypotheses. Thus, we also recommend controlled experiments to measure how concrete factors affect developers' decisions. For example, in recent work we examined how using Stack Overflow compared to official documentation affected the security of code Android developers wrote in response to short programming tasks [4], Chapter 3. We also conducted an experiment comparing how different cryptography APIs affect the code developers write [3], Chapter 4. Researchers should also measure the usability of existing bug-finding and fuzzing tools to identify problems and pain points; these studies could be modeled on investigations of usability for security tools used by end users, such as [83, 92, 98, 212, 213, 219, 224, 272].

In addition to field studies and lab measurements, expert review (including, e.g., *cognitive walkthroughs* and *heuristic evaluations*) of tools and APIs for usability can provide valuable feedback to their authors with less time and expense. We propose that researchers evaluate groups of related APIs and tools to provide clear evidence of the benefits and drawbacks of each. Expert reviews are frequently used in HCI generally and in usable security specifically [38, 56, 79, 272].

Key research questions:

- How well do current APIs, documentation, and tools support secure behavior?
- In which ways should future APIs, documentation, and tools be designed to encourage secure behavior?
- Which of APIs, documentation, and tools has the most promising impact on security; where should we place the focus of our research?

1.2.4 Improving the Status Quo

Here, we make recommendations for improvements to APIs, tools, and other developer resources.

Usable security APIs The software engineering community has developed guidelines for designing usable APIs and tools generally [34, 57, 120, 165, 173], and security researchers have considered API usability at a high level as well [115, 278]. Guidelines from all these sources should be synthesized and extended to provide concrete objectives for security APIs. We developed a framework for measuring the usability of security APIs, and applied this framework to evaluate security APIs in the wild [3], Chapter 4. Further work should be done both to make existing APIs more usable—including via better documentation—as well as to introduce new APIs that balance security and usability.

Secure, usable information resources We have shown that developers make insecure choices when the (usable) resource they turn to for help is offering quick but insecure fixes[4], Chapter 3. To address this, we advocate making official documentation (which already promotes security) more interactive and usable, and to introduce security monitoring to usable resources. More research is needed on how to best combine usability with security in developer resources.

Developer tool support Integrating tool support into developer environments can both raise security awareness and provide direct security feedback. For example, we developed an exemplar Android Studio plugin that applies static code analysis to help developers to turn insecure choices into more secure ones [172]. While developer support and IDEs that make developing faster and easier exist, no security tools are rarely in use.

Taking developers out of the loop We recommend removing developers from the security loop whenever possible. We have shown in the past that developers who implement custom SSL/TLS handling nearly always make insecure choices; in response, we suggested configurable TLS handling at the OS level [85, 176]. In a similar vein, we recommend further research aimed at moving security management and security-critical decisions from apps to the OS and framework levels whenever possible. This includes but is not limited to automatic security library updates, or automatic permission requests on Android. Not only could this reduce developers' opportunities to make errors, but it is also compatible with the tendency to prioritize reducing development time and effort over security correctness. Research is needed

to identify cases where this is possible as well as to suggest effective ways to remove developers from the security loop without overly restricting functionality.

This dissertation presents a systematic approach to studying software developers within the security ecosystem. While studies with developers that target specific security tools and APIs are becoming more common [35, 280, 285], topics are fragmented and quality research norms have not yet been firmly established. This research is challenging in many ways: the population is not well-understood so far, and these higher-skilled professionals are harder to recruit. Research that aims to support these professionals needs interdisciplinary research teams that understand the highly technical security problems can leverage human factor research methods to understand the actors' limitations and behaviors, then develop appropriate solutions.

This dissertation introduces promising directions for human factor security research with developers, including work to validate promising research methods and identifying key areas of focus. Mixed methods studies of developers using APIs and documentation are presented as well as the security outcomes for their code.

1.3 Thesis Statement

The purpose of this work is to explore the problem space of human factors in secure programming from a usable security and privacy perspective and provide guidance for future researchers and practitioners to build security mechanisms and tools for developers that are easier to use for non-security experts. Accordingly, the central thesis of this dissertation is:

Many developers are not security experts, but still use security tools, APIs and mechanisms to build computer systems. Therefore, unnecessarily complex and unusable security tools, APIs, documentation and IDEs lead to a large number of software vulnerabilities and data breaches. Considering software developers as human factors can lead to better software security and less data breaches than current tools, APIs, documentation and IDEs provide.

Recognition: *For the research that contributed to this thesis, I received the John Karat Usable Privacy and Security Student Research Award, which annually recognizes the best student in the usable privacy and security community* [2]*. The paper "You get where you're looking for: The Impact of Information Sources on Code Security", Chapter 3, won the National Security Agency's 2016 Best Scientific Cybersecurity Paper Competition* [3]*.*

1.4 Contributions

The research that contributed to this thesis has been published as conference papers. This section presents a summary of the the publications that contributed to this thesis, where they were published, and which author contributed in which way. As is common with collaborative research, this research would not have been possible without the significant contributions of the co-authors. Here, * denotes the main author, † denotes an author with significant contribution, and $^{+}$: denotes a supervising author with significant contribution. These publications have been mildly edited and contextualized where needed for inclusion in this thesis. The publications that contributed to this dissertation are the following:

[2] `https://www.usenix.org/conference/soups2020/karat-call-for-nominations`

[3] `https://cps-vo.org/node/39262`

You are not your developer, either: A research agenda for usable security and privacy research beyond end users

This paper contributed to the introduction of this dissertation. We identify research gaps and promising directions for future work in usable security and privacy for software developers and other expert users.
Authors: Yasemin Acar*, Sascha Fahl+, Michelle L. Mazurek+
Published at 2016 IEEE Cybersecurity Development, Acceptance Rate: 38%.
Contributions to the paper: *All authors reviewed the literature, developed future research directions and co-wrote the paper.*

Paper summary While researchers have developed many tools, techniques, and protocols for improving software security, exploits and breaches are only becoming more frequent. Some of this gap between theoretical security and actual vulnerability can be explained by insufficient consideration of human factors, broadly termed usability, when developing these mechanisms. In particular, security mechanisms may be difficult to use, may conflict with other priorities, or may assume more security knowledge than users possess. For almost 20 years, the usable security community has investigated how to improve the usability of security tools and interfaces aimed at end users. More recently, the community has begun to apply similar techniques in the context of improving security tools—such as APIs and bug-finding software—aimed not at end users but at developers, whose security errors are magnified across all users of their products. In this paper, we review key lessons learned from usable security for end users and consider how to apply them in the context of developers. We propose a research agenda aimed at developing a high-quality, comprehensive literature for usable security for developers, including: investigating how to conduct reliable research in this context; understanding developers' attitudes, knowledge, and priorities; measuring the status quo; and developing improved tools and interventions in the future.

SoK: Lessons Learned from Android Security Research for Appified Platform

This paper contributed to Chapter 2 of this dissertation, **Systematization of Android Security Research**. We survey the literature on Android security research, identify lessons learned for appified platforms and identify Android developers as a promising but under-researched population.
Authors: Yasemin Acar*, Michael Backes, Sven Bugiel*, Patrick McDaniel, Sascha Fahl*, Matthew Smith.
Published at 2016 IEEE Symposium on Security and Privacy, Acceptance Rate: 13%
Contributions to the paper: *Sascha Fahl, Sven Bugiel and I conducted the literature review and systematization, and wrote the paper. The remaining authors provided guidance.*

Paper summary Android security and privacy research has boomed in recent years, far outstripping investigations of other appified platforms. However, despite this attention, research efforts are fragmented and lack any coherent evaluation framework. We present a systematization of Android security and privacy research with a focus on the appification of software systems. To put Android security and privacy

research into context, we compare the concept of appification with conventional operating system and software ecosystems. While appification has improved some issues (e.g., market access and usability), it has also introduced a whole range of new problems and aggravated some problems of the old ecosystems (e.g., coarse and unclear policy, poor software development practices). Some of our key findings are that contemporary research frequently stays on the beaten path instead of following unconventional and often promising new routes. Many security and privacy proposals focus entirely on the Android OS and do not take advantage of the unique features and actors of an appified ecosystem, which could be used to roll out new security mechanisms less disruptively. Our work highlights areas that have received the larger shares of attention, which attacker models were addressed, who is the target, and who has the capabilities and incentives to implement the countermeasures. We conclude with lessons learned from comparing the appified with the old world, shedding light on missed opportunities and proposing directions for future research.

You get where you're looking for: The Impact of Information Sources on Code Security

This paper contributes to Chapter 3 of this dissertation, **On the Impact of Information Sources on Code Security**. We conduct a security developer study with Android developers to compare the impact of developer documentation on code security and show that documentation usability and secure code examples are crucial for secure software.
Authors: Yasemin Acar*, Michael Backes, Sascha Fahl+, Doowon Kim†, Michelle L Mazurek+, Christian Stransky*.
Published at 2016 IEEE Symposium on Security and Privacy, Acceptance Rate: 13%.
Contributions to the paper: *Christian Stransky, Sascha Fahl, Michelle L. Mazurek and I designed the online survey and evaluated it and designed the controlled experiment, conducted the experiment and evaluated the experiment. Doowon Kim helped in conducting the experiment. Sascha Fahl and I designed the API call confirmation study. Christian Stransky and Sascha Fahl conducted and evaluated the API call search. Christian Stransky, Doowon Kim, Michelle L. Mazurek, Sascha Fahl and I co-wrote the paper. We are grateful to Sven Bugiel for assisting in the controlled experiment, and Marten Oltrogge for assisting in the API call search.*
Recognition: *This paper was recognized as the 2016 Best Scientific Cybersecurity Paper by the National Security Agency (NSA). Based on the research in this paper, Google awarded Sascha Fahl a Faculty Research Award, on which I was the lead research student* [4].

Paper summary Vulnerabilities in Android code – including but not limited to insecure data storage, unprotected inter-component communication, broken TLS implementations, and violations of least privilege – have enabled real-world privacy leaks and motivated research cataloguing their prevalence and impact. Researchers have speculated that appification promotes security problems, as it increasingly allows inexperienced laymen to develop complex and sensitive apps. Anecdotally, Internet resources such as Stack Overflow are blamed for promoting insecure solutions that are naively copy-pasted by inexperienced developers.

In this paper, we for the first time systematically analyzed how the use of information resources impacts code security. We first surveyed 295 app developers who have published in the Google Play market concerning how they use resources

[4] `https://services.google.com/fh/files/blogs/v2_final_list.pdf`

to solve security-related problems. Based on the survey results, we conducted a lab study with 54 Android developers (students and professionals), in which participants wrote security- and privacy-relevant code under time constraints. The participants were assigned to one of four conditions: free choice of resources, Stack Overflow only, official Android documentation only, or books only. Those participants who were allowed to use only Stack Overflow produced significantly less secure code than those using, the official Android documentation or books, while participants using the official Android documentation produced significantly less functional code than those using Stack Overflow.

To assess the quality of Stack Overflow as a resource, we surveyed the 139 threads our participants accessed during the study, finding that only 25% of them were helpful in solving the assigned tasks and only 17% of them contained secure code snippets. In order to obtain ground truth concerning the prevalence of the secure and insecure code our participants wrote in the lab study, we statically analyzed a random sample of 200,000 apps from Google Play, finding that 93.6% of the apps used at least one of the API calls our participants used during our study. We also found that many of the security errors made by our participants also appear in the wild, possibly also originating in the use of Stack Overflow to solve programming problems. Taken together, our results confirm that API documentation is secure but hard to use, while informal documentation such as Stack Overflow is more accessible but often leads to insecurity. Given time constraints and economic pressures, we can expect that Android developers will continue to choose those resources that are easiest to use; therefore, our results firmly establish the need for secure-but-usable documentation.

Comparing the Usability of Cryptographic APIs

This paper contributes to Chapter 4 of this dissertation, **Comparing the Usability of Cryptographic APIs**. We investigate the impact of cryptographic API usability on code security in a security developer study with experienced Python developers and find that API simplicity is insufficient for secure outcomes, and that safe defaults and supportive documentation is needed.
Authors: Yasemin Acar*, Michael Backes, Sascha Fahl+, Simson L. Garfinkel+, Doowon Kim†, Michelle L. Mazurek+, Christian Stransky*.
Published at 2017 IEEE Symposium on Security and Privacy, Acceptance Rate: 13%.
Contributions to the paper: *Christian Stransky, Doowon Kim and Sascha Fahl conducted the preliminary survey of libraries; based on critera all authors discussed, we chose the libraries. Christian Stransky, Michelle L. Mazurek, Sascha Fahl and I designed the study. Simson L. Garfinkel and I conducted the literature review. Michelle L. Mazurek and I designed the questionnaire. Christian Stransky and Sascha Fahl conducted the experiment. Christian Stransky, Doowon Kim, Michelle L. Mazurek, Sascha Fahl and I evaluated the experiment. Christian Stransky, Doowon Kim, Michelle L. Mazurek, Sascha Fahl and I co-wrote the paper. Simson L. Garfinkel, Michelle L. Mazurek and Sascha Fahl supervised throughout.*
Recognition: *After its publication, this paper was invited for presentation at Real World Crypto 2018.*

Paper summary Potentially dangerous cryptography errors are well-documented in many applications. Conventional wisdom suggests that many of these errors are caused by cryptographic Application Programming Interfaces (APIs) that are too complicated, have insecure defaults, or are poorly documented. To address this

problem, researchers have created several cryptographic libraries that they claim are more usable; however, none of these libraries have been empirically evaluated for their ability to promote more secure development. This paper is the first to examine both how and why the design and resulting usability of different cryptographic libraries affects the security of code written with them, with the goal of understanding how to build effective future libraries. We conducted a controlled experiment in which 256 Python developers recruited from GitHub attempt common tasks involving symmetric and asymmetric cryptography using one of five different APIs. We examine their resulting code for functional correctness and security, and compare their results to their self-reported sentiment about their assigned library. Our results suggest that while APIs designed for simplicity can provide security benefits—reducing the decision space, as expected, prevents choice of insecure parameters—simplicity is not enough. Poor documentation, missing code examples, and a lack of auxiliary features such as secure key storage, caused even participants assigned to simplified libraries to struggle with both basic functional correctness and security. Surprisingly, the availability of comprehensive documentation and easy-to-use code examples seems to compensate for more complicated APIs in terms of functionally correct results and participant reactions; however, this did not extend to security results. We find it particularly concerning that for about 20% of functionally correct tasks, across libraries, participants believed their code was secure when it was not.

Our results suggest that while new cryptographic libraries that want to promote effective security should offer a simple, convenient interface, this is not enough: they should also, and perhaps more importantly, ensure support for a broad range of common tasks and provide accessible documentation with secure, easy-to-use code examples.

Security Developer Studies with GitHub Users: Exploring a Convenience Sample

This paper contributes to Chapter 5 of this dissertation, **Exploring a GitHub Sample for Security Developer Studies**. We perform a security developer study with GitHub users to explore sample parameters, and find that they represent a diverse population that vary strongly in their software development experience, and particularly in their security expertise.

Authors: Yasemin Acar*, Christian Stransky*, Dominik Wermke†, Michelle L. Mazurek+, Sascha Fahl+.

Published at Symposium on Usable Privacy and Security 2017, Acceptance Rate: 27%.

Contributions to the paper: *Christian Stransky, Michelle Mazurek and I designed the study. Christian Stransky, Sascha Fahl and I conducted the study. All authors evaluated and co-wrote the paper.*

Paper summary The usable security community is increasingly considering how to improve security decision-making not only for end users, but also for information technology professionals, including system administrators and software developers. Recruiting these professionals for user studies can prove challenging, as, relative to end users more generally, they are limited in numbers, geographically concentrated, and accustomed to higher compensation. One potential approach is to recruit active GitHub users, who are (in some ways) conveniently available for online studies. However, it is not well understood how GitHub users perform when working on security-related tasks. As a first step in addressing this question, we

conducted an experiment in which we recruited 307 active GitHub users to each complete the same security-relevant programming tasks. We compared the results in terms of functional correctness as well as security, finding differences in performance for both security and functionality related to the participant's self-reported years of experience, but no statistically significant differences related to the participant's self-reported status as a student, status as a professional developer, or security background. These results provide initial evidence for how to think about validity when recruiting convenience samples as substitutes for professional developers in security developer studies.

Developers need support, too: A survey of security advice for software developers

This paper contributes to Chapter 6 of this dissertation, **A Survey of Security Advice for Software Developers**. We survey web resources for developers, evaluate their usefulness and effectiveness in promoting security in practice, and identify important gaps in the current ecosystem.
Authors: Yasemin Acar*, Christian Stransky*, Dominik Wermke*, Charles Weir†, Michelle L. Mazurek, Sascha Fahl+.
Published at 2017 IEEE Cybersecurity Development
Contributions to the paper: *Christian Stransky, Dominik Wermke, Michelle L. Mazurek, Sascha Fahl and I designed the study. Christian Stransky, Dominik Wermke and Charles Weir conducted the coding. All authors evaluated and co-wrote the paper.*

Paper summary Increasingly developers are becoming aware of the importance of software security, as frequent high-profile security incidents emphasize the need for secure code. Faced with this new problem, most developers will use their normal approach: web search. But are the resulting web resources useful and effective at promoting security in practice? Recent research has identified security problems arising from Q&A resources that help with specific secure-programming problems, but the web also contains many general resources that discuss security and secure programming more broadly, and to our knowledge few if any of these have been empirically evaluated. The continuing prevalence of security bugs suggests that this guidance ecosystem is not currently working well enough: either effective guidance is not available, or it is not reaching the developers who need it. This paper takes a first step toward understanding and improving this guidance ecosystem by identifying and analyzing 19 general advice resources. The results identify important gaps in the current ecosystem and provide a basis for future work evaluating existing resources and developing new ones to fill these gaps.

1.5 Related and Concurrent Work

The work in this dissertation has supported that (a) secure software development is subject to human factor challenges and (b) software security can be improved by addressing these challenges and supporting software developers. While our studies were conducted, several other avenues were explored by other research groups, either concurrently or following our work. We published our research agenda for this

field in 2016 [5], Section 1.1, since then, many of our calls to action have been realized, both by works in this thesis, and by the broader research community. A summary of the work in this field has since been published: In 2018, Tahaei and Vaniea conducted a survey of 49 studies with developers as participants, identifying that research is being conducted into tools, mindsets and methodology, finding that privacy engineering and team efforts are currently understudied [243]. A thoughtpiece on how concepts from usability work with end users might or might not translate to work with developers was written by Pieczul et al. in 2017 [187], and discussed at the New Paradigms in Security Workshop, where the first ideas about human factors in security, by Mary Ellen Zurko, also originated. This dissertation explored the concept of human factors in secure software development. Initially, we systematized the field of Android security research and identified secure software development challenges [2], Chapter 2. This knowledge of security challenges in Android has helped inform our work with software developers. We presented a study on how information sources impact the ability to write functional, secure code [4], Chapter 3. Since then, Stack Overflow as a resource, and how developers interact with it, has been studied [147], finding that developers tend to select code snippets based on supplementary information, regardless of whether the example is secure. We found in 2017 that easy-to-implement suggestions in the software development environment, based on static code analysis, can help application developers fix possible insecure code as they type [172]. In 2018, we found that even with a relatively easy-to-use method for obfuscation built into Android Studio, obfuscation is not widely used and developers struggle to implement it beyond the basic use case [271]. This suggests that tooling can help catch some "low-hanging fruit" software security issues, but a more comprehensive approach may be needed to cover those cases that cannot be addressed automatically.

We compared the usability of cryptographic APIs and found that certain deficiencies in usability track with insecure programming outcomes [3], Chapter 4, which resulted in interesting discussions with some of the developers of some of the APIs we investigated. After this paper was published, Gorski et al. [110] developed a warning message that can be implemented by library developers, and is shown at compile time to prevent going forward with potentially insecure code. This is an exciting result, as it shows that those experts who are already knowledgeable about the libraries they write can make additional usable contributions to code security in projects that use their libraries. The same team conducted a focus group study with software developers to co-design said warning message for better acceptance [109], finding that developers have varying preferences depending on where in the process the message will be shown. Patnaik et al. further identified developers' "usability smells", common problems with cryptographic APIs [182]. While the software engineering community has investigated how developers choose APIs [141], it is unclear whether this transfers to cases relevant to software security. There is also no tried and tested comprehensive blueprint yet for how to make security APIs usable.

Assal and Chiasson found that general software and web developers vary in their adherence to the secure development lifecycle [17]. Tahaei et al. explored computer science students' mindsets, finding that their security mindsets somewhat resemble those of professional developers, and that security is rarely a priority, and quickly finding functional solutions by implementing pre-existing code is common [242]. Assal et al. surveyed software developers, finding that vulnerabilities may stem from a lack of organizational support [18]. Votipka et al. further explored developers' mindsets: They found that hackers and testers were two programmer

archetypes [261]. They also further investigates how developers create vulnerabilities [260] and developed a scale for secure development self-efficacy [259]. Thomas et al. find that application developers often interact with security via auditors, outsourcing security reviews. They suggest incentivizing developers to integrate security into development in the first place [250]. Poller et al. found that security training does not necessarily translate into secure behavior [189]. Haney at al. found strong security mindsets in organizations that develop cryptographic products, finding that those mindsets both develop from the top down, and stay with employees as a sense of identity [116]. This shows the important role of mindset, practice and education, which is promising, given that universities, where many developers obtain their education, are starting to include security, usability and combinations thereof in their coursework.

We hope that the results in this dissertation, and more broadly, in this field, will continue to help practitioners in implementing software securely, and inspire new research on how to support software developers in writing secure code.

Chapter 2

Systematization of Android Security Research

Disclaimer: *The contents of this chapter were previously published as part of the conference paper "SoK: Lessons Learned From Android Security Research For Appified Software Platforms", presented at the 2016 IEEE Symposium on Security and Privacy. This research was conducted as a team with my co-authors Sven Bugiel, Sascha Fahl, Matthew Smith, Patrick McDaniel and Michael Backes; this chapter therefore uses the academic "we". The idea and initial concept for the systematization came from myself and Sascha Fahl. Sascha Fahl, Sven Bugiel and I conducted the literature survey. We received input on the systematization from all other authors and co-wrote the paper for publication. This work reflects the state of early-2016 Android, as well as early-2016 research on appified ecosystems. Empirical research in Chapter 3 builds on knowledge of that same time-period Android.*

2.1 Motivation

Over the last couple of years, the *appification* of software has drastically changed the way software is produced and consumed and how users interact with computer devices. With the rise of web and mobile applications, the number of apps with a highly specialized, tiny feature set drastically increased. In appified ecosystems, there is an app for almost everything, and the market entrance barrier is low, attracting many (sometimes unprofessional) developers. Apps are encouraged to share features through inter-component communication, while risks are communicated to users via permission dialogs. Based on the large body of research available for Android as the pioneer of open source appified ecosystems, a systematization of Android security and privacy research can help to understand the progress that has been made, as well as remaining research gaps in appified ecosystems. Therefore we focus on the dominant appified ecosystem with a large real-world deployment: Android.

Necessity of a Systematization of Android/Appification Security. The large body of literature uncovered a myriad of appification-specific security and privacy challenges as well as countermeasures to face these new threats. As with all new research fields, there is no unified approach to research. As a consequence, efforts over the last half decade necessarily pioneered ways to examine and harden these systems. A problem with this approach is that there are lots of fragmented efforts to improve security and privacy in an appified platform, but no unified framework or understanding of the ecosystem as a whole. Therefore, we believe that it is time to systematize the research work on security and privacy in appified platforms, to offer a basis for more systematic future research.

Challenges and Methodology of the Systematization. While the fragmentation of the Android security research is our main motivation, it is at the same time our biggest challenge. Contributions to this research field have been made in many different areas, such as static code analysis, access control frameworks and policies, and usable security for end users as well as app and platform developers. To objectively evaluate and compare the different approaches, our first step will be to create a common understanding of the different security and privacy challenges and a universal attacker model to express these threats. Security solutions are by default designed with a very specific attacker model in mind. We found that in most Android research, this attacker model has been only implicitly expressed. However, to understand the role of a (new) approach within the context of Android's appified ecosystem, it is also important to understand which attacker capabilities it does *not* cover and how different approaches can complement one another. By studying the evaluation details of many representative approaches from the literature, we create a unified understanding of attacker capabilities. This forms the basis for analyzing the security benefits of different solutions and lays the groundwork for comparing approaches with respect to their role in the overall ecosystem.

One insight from our analysis of the challenges in Android's appified ecosystem, is that some security issues are new and unique to Android, as caused by the appification paradigm or the result of design decisions of its architects. Other well-known problems are aggravated by appification, while many security issues are lessened or solved by the appification paradigm. Such understanding is key to transcending Android to develop a broader picture of the future of software systems and the environments they will be placed in.

In particular, the tight integration of many non-traditional actors in the appified ecosystem creates interesting problems as well as opportunities. Platform developers, device vendors, app markets, library providers, app developers, app publishers, toolchain providers and end users all have different capabilities and incentives to contribute (in)securely to the ecosystem. Our systematization makes the important contribution of showing how previous research has interacted with these actors, identifying contributing factors to our research community's work creating a real-world impact.

Based on our systematization of this knowledge, we draw lessons learned from our community's security research that provide important insights into the design and implementation of current and future appified software platforms. We also create an overview of which areas have received focused attention and point out areas where research went astray. Finally, we address underrepresented areas that could benefit from or require further analysis and effort.

Please note that we are not discussing plausible problems and benefits of research solutions for adoption by Google or other vendors. Such factors can be manifold, such as technical reasons (e.g., backwards compatibility), business decisions (e.g., interference with advertisement networks), protection of app developers (e.g., intrusion of application sandboxes), or usability aspects. However, without concrete first-hand knowledge, any such discussion would merely result in speculation, which we do not consider a tangible contribution of a systematization of knowledge.

Systematization Methodology

There is a huge body of research work on Android security with (conservatively) over 100 papers published. Since we aim to systematize this research as opposed to offering a complete survey [240], we extracted key aspects and key papers to create

a foundation for our systematization. The focus of our systematization is on security issues and challenges in the context of appification and the app market ecosystem. We include both offensive works (i. e. papers that uncovered new security issues or classes of attacks) as well as defensive ones (i. e. papers that focus on countermeasures or new security frameworks). However, we do not focus on malware on appified platforms, as this has been dealt with in prior work [294]. We also exclude hardware-specific or other low level problems on mobile platforms, such as CPU side-channels, differential power analysis, or base-band attacks, which are independent from appification.
We selected the research based on the following criteria:

- *Unique/Pioneering*—Security issues which are unique to the Android ecosystem, i.e. have never been seen before.
- *Aggravated*—Security issues which have greater impact on an appified ecosystem than on traditional computing.
- *Attention*—Research on aspects that received more attention (i. e. many papers dealt with this specific aspect or the papers received high citation counts).
- *Impact*—Security research that affected a large number of users (or devices).
- *Scope*—Security issues which involve a large fraction of the appified world's actors. We include these issues since they are particularly hard to fix.
- *Open Challenges*—Research worked on issues or countermeasures that remain "unfinished" and highlight interesting and important areas of future work.

In the following, we systematize the research using the above rubric, extract a unified attacker model and evaluate the work both in terms of content and also on its placement within the Android ecosystem. We identify actors that are responsible for the problems, would benefit from solutions, and/or have the capability to implement and deploy them.

2.2 Problem and Research Areas

To identify important problem and research areas, we compare aspects of traditional software ecosystems with appified platforms, mainly focusing on Android.

Conventional Software Ecosystem vs. Appified Platform (Android)

We start our systematization by categorizing and summarizing key security challenges and issues that have been identified in the literature in both conventional software ecosystems and the appified world. Our intention for systematizing the key security challenges is to provide a systematic approach to help security researchers understand the (old and new) challenges that have been identified and to lay the foundation for a discourse on addressing these challenges.

Defining the Access to Resources

Controlling access to resources on a computer system requires 1) accurate definition of the security principals and protected resources in the system; 2) a non-bypassable and tamper-proof validation mechanism for any access (*reference monitor*); and 3) a

sound security policy that governs, for all requested accesses in any system state, whether access is allowed or should be denied. Android deviates from conventional OSes in all three aspects:

System Security Principals Conventional systems are primarily designed as multi-user systems with human users that have processes executing on their behalf. A small number of dedicated user IDs is assigned to system daemons and services that do not execute on behalf of a human user.

Appified security models build on the classic multi-user system: not only is the human user of the system considered a principal, but in fact all app developers that have their app(s) installed on the system are considered as security principals. Developers are represented by their app, which receives a distinct user ID (UID), exactly like the pre-installed system apps receive a UID. In recent Android versions with multi-(human)-user support, the traditional UID scheme is further extended: the UID is now a two-dimensional matrix that identifies the combination of the app UID (i.e., developer) and human user ID under which the app is currently running.

Implementation of the Reference Monitor Conventionally, reference monitoring is typically managed by the OS, e.g., the file system and network stack, so that user processes can build their access control on top.

Appified ecosystems also use the OS for low-level access control. However, the extensive application frameworks on top of which apps are deployed provide a different interface: following the paradigm of IPC-based privilege separation and compartmentalization in classical high assurance systems, security- and privacy-critical functionality is consolidated into dedicated user-space processes. Exposed IPC interfaces enforce access control on calling processes.

Security Policy In conventional software systems, multiple privilege level(s) for a process are defined: Processes can run as superuser (root), system services, with normal user privileges, guest privileges, and so on. All processes running under a certain privilege level share the same set of permissions and may access the same set of resources.

Modern appified ecosystems make a clearer distinction between system and third-party apps: Direct access to security- and privacy-sensitive resources (e.g., driver interfaces or databases) is only permitted to selected applications and daemons of the application framework. This policy is implemented, as in the conventional platforms, in the OS access control policies (i.e., discretionary and mandatory access control). However, system apps may request access to permissions that are not available to third-party apps. Third-party apps have by default no permissions set, but may request their permissions from a set commonly available to all third-party apps.

Sharing Functionalities

In conventional operating systems, third-party apps are usually self-contained and heavily used to incorporate external functionalities as libraries (e.g. the OpenSSL library to make TLS available in a program).

In addition to third-party libraries, in the appified world, apps also share functionality through inter-component communication (ICC), i.e., by providing a Service that can be accessed through Intents or persistent IPC connections. ICC is heavily

used to access system apps such as the map, phone, or Play app, but also popular third-party services, e.g., as offered by the Facebook and Twitter apps.

Software Distribution

Conventionally, software is distributed in a decentralized way: It can be downloaded from websites, purchased in online stores or shipped on physical media such as USB sticks or CDs. Software comes either in compiled binaries or, in case of open source software, as source code that needs to be compiled before installation.

Appified ecosystems often make use of centralized stores that distribute software/apps. These app stores allow developers to upload and distribute their software in a highly organized way. The app markets provide search, feedback and review interfaces for users and allow for centralized security mechanisms that can be enforced by the markets directly. We distinguish between commercial app markets such as Google Play and central software repositories that are widely used in different Linux distributions. In addition to simply distributing software by streamlining the process of searching and installing apps, commercial app markets have additional responsibilities such as billing, DRM (e.g., forward locking on Android) and in-app purchasing.

Software Engineering

Development Process Previously, single developers/companies developed software and in many cases distributed it themselves. They followed agreed-on rules (e.g., IDE, libraries, or frameworks to use) and could outsource in a regulated way to contracted (sub-)companies. In appified ecosystems, a chain of actors is responsible for the distribution of software, which is much more loosely coupled than the more stringent traditional development chains: The original developer, (often) a publisher, and increasingly development frameworks are involved.

Programming Environment In conventional operating systems, developers can choose what programming language they want to use (within the design space that the project leaves them), and a wide range of programming languages and frameworks are usually available to implement software. Appified ecosystems dictate programming languages and frameworks to enforce compatibility with their application framework and hence robustness of the deployed applications. Android developers, for instance, are required to use Java and the Android SDK/NDK. App creators play a crucial role in modern appified ecosystems: They provide easy-to-use clickable interfaces to produce software that can be run on multiple platforms.

Present Classes of Programming Errors

Programming errors, such as logic errors and run-time errors, are the dominant sources of software vulnerabilities in conventional software ecosystems. While recent years have demonstrated that they are also present in mobile platforms with the same devastating effects, the API-dependent design of apps has introduced a new range of problems into the appified world as a direct consequence of misuse of programming APIs of the surrounding application framework. This differs from the traditional ecosystem, where this class of errors is limited mostly to library APIs, since the application framework API is a necessity to make apps operational.

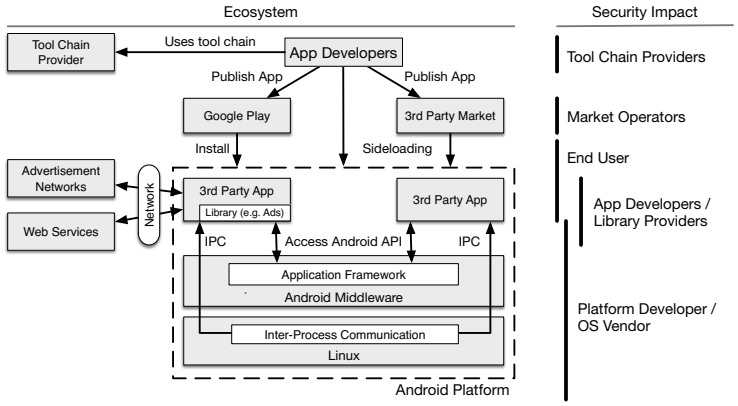

FIGURE 2.1: The Android ecosystem: Actors and their impact on the ecosystem's security.

Webification

In conventional software ecosystems, software is mainly self-contained and its primary functionality does not depend on the availability of remote resources such as web services. The appification paradigm has seen a shift towards increasingly web service-oriented architectures that depend on server backends to provide their promised functionality. At the end of the spectrum are apps that consist merely of a webview component that appears to be local app logic, but in fact is not much more than a restricted web browser for the service's backend web servers.

Software Update Mechanisms

Conventional OS updates are centrally organized, while the updating process for third-party software takes, in contrast, a greater effort: Every program needs to be updated (and hence, often started and restarted) separately. Only systems with a central software distribution channel improve on this situation (e.g., Linux distributions). The situation for updates in appified ecosystems is currently the exact opposite. Fragmentation is a huge issue in appified ecosystems, such as Android, and impedes the OS update process. As many different network providers and device vendors customize parts of the operating system, they need to manage OS updates on their own, resulting in lengthy and complicated update procedures. As a result, many Android devices do not receive OS updates at all. In contrast, app updates are straightforward and fast, as centralized app stores push updates immediately to their users.

2.3 Android/Appified Ecosystem

As an example for appification, we provide an overview of the Android ecosystem, the actors involved and their impact on the ecosystem's overall security. We use Figure 2.1 as our reference to introduce the actors and their interaction patterns.

2.3.1 Ecosystem Overview

At the core of appification ecosystems are the *app developers*, producing the millions of apps available for the *end users*. The number of Android app developers is vastly

larger than for the traditional desktop software ecosystem. For instance, in the current Play market[1] roughly 460,000 distinct developer accounts have published applications, where an account can also belong to an entire company or team of developers. These app developers rely on the rich APIs of the platform SDK, which is provided by the *platform developers*. These APIs provide access to core functionalities (e.g., telephony, connectivity and sensors like accelerometers) as well as to user data (e.g., contact management, messaging, picture gallery).

Developers can request access to those functionalities by requesting permissions in their app's manifest file (e.g. the `CONTACTS` permissions grants access to the user's address book). End users are presented permission dialogs at install time. Those dialogs present all the permissions previously requested by a developer and inform users about an app's resource access. Since version 6 (*Marshmallow*), Android also introduced, like iOS has done several iterations before, the concept of dynamic permissions: a small subset of all permissions are granted by the user at runtime when an app requests access to protected interfaces instead of statically at install time, and those selected permissions can also be revoked again by the user. It is also possible for developers to define custom permissions that can grant access to their app's functionality to other apps written by the same developer, system apps, or all apps installed on the device.

Android apps are composed of Java code (compiled to bytecode format for the CPUs of mobile platforms) and of native code in the form of C/C++ shared libraries. *Library providers* such as *advertisement networks* support developers in creating ad-supported apps by offering dedicated *ad libraries* that apps can rely on, thus firmly integrating the ad library in the final application package. Many apps connect to *web-services* (e.g., cloud-based services or other backends) and use web-technologies such as HTML, CSS and Javascript. This move to web apps is typical for the appification paradigm.

Typically for the shift to appification is the way monetization works: App developers can sell their apps to end users for fixed one-time prices (using central app stores such as Google Play), they can collaborate with *advertisement networks* by displaying advertisements in their apps and receiving shares of the advertisement revenues, or they can offer in-app purchases, e.g., users can buy additional features of the app. Those options are not mutually exclusive, but conventionally paid apps refrain from displaying ads. Together they lower the economic burden on developers and streamline the process of purchasing and installing apps for end users [158].

Unlike other current appified ecosystems, Android allows (and actually encourages) inter-component communication (ICC), which prompts developers to divide their apps into smaller parts (e.g., plugins) and allows them to act as service providers (e.g., Facebook app, Play app, etc.). Technically, ICC is based on the Linux kernel's inter-process communication—primarily via a new IPC mechanism called *Binder*. However, since logical communication occurs between application components such as databases, user interfaces, and services, this Android-specific IPC has been coined as *Inter-Component Communication* in the literature [76].

2.3.2 Involved Actors

Software ecosystems involve a number of actors that each have their own rights and duties, which differ between appified and conventional ecosystems in some aspects.

[1] Approximately 1.5 million free apps crawled in February 2016.

TABLE 2.1: All actors in the ecosystem and the impact of their security decisions on the remaining actors.

Actor	OS Developer	Hardware Vendor	Library Provider	Software Developer	Toolchain Provider	Software Publisher	Software Market	End User
OS Developer		◐	●	●	◐	●	●	●
Hardware Vendor	○		●	●	○	○	○	●
Library Provider	○	○		●	○	○	○	●
Software Developer	○	○	◐		○	○	○	●
Toolchain Provider	○	○	○	●		○	○	◐
Software Publisher	○	○	◐	◐	○		◐	●
Software Market	○	○	◐	◐	○	○		●
End User	○	○	○	○	○	○	○	

● = fully applies; ◐ = partly applies, ○ = does not apply at all.

We differentiate these actors as groups of ecosystem participants, describe their primary task(s), their power to influence the security and privacy of the ecosystem with their decisions, and then give concrete examples of each class of actors. Table 2.1 illustrates the different actors, their influence on the ecosystem's security and privacy, and their interaction with each other.

Although feedback loops can be established between any number of actors, in the following discussion we focus on the potential *direct* impact of a security decision made by one user on all other actors. We do not consider *indirect* impact, e.g., when users protest against or boycott certain apps and thus force app or platform developers to react.

Platform Developers

Platform developers are responsible for providing the Android Open Source Platform (AOSP). They make basic system and security decisions and *all other actors* build on their secure paradigms. Library providers and app developers are bound to the provided SDK, and app markets have to rely on Android's open approach (instead of, for example, Apple's walled-garden ecosystem). An exception is that device vendors can implement their own security decisions and need not adhere to Android's paradigms. In reality, though, they mostly build upon the provided foundations.

Device Vendors

Device vendors adopt the AOSP and customize it for their different needs. A variety of device vendors currently share the market for mobile devices using Android [180]. Besides adaptation of the basic Android software stack to the vendor-specific hardware platforms, vendors customize in order to distinguish their Android device from their competitors'. Thus, many versions of vendor-specific apps and modified versions of Android's original user interface are being distributed with Android-based platforms. The impact of device vendors on the ecosystem's security is significant: Although, naturally, their customizations only affect their customers, this user-base can be large in case of big vendors such as Samsung or HTC. Device vendors can adopt security decisions from the platform developers or add their own solutions (cf. Samsung KNOX [217]) on which library and app developers can build. However, device vendors cannot change the way apps are published in markets, which is why their impact on publishers and markets is very limited—e.g. they could not enforce CA-signed instead of self-signed certificates for app signing practices without breaking Android's guidelines.

Library Providers

Based on the platform's API, library providers build their own APIs to offer new features such as *ad services* or to make the use of (possibly unnecessarily) complicated platform APIs easier for app developers. Libraries exist for UI components (they can but need not be attached to network tasks) as well as for ads or crash reports. Library developers have the power to make all apps that include them either more or less secure. Library developers suffer or benefit from security decisions made by platform developers and device vendors. However, their decisions do not affect the platform security in general. Their positive/negative security decisions propagate to app developers who choose to use their libraries—they can, for example, wrap badly designed programming interfaces from platform developers. Their decisions affect neither app publishers nor markets directly. Typically, library providers offer *ad services*, *networking features* or *app usage evaluation features*.

App Developers

App developers write apps using the APIs defined by platform developers and of those libraries they choose to include. They can opt to write code themselves or use existing third-party code. In theory, they can make essential contributions to security. In practice, they make unsafe choices and implement features in the least laborious way, which is frequently not the most secure choice.

While app developers can break secure default interfaces provided by platform developers/device vendors (e.g. crypto primitive API misuse), this has no effect on the platform security in general. Their decisions neither affect app publishers nor markets directly. Still, app developers may impact libraries' security (e.g. as fraud is a frequently evaluated issue).

Toolchain Providers

Toolchain providers offer helpful tools for app developers (e.g. the Eclipse ADT for Android app development). They can implement many analysis tools that help discover API misuse. Toolchain providers can fix some weaknesses introduced by platform providers and device vendors (e.g. confusing permission descriptions, or

hard-to-use APIs). All app developers and their users benefit/suffer from good/bad toolchain provider support.

App Publishers

App publishers are professional service providers that help developers publish their apps to certain markets. They receive either binary or source code, add certain properties like ads, and distribute the app to one or more app markets. In theory, they can run preliminary analyses on the code and report or fix bugs, as well as filter malware. If app signing is delegated to the app publishers, they could also surreptitiously insert malicious code. Several app publishers maintain substantial numbers of apps [80] and thereby may substantially impact markets' security. Hence, a single publisher's impact on the ecosystem's security is rather impressive.

App Markets

App markets—Google Play is the most popular one—distribute apps from developers to the end users. Users as well as app developers rely on them to make sure that the apps are distributed in a consistent, unchanged, reliable, and benign way. In theory, app markets have the potential to find not only malware, but also buggy and unsafe code. To do this, they can apply various kinds of security analyses techniques—such as static or dynamic code analysis—on all apps they distribute. For example, Google Play runs supposedly multiple tests on apps prior to distribution, including static/dynamic analysis and machine learning [105]. However, they do not run deeper checks to detect dangerous misuse of the Android API. No app market runs (theoretically possible) runtime tests, nor do they exclude apps signed with the same key corresponding to different developers.

Users

Users are app consumers in the ecosystem. They can make the decision to install (non-pre-installed) apps, and have to confirm the permissions that apps request. They are the most likely target of attacks. In theory, they can make safe choices, as well as choose not to use important credentials. However, a single user's impact on the ecosystem's security is negligible. Users as a group have to rely on security decisions made by all other actors in the ecosystem.

2.3.3 Global Attacker Model

We provide a taxonomy for *attacker capabilities* on Android. This taxonomy reflects the *threat models* we extracted during our systematization in Section 2.4 and helps to later on compare proposed countermeasures.

When considering the attacker capabilities, we had the options to order them *across capability categories* or *within categories*. We decided to order them *within categories*, since our categories depend on too many distinct factors to be comparable and since we base our systematization on those categories. For instance, a user connecting frequently to public Wi-Fi access points is susceptible to network attacks, but this behavior does not influence other capability categories like, e.g., piggybacking apps. We order the attacker capabilities vertically, i.e., we rate the power of attackers in specific capability categories. We use the following semantics to note attacker capabilities in each category: Solid circles (●) denote strong capabilities corresponding

to a weak attacker model. Half-filled circles (◐) denote common attacker capabilities, while hollow circles (○) describe the absence of any capability in the category, strengthening the attacker model.

Next, we introduce our categories for attacker capabilities, informally define the exact capabilities attackers may have in each category, and explain our ordering of those capabilities.

C1—*Dangerous permissions:* The attacker has code running on the victim device, which has been granted *dangerous* permissions (●) that give access to privacy sensitive user data or control over the device that can negatively impact the user. Dangerous permissions must be explicitly granted by the user during app installation. We assume *normal* permissions (◐) when the attacker has been granted only permissions that are of lower risk and automatically granted by the system.

C2—*Multiple apps:* Attacker-controlled apps are running on the user device. Full capability indicates that the attacker has two or more apps running on the victim device (●). This would enable collusion attacks via overt and covert channels. Half-capability (◐) means that only one attacker-controlled app is running on the device. In general, the capability of having at least one app on the user device enables the attacker to engage in ICC with other apps on the device or to scan the local file system to the extent the attacker-controlled apps' permissions allow this.

C3—*Piggybacking apps:* The attacker re-packages other apps and is able to modify the existing code or include new code (●). A limited piggybacking capability (◐) is assumed if the attacker provides code that is intentionally loaded by app developers into their apps (e.g., libraries). Limited piggybacking is assumed to be the weaker capability, because libraries used by developers are hosted by the app (i.e., share the host sandbox) limiting the attacker to the host app's permissions. In contrast, repackaging apps allows the attacker to request more permissions for the repackaged app.

C4—*Native code:* The attacker has an app containing native code, i.e., shared libraries. This requires having at least one app on the device under control (**C2**.◐). Native code that implements exploit payload, native programs, or zipper/crypto routines for obfuscation are considered as full capability (●). Non-exploit code that still provides the means to modify the app's memory space is assumed as half-capability (◐). Although Android's design permits all apps to contain native code, there are apps that contain none (○).

C5—*Dynamic code loading:* The attacker is able to dynamically load code at runtime (●) into an app (e.g., using the Java reflection API). This requires having at least one app on the device under control (**C2**.◐). Half-capability (◐) is assumed if the attacker can inject code into another, benign but insecure app. Dynamic code loading is assumed to be a stronger capability than code injection, since dynamic loading allows the attacker to use obfuscation techniques to execute the attack surreptitiously.

C6—*Network attacks:* The attacker is capable of modifying/interrupting/forging the Wi-Fi and cellular network communication of the end user device (●). We assume a passive attacker (◐) if the attacker is only able to eavesdrop on the communication. Technically, a network attack can be accomplished as in traditional attacker models by, e.g., setting up a rogue access point or base station. On Android, an attacker can gain the same capability through a malicious VPN app, through which all network

traffic of all processes is routed when it is activated by the user. This requires at least **C2.o**.

2.4 Systematization of Research Areas in Appified Ecosystems

Building on the differences between conventional and appified ecosystems as well as the actor and global threat model of the Android ecosystem, we now identify fields of research that we think need to be systematized, considering a number of representative research papers for each field. We discuss challenges in the respective fields, regarding the global actor model, identifying the involved actors, their respective roles in causing a specific problem and their potential in resolving it. Referring to the global threat model, we summarize the attacker capabilities assumed in the threat models required to exploit the problem areas. Moreover, we present selected, representative Android security countermeasures if available. We do not claim that our systematization is all-encompassing, nor that it includes all problem fields ever identified for Android nor all countermeasures to known problems; however, we took great care to choose a representative selection (see Section2.1).

2.4.1 Permission Evolution

The concept of permission-based access control for privileged resources is one of the cornerstones of Android's security design and has received a lot of attention by the security research community.

Challenges

We sub-categorize the identified problems and challenges according to the most affected actors in the ecosystem: the end users and app developers.

Permission Comprehension and Attention by End Users To effectively inform end users about the privacy risks that an app imposes, it is imperative that end users are capable of correctly perceiving the risk of granting the access rights requested by apps. Pioneering work showed that only a very small fraction of users could correctly associate privacy risks with the respective permissions [91]. One potential root cause for this lack of understanding seems to be that permissions communicate resource access, but do not explain how accessed data is processed and distributed [89]. Hence, users tend to underestimate the risks ("the app will not misuse its permissions") or overestimate the risks ("the app will steal all my private information") [91]. A lack of user comprehension of permissions allows attackers to create malicious apps that request all necessary sensitive permissions for their operations (as demonstrated, e.g., by the Geinimi Trojan [279]).

Apps published after Android v6.0 may request a small subset of privacy-related permissions during runtime instead of at installation. Requesting permissions dynamically when they are required by the app should provide users with more contextual information and help them in their decision making process. However, Wijesekera et al. [274] have shown that this desired contextual integrity—i.e., personal information is only used in ways determined appropriate by the users—is not necessarily provided by dynamic permissions and runtime consent dialogs: A majority

TABLE 2.2: Security challenges in the appified ecosystem, actors capable of redressing the problems, and attacker capabilities considered in threat models of different problem areas.

				Causative Actors									Fixable by								Attacker capabilities					
Problem area	**Focus**	**References**	Discussed in Section	A1. Platform Developers	A2. Device Vendors	A3. App Markets	A4. Library Providers	A5. App Developers	A6. App Publishers	A7. Toolchain Providers	A8. End Users	R1. Platform Developers	R2. Device Vendors	R3. App Markets	R4. Library Providers	R5. App Developers	R6. App Publishers	R7. Toolchain Providers	R8. End Users	C1. Sensitive permissions	C2. Multiple apps	C3. Piggybacking apps	C4. Native code	C5. Dynamic code loading	C6. Network attacks
Permission-based Access Control and Least Privilege	Permission Attention and Comprehension by End Users	[26, 89, 91]	2.4.1	●	○	●	◐	◐	○	○	◐	●	○	●	◐	◐	○	◐	○	●	◐	○	○	○	○
	Permission Comprehension by App Developers	[19, 28, 75, 87, 268, 277]	2.4.1	●	○	○	◐	◐	○	○	○	●	○	○	○	○	○	◐	○	○	◐	○	○	○	●
	Permission Attention by App Developers	[53, 65, 75, 93, 96, 112, 151, 277, 293]	2.4.1	◐	○	○	○	●	○	○	○	◐	◐	○	○	●	○	○	○	○	◐	○	○	○	○
	Missing Privilege Separation	[63, 74, 114, 183, 188, 237, 258]	2.4.2	●	○	○	●	◐	○	○	○	●	○	○	◐	○	○	○	○	●	◐	◐	○	●	○
	Missing Efficacy of Security Apps	[204, 294, 295]	2.4.2	●	○	○	○	○	○	○	○	●	○	○	○	○	○	○	◐	●	◐	○	●	◐	○
Webification Issues	—	[55, 128, 152, 155, 163, 262]	2.4.3	●	○	○	●	○	○	●	○	●	○	○	●	○	○	●	○	○	◐	○	○	○	●
API Misuse of App Development Frameworks	—	[50, 70, 82, 84, 85, 188, 233]	2.4.4	◐	○	○	●	●	○	○	○	◐	○	◐	●	●	○	◐	○	○	●	○	○	●	●
Software Distribution Channels	App Piracy and Malware Incentives	[62, 90, 101, 290, 291, 295]	2.4.5	○	○	◐	○	●	○	○	○	○	○	◐	○	○	○	○	◐	○	○	●	○	○	○
	Application Signing Issues	[26, 80]	2.4.5	●	○	◐	○	●	●	○	○	●	○	◐	○	●	●	○	○	○	○	●	○	○	○
Vendor Customizations and Fragmentation of the Ecosystem	—	[1, 112, 160, 277, 283]	2.4.6	●	●	○	○	○	○	○	○	●	●	○	○	○	○	○	○	●	◐	○	○	○	○

● = fully applies; ◐ = partly applies, ○ = does not apply at all.

TABLE 2.3: Categorization of proposed Android security countermeasures, their potential implementers, and their addressed attacker model.

				Possible implementers								Considered attacker model					
Problem area	**Focus**	**Solution**	**Reference**	R1. Platform Developer	R2. Device Vendor	R3. App Market	R4. Library Provider	R5. App Developer	R6. App Publisher	R7. Toolchain Provider	R8. User	C1. Sensitive permissions	C2. Multiple apps	C3. Piggybacking apps	C4. Native code	C5. Dynamic code loading	C6. Network attacks
Permission evolution (Section 2.4.1)	System Security Extension	Kirin	[75]	●	●	○	○	○	○	○	◐	●	○	○	○	○	○
		TaintDroid	[73]	●	●	○	○	○	○	○	○	●	◐	○	○	○	○
		Apex	[170]	●	●	○	○	○	○	○	◐	●	◐	○	○	○	○
		Sorbet	[96]	●	●	○	○	◐	○	◐	○	●	◐	○	○	○	○
		QUIRE	[67]	●	●	○	○	◐	○	○	○	○	◐	○	◐	○	○
		IPC Inspection	[93]	●	●	○	○	○	○	○	○	○	◐	○	◐	○	○
		XManDroid	[39]	●	●	○	○	○	○	○	○	●	●	○	◐	○	○
	SDK / Tool-chain Extension	Stowaway	[87]	○	○	○	○	◐	●	●	○	●	○	○	○	○	○
		PScout	[19]	○	○	○	○	◐	●	●	○	●	○	○	○	○	○
		Curbing Permissions	[256]	○	○	○	○	◐	●	●	○	●	○	○	○	○	○
	HCI Modifications	Decision making process	[135]	○	○	●	○	○	○	○	●	●	○	○	○	○	○
		Using personal information	[118]	○	○	●	○	○	○	○	●	●	○	○	○	○	○
	(Meta) Data Analysis	WHYPER	[181]	○	○	●	○	◐	◐	○	◐	●	○	○	○	○	○
		AutoCog	[202]	○	○	●	○	◐	◐	○	◐	●	○	○	○	○	○
		DescribeMe	[287]	○	○	◐	◐	●	●	◐	○	●	○	◐	○	○	○
	User study	Permissions remystified	[274]	●	●	○	○	●	○	○	●	●	◐	○	○	○	○
Permission revolution (Section 2.4.2)	System Security Extension	Saint	[177]	●	●	○	○	◐	○	○	○	●	◐	○	◐	○	○
		CRePE	[59]	●	●	○	○	○	○	○	◐	●	◐	○	◐	○	○
		TISSA	[296]	●	●	○	○	○	○	○	○	●	◐	○	◐	○	○
		SE Android	[227]	●	●	○	○	○	○	○	○	●	◐	○	●	○	○
		TrustDroid	[40]	●	●	○	○	◐	○	○	○	●	◐	○	●	○	○
		FlaskDroid	[41]	●	●	○	○	○	○	○	○	●	◐	○	●	○	○
		ASM	[122]	●	●	○	○	(◐)‡	○	○	(◐)‡	(●)‡	(◐)‡	○	(●)‡	○	○
		Compac	[266]	●	●	○	○	○	○	○	○	●	◐	●	○	○	○
		AdDroid	[183]	●	●	○	○	◐	◐	○	○	●	◐	◐	○	○	○
		AdSplit	[223]	●	●	○	○	◐	○	○	○	●	◐	◐	○	○	○
		LayerCake	[210]	●	●	○	○	○	○	○	○	○	◐	◐	○	○	○
	Binary modifications	Aurasium	[284]	○	○	◐	○	○	○	◐	●	●	◐	○	◐	●	○
		Dr. Android,Mr. Hide	[127]	○	○	●	○	●	◐	◐	●	●	◐	○	◐	●	○
		I-ARM Droid	[66]	○	○	◐	○	○	◐	◐	●	●	○	○	○	●	○
		AppGuard	[21]	○	○	◐	○	○	◐	◐	●	●	○	○	○	●	○
		Boxify	[20]	○	○	○	○	○	○	○	●	●	◐	○	◐	●	○
Webification (Section 2.4.3)	System Security Extensions	Morbs	[262]	●	●	○	○	◐	◐	◐	○	○	◐	○	○	○	●
	SDK / Tool-chain Modification	NoFrak	[155]	○	○	○	●	◐	●	○	○	○	○	○	○	○	●
		NoInjection	[128]	○	○	○	●	◐	●	○	○	○	○	○	○	○	●
Programming-induced leakage (Section 2.4.4)	SDK / Tool-chain Extension	MalloDroid	[82]	○	○	●	○	◐	●	●	○	○	○	○	○	○	●
		CryptoLint	[70]	○	○	●	○	◐	●	●	○	○	◐	○	○	○	◐
		SSL API Redesign	[85]	●	●	○	○	○	○	◐	○	○	○	○	○	○	●
	App Analysis	SMV-Hunter	[233]	○	○	●	○	◐	●	●	○	○	○	○	○	○	●
		CHEX	[151]	○	○	◐	○	●	●	○	○	●	○	○	○	○	○
		SCanDroid	[49]	○	○	●	○	○	◐	○	◐	●	○	◐	○	○	○
		AndroidLeaks	[100]	○	○	●	○	○	◐	○	○	●	○	◐	○	○	○
		FlowDroid	[16]	○	○	●	○	○	◐	○	○	●	○	◐	○	○	○
Software Distribution (Section 2.4.5)	Market solution	Meteor	[27]	◐	◐	○	●	○	◐	○	◐	●	○	●	○	○	○
		MAST	[47]	○	○	●	○	○	○	○	○	●	○	●	●	●	○
		Application Transparency	[80]	○	○	●	●	○	○	○	○	○	○	●	○	○	○
	(Meta) Data Analysis	DroidRanger	[295]	○	○	●	○	○	●	○	○	○	○	●	●	●	○
		DNADroid	[62]	○	○	●	○	○	●	○	○	○	○	●	○	○	○
		RiskRanker	[113]	○	○	●	○	○	●	○	○	●	○	●	●	●	○
		CHABADA	[108]	○	○	●	○	○	○	○	○	●	○	●	○	○	○
		Collaborative Verification	[78]	○	○	●	○	●	○	○	○	●	○	○	○	○	○
		MassVet	[51]	○	○	●	○	○	○	○	○	●	○	●	○	○	○
	SDK / Tool-chain Extension	AppInk	[289]	○	○	○	○	◐	●	●	○	○	○	●	○	○	○
Software Update Mechanism (Section 2.4.7)	(Meta) Data Analysis	SecUp	[283]	●	●	○	●	○	○	○	●	(●)†	◐	○	○	○	○

● = actor must implement solution/attacker capability fully addressed; ◐ = actor should/can participate in solution/attacker capability partially addressed
○ = actor not involved/attacker capability not addressed.
† Requests sensitive permissions and attributes defined by a future Android OS version; ‡ Depends on loaded security module

of privacy-related permission requests occur when the user is not interacting with the requesting application or even with the phone, and, moreover, requests occur at a frequency that prohibits involving the user in every decision making process. As a consequence, users failed to establish the connection between the permission request and the apps' functionality and consent dialogs are only shown during first request to grant access until manually revoked by the users although subsequent permission checks might occur in a different privacy-context than the initial request.

Permission Comprehension and Attention by App Developers Android's security design requires app developers to contribute to platform security by requesting, defining, and properly enforcing permissions in order to retrieve and protect sensitive user data. Thus, even more than for end users, it is imperative that app developers understand permissions and the security tools at their disposal.

Permission Comprehension by App Developers. A number of studies [19, 28, 75, 87, 268] give insight into how app developers comprehend permissions and, in particular, how the SDK supports them in their task to realize least-privileged apps (e.g., considering the stability of the permission set or the extent to which permission-protected APIs are well-documented). Between 30% [87] and 44.8% [268] of the studied apps requested unnecessary permissions, i.e., were over-privileged and in clear violation of the least-privilege principle. Moreover, several apps have been found that request non-existent or even wrong permissions. Even developers of system apps, who have access to the highest privileged and highly dangerous API functions, did not exhibit a significantly better understanding of permissions [277].

To understand the root causes behind the developers' incomprehension of permissions, the studies analyzed the Android API documentation, finding that the API is insufficiently documented and does not identify all permission-protected APIs. Even worse, the documentation also contained errors, e.g., describing the wrong permission required for an API function. Confusing permission names also contribute to these misconceptions. These inconsistencies and the instability of the API impede a clear and well-developed documentation and thereby contribute to the developers' incomprehension of permissions and to confusion about permission usage.

Permission Attention by App Developers. Besides developers' (lack of) comprehension of permissions, the thoughtfulness of developers when enforcing permissions was studied, as well as their level of comprehension of the mechanisms at their disposal to accomplish this task. Although Android's security design incorporates important lessons learned from prior operating system security research [216], the fact that it allows and even encourages differently privileged apps to communicate with each other has piqued the security research community's interest in how this can be exploited by unprivileged apps to escalate their privileges [53, 65, 75, 93, 96, 112, 151, 277, 293]. In particular, various works have identified an increase in failure of app developers to properly protect their app's IPC-exposed (or `exported`) interfaces and to transitively enforce permissions [93]. This opens the attack surface for confused deputy attacks[2] to, e.g., initiate phone calls [75], hijack ICC [53], or exfiltrate sensitive user data [151, 293]. The root cause of many of those identified vulnerabilities is that application components were by default exported to be IPC-callable and thus require that the developers either explicitly protect them with permissions or hide the components. As indicated by the uncovered vulnerabilities, most developers are

[2]The literature has yet to agree on a fixed term. Other works designate this attack category as permission re-delegation [93], as component hijacking [151], or as capability leaks [48, 112]. We use the term confused deputy [119].

unaware of these conditions. To phrase this in the terms of Saltzer's and Schroeder's secure design principles [216]: Android failed to implement fail-safe defaults.

Countermeasures

Recent changes [107] in the default installer app for Google Play aim to improve permission perception for users. Installers present permissions with low granularity in groups, while some commonly requested permissions, like `INTERNET`, are not presented at all anymore. This shift in permission presentation can be viewed as a pure user experience decision, not as an enabler of user comprehension.

Research has made several suggestions to enhance the usability of permissions for both end users and developers: Kelley et al. [135] propose to enrich permission dialogs with more detailed privacy-related information to help users make a more informed decision. Porter Felt et al. [88] propose making the permission-granting mechanism dependent on the kind of permission that is requested, e.g., auto-granting non-severe permissions with reversible side-effects, trusted UI for user-initiated or alterable requests, or confirmation dialogs for non-alterable, app-initiated requests that need immediate approval. A concrete realization of trusted UI are access control gadgets by Roesner et al. [211] that allow a user-driven delegation of permissions to apps whenever such widgets can be effectively integrated into the apps' workflows. Wijesekera et al. [274] suggest more intelligent systems that learn about their users' privacy preferences and only confront users with consent dialogs when a permission request is unexpected for the user. This consent dialog should provide sufficient contextual cues for users, e.g., clearly indicating the app requesting the access to protected resources as well as clearly communicating why the resource is accessed. Liu et al. [149] propose eliminating the burden of understanding the enormous list of permissions by using a limited set of privacy profiles including certain permissions instead; and Felt et al. [87] propose to improve API documentation to simplify permission requests for app developers.

Multiple system extensions have been suggested to enhance the permission system: The seminal *Kirin* [75] OS extension used combinations of permissions requested by an app to detect potential misuse of permissions and also revealed confused deputy apps on AOSP. *Apex* [170] introduced dynamic and conditional permission enforcement to Android. *TaintDroid* [73] used dynamic taint tracking to reveal for the first time how apps actually use permission-protected data and uncovered a number of questionable privacy practices that motivated enhancements to the permission system and access control on ICC. *Sorbet* [96] was first to model Android permissions and uncovered problems with desired security properties (like controlled delegation of privileges) on Android.

Some system extensions specifically aim at mitigating confused deputies: *XManDroid* [39] primarily augments the permission enforcement with policy-driven access control, where policies specify confused deputy and collusion attacks [65, 154] states. *QUIRE* [67] establishes provenance information along ICC call paths, enabling callees to evaluate their trust in the caller. *IPC inspection* [93] reduces the privileges of callees to the privileges of the caller.

WHYPER [181] and *AutoCog* [202] apply NLP techniques to automatically derive the required permissions from app descriptions, taking developers out of the loop, and check whether described functionality and actually requested permissions correspond. *DescribeMe* [287] takes the opposite track and generates security-centric app descriptions from analysis of app code in order to increase user understanding of the app.

Actors' Roles

Platform developers (**A1**.●) and market operators (**A3**.●) are fully responsible for the permission comprehension problems, as the platform enforces use of the current permission system, and the platforms' and the markets' installers communicate the privacy risks of installing applications to users. Library providers (**A4**.◐) contribute to this problem through their permission requests. App developers (**A5**.◐) tend to over-privilege their apps (either for their own needs, or on behalf of library providers their apps use), making apps appear unnecessarily dangerous. End users (**A8**.◐) tend to pay little attention to permissions [91], and only have the option of accepting everything or not installing the app at all.[3] Thus, while end users' behavior eventually opens the door to misuse by malware, end users have limited options and capabilities to detect whether permissions are being misused.

This problem could potentially be fixed by platform developers (**R1**.●) by changing their access control paradigm and avoiding conditions for some of the identified vulnerabilities (e.g. failing to implement fail-safe defaults). Additionally, by helping app developers (**R5**.◐) and library providers (**R4**.◐) in realizing security best practices for defensive programming through tool support [87, 256] (**R7**.◐), this indirectly helps end users. App markets (**R3**.●) could make their permission dialogs more comprehensive, demand justification from app developers and run static analyses on received app packages to adjust permissions accordingly.

Lesson Learned

In conventional ecosystems, neither developers nor users were involved in the process of requesting or granting fine-grained permissions to access resources on a computer. Allowing developers to request and define fine-grained permissions and presenting end users permission dialogs is a good idea in theory. However, research illustrates that this approach overburdens both: Developers tend not to focus their efforts on the selection process for permissions [87], while end users neither understand nor pay much attention to Android's permission dialogs [54, 91]. Research has strived to improve permission dialogs [135, 149, 274], but none of these approaches has solved the two-sided usability and comprehension problem. Permission dialogs have issues similar to warning messages: They fail to lead to the desired effect, as users tend to click through them, misunderstand their purpose, and hence do not benefit from them.

Instead of continuing the current line of research, we propose a clean break and a shift towards taking both users and developers out of the loop: Approaches that try to automatically derive the required permissions for an app based on its category, description, and similarity to other apps seem to take a more promising track [181, 202, 287]. Another promising alternative seems to be authorizing entire information flows instead of only access to resources. Although not new [164, 286], this idea seems worth being re-investigated for appified platforms that put the burden of granting permissions onto their end users.

[3] While this has changed with Android v6.0, developers nullify this change by making their apps compatible with older Android versions.

Our assessment (Permission Evolution): The decision to realize permissions as implemented by Android was understandable at Android's launch, but the concept has failed in practice, and was presumably doomed to fail from the beginning.

2.4.2 Permission Revolution

A dedicated line of research has investigated the possibilities of extending alternative access-control models to the Android platform to establish more flexible, fine-grained, and mandatory control over system resources and ICC. This research followed two major directions: OS extensions and Inlined Reference Monitoring (IRM).

Challenges

Missing Privilege Separation The most common third-party code distributed with apps is analytic and advertisement libraries that display ads in order to monetize the app [258]. More than 100 unique ad libraries are available for the different ad networks included in more than half of all apps [74, 114, 183, 237, 258].

The host app and third-party libraries engage in a symbiotic relationship that currently requires mutual trust. Libraries execute in the context of their host app's sandbox and inherit all privileges of their host app. However, ad libraries tend to exploit these privileges and exhibit a variety of dangerous behaviors, including misconduct such as insecure loading of code from web sources [188] as well as collecting users' private information [114]. Inversely, developers of host apps have a strong interest in monetizing their apps. Fraudulent app developers can exploit the symbiotic relationship [63] to surreptitiously steal money from the ad network by faking click events [63]. Android's design failed to provide privilege separation between these two principals [216], worsening the privacy threat of ad libraries to users' data in comparison to conventional browser-based ads [237].

Ineffective Security Apps Android follows the mantra that *"all applications are created equal"* [179]. However, this also implies that apps by external security vendors, such as anti-virus apps, do not have higher privileges than other apps. Studies have investigated to what extent this philosophy influences the efficacy of such security apps [204, 294, 295]. Prior systematization of existing Android malware has evaluated the effectiveness of existing anti-virus apps for Android and reported that detection rates vary from 54.7%-79.6% [204, 294, 295]. One study [204] suggests that platform support for anti-virus apps is essential to improve their efficacy.

Lack of Support for Mandatory Access Control Mobile devices are often used in fields with strong security requirements, such as enterprises and government sectors. Conventional operating systems in those contexts apply advanced access control models that protect more sensitive information (e.g., non-interference between two distinct security levels). The support for mandatory access control is a cornerstone of the platform security of such established systems. Conversely, Android lacks any support for mandatory access control.

While the requirement of supporting advanced access control schemes is intuitive and plausibe, we are not aware of any academic security requirements analysis that focuses on those particular stakeholders (i.e., enterprise and government sectors) on mobile devices and that could describe the particular challenges that come

with enabling support for such access control schemes on mobile devices. Only governmental guidelines have been published, e.g., by NIST [234]. Consequently, academic research has explored the particular challenges of adding mandatory and alternative access control models to Android from different angles, not all of which directly relate to high-security deployment.

Countermeasures

To provide advanced access control models and robust defenses against malware on Android, research has followed two main directions for adding access control to Android based on the responsible deployment actor.

Alternative Access Control Models Early work [59, 177, 296] explored how access control within Android's application framework can be more semantically rich and dynamic and introduced mechanisms that have since been adopted by several follow-up works. The seminal *Saint* [177] architecture allows app developers to define policy-based restrictions and conditions on ICC to and from their app. *CRePE* [59] extended Android with context-related access control for system resources, where context is defined as the device state and senseable environment. *TISSA* [296] introduced access control mechanisms for fine-grained data sharing, such as returning filtered, fake, or empty data from calls to framework APIs.

More recently, the *SE Android* [227] project solved the technically complex challenge of porting SELinux-based mandatory access control from the desktop domain to Android. While SE Android focused on the Android OS, *FlaskDroid* [41] demonstrated how SELinux' type enforcement can be extended into the userspace components of the Android application framework and benefit privacy protection.

Prior work specifically addressed the lack of privilege separation between the different security principals on Android. *AdDroid* [183] and *AdSplit* [223] both propose separating advertisement code into separate processes. *LayerCake* [210] investigated the more general problem of secure cross-application interface embedding on Android, e.g., integrating ad libraries or social network plugins into the host app's UI while mitigating common threats such as click fraud, overlays, or focus stealing. *Compac* [266] demonstrates the applicability of stack inspection in conjunction with ICC tagging to establish per-component access control for Android apps.

Inlined Reference Monitoring A parallel line of work [21, 66, 127, 284] has investigated inline reference monitoring [77] for enforcing more fine-grained and dynamic access control policies for privacy protection. These works were mainly motivated by the deployability benefits of binary rewriting as a foundation for IRM in contrast to OS modifications, which empower end users to enhance their privacy independently from platform developers and device vendors.

IRM solutions on Android currently make the inherent tradeoff of abandoning a strong security boundary between untrusted code and reference monitor, and hence their attacker model focuses on curious-but-benign applications rather than on malicious code. Moreover, modifying third-party code involves legal considerations. Most recent advances in this field [20] introduce application virtualization techniques to Android to avoid third-party code modifications and separate the reference monitor from untrusted code.

Actors' Roles

The platform developers are able (**A1**.●, **R1**.● and **R2**.●) to integrate more advanced access control models, to offer better privilege separation between third party security principals, and to provide means to integrate external security apps. The lack of support for third-party security apps is particularly noticeable for the platform developer actor, since Android's security philosophy shifts responsibility for privacy protection to end users by forcing them to grant/deny permission requests and by allowing them to load applications from arbitrary sources (i.e. to bypass controlled distribution channels like markets). Furthermore, the problem of missing privilege separation could also be alleviated by ad network providers (**A4**.●, **R4**.◐) by refraining from clearly unacceptable behavior and by implementing security best practices.

Binary rewriting solutions for IRM currently need to be deployed by end users (**R8**.●), who also need to configure policies. Their technical approach would also allow software distribution channels or toolchain providers to implement IRM solutions for apps they distribute/create (**R2**.● and **R7**.●).

Lesson Learned

Android adopted design principles from earlier high-assurance systems, and research has proposed valuable access control extensions to their implementations on Android. Although most of the proposed OS extensions are not based on a concrete requirements analysis but rather on postulated challenges, the recent developments of Google's AOSP have a posteriori validated this research; and, in fact, research results can be found in current real-world deployments within the bounds imposed by Google's business model (for instance, SELinux MAC & KNOX [153], dynamic permissions, AppOps, VPN apps, after-market ROMs). Research ideas for privilege separation within app sandboxes, in contrast, should be pushed to maturity and have to be brought to the attention of platform developers. Like mash websites that combine various security principals that are now privilege separated by the browser's sandboxing mechanims, mobile apps that mash various security principals require an adequate privilege separation. IRM solutions are an interim idea, but do not take the user out of the loop (see Section 2.4.2) and are limited in their security guarantees.

Since access control enforcement on Android has been well studied, the research community should shift focus to the canonical challenges of policy generation and verification. Almost no attention has been given to developing useful and real policies. Drawing from experience on desktop systems, policies are moving targets that require decades to develop; research for mobile systems should support this process. In particular, Android's strong requirement for sharing functionality between apps and the shift to privacy protection are unexplored for global policies. Moreover, at the moment enforcement mechanisms on Android are implemented as best-effort, and the history of OS security has shown the need for verifying complex enforcement mechanisms and their policies.

Our assessment (Permission Revolution): Retrofitting Android with mandatory access control has created valuable ideas that influenced real-world deployments. Better privilege separation of apps should be pushed to maturity. The research community should now refocus on open challenges for policy generation and system verification.

2.4.3 Webification

An ongoing trend for mobile apps is *webification*, the integration of web content into mobile apps through technologies like WebView. Seamless integration of apps with HTML and JavaScript content provides portability advantages for app developers. Through its APIs, WebView allows apps a rich, two-way interaction with the hosted web content: Apps can invoke JavaScript within the web page, and also monitor and intercept events in the page as well as register interfaces that web content can invoke to use app-local content outside the WebView sandbox. By now, *mobile web apps* make up 85% of the free apps on Play [152, 163].

Challenges

The webification of apps raises new security challenges that are unique to appified mobile platforms.

Foremost, the two-way interaction between a host app and its embedded web content requires app developers to relax the WebView sandboxing. This enables app-to-web and web-to-app attacks [152, 155, 163]. In app-to-web attacks, malicious apps can inject JavaScript into hosted WebViews to extract sensitive user information and use the WebView APIs to navigate the WebView to untrusted websites. In web-to-app attacks, untrusted web content (possibly also injected into an insecure HTTP/S connection [163]) can leverage the JavaScript bridge to the host app to escalate its privileges to the level of its hosting app's process to access local system resources. In particular, popular web app creator frameworks, such as PhoneGap, open a large attack surface for those kind of attacks through their large web-to-app and app-to-web interfaces [155].

Further, it has been shown [163, 262] that data flows between apps that host different web origins can cross domains through the default Android ICC channels, enabling cross-site scripting and request forgery attacks by malicious apps or untrusted web content within an app. Specifically on mobile platforms, various means enable code to be injected into web content and cross-site scripting attacks to be conducted [128].

Countermeasures

To solve the new security challenges raised by webification, different defense strategies have been proposed: *NoFrak* [155] extends the PhoneGap framework with capability-based access control for web origins to restrict access by web content to the functionality of the JavaScript bridge. Along the same lines, *NoInjection* [128] adds sanitization to the bridge of PhoneGap to prevent code injections. *Morbs* [262] proposes an extension to the Android application framework to attach origin information on ICC channels that can cross origin between apps, thus enabling apps to apply a same-origin policy and prevent the reported cross site scripting and request forgery attacks. Additionally, different modifications to the Android WebView and Android IDEs have been discussed [163, 262], such as supporting whitelisting of web origins that have access to the JavaScript bridge; displaying the security of WebView connections to the end user; or lint tools to warn app developers about insecure TLS certificate validation in WebViews.

Actors' Roles

Fundamentally, platform developers are required to integrate better isolation of web origins in WebViews and support origin-based access control on data flows (**R1**.●). Additionally, providers of web app frameworks and app publishers are responsible for securing their web-to-app and app-to-web bridges (**R4**.● and **R6**.●).

Lesson Learned

The trend towards web apps and usage of web technologies lowered the hurdle for writing apps even more. However, some of the same mistakes known from web applications in browsers were replicated and new problems arose. Cross-origin and web-to-app/app-to-web vulnerabilities constitute serious security challenges for the move towards web apps. However, since such issues are fixable by platform developers and do not require tens of thousands of developers or millions of end users to adopt new security mechanisms, we think this trend is worth pushing in the future.

Our assessment (Webification): Using standard web technology for building apps has proven satisfactory, if somewhat initially shaky. After well-known web security issues have been fixed and integrated with the platform's app sandboxing, this trend should continue.

2.4.4 Programming-induced Leaks

This section deals with challenges and countermeasures regarding data leaks caused by developer errors for apps, frameworks, and libraries.

Challenges

Android provides a comprehensive set of APIs for app developers. A fraction of these APIs are security-related and provide interfaces for Android's permission system, secure network protocols and cryptographic primitives. Prior work has investigated the quality of security-related API implementations: Fahl et al. [82] investigated security issues with customized TLS certificate validation implementations in Android apps and found widespread, serious problems with how developers used TLS. In follow-up work, they conducted developer interviews to learn the root causes of misusing Android's integrated TLS API and found that the current API is too complex for many developers [85]. Although Android provides safe defaults, in ≈95% of the cases app developers decided to implement customized certificate validation mechanisms, the result being an active MITMA vulnerability.

An analysis on the programming quality of cryptographic primitives such as block ciphers and message authentication codes in Android apps by Egele et al. [70] found that 88% of the analyzed apps made at least one mistake when using those primitives. The authors came to the conclusion that Android's default configuration for cryptographic primitives is not safe enough and that the API documentation in this area is poor. It was also found that apps load code via insecure channels (e.g., http) without verification of the loaded code [188]. Of the hereby analyzed apps, 9.25% are vulnerable to insecure code loading, meaning attackers can inject malicious code into benign apps and turn them into malware. The authors came to the conclusion that this is an API issue, since Android's API does not provide secure remote code loading.

Countermeasures

MalloDroid [82] is a static analysis tool to detect broken TLS certificate validation implementations in Android apps. Fahl et al. [85] propose a redesign of Android's middleware/SDK to prevent developers from willfully or accidentally breaking TLS certificate validation. *SMV-Hunter* [233] is a similar approach, additionally applying dynamic code analysis techniques. *CryptoLint* [70] is a static analysis tool to detect misuse of cryptographic APIs on Android. *CHEX* [151] is a static analysis tool to automatically detect component hijacking vulnerabilities. *ScanDroid* [49] is a modular data flow analysis tool for apps, which tracks data flows through and across components. *AndroidLeaks* [100] is a large-scale analysis tool to detect privacy leaks in apps with the intention to reduce the overhead for manual security audits. *FlowDroid* [16] applies static taint analysis techniques to detect (un-)intentional privacy leaks in Android apps.

Actors' Roles

Apps that misuse the above security related APIs leave their apps vulnerable to other apps installed on the device (**C2**.●), to malicious dynamic code loading (**C5**.●) or network attacks (**C6**.●).

A common conclusion of the above API misuse studies is that Android's API design does not provide safe defaults (**A1**.◐) in many cases [70]) and when it does, these defaults often do not match the average developer's needs [85] (**A4**.●, **A5**.●). A study to identify the root causes of these issues conducted with Android developers [85] suggests a redesign of existing security related APIs with the developer's needs in mind (**R1**.◐). Better toolchain support to support secure API usage (**R7**.◐) could help the developers of apps (**R5**.●) and library providers (**R4**.●) to write more secure code. App markets (**R3**.◐) could run analyses on apps to prevent insecure apps from being installed on end users' devices.

Lesson Learned

Previous research uncovered numerous programming issues. A high number of (new) developers code (mobile/web) apps, and security APIs seem to pose a severe challenge for many of them. Developer interviews illustrated that many inexperienced developers write (mobile/web) apps and struggle to provide the basic functionality, which leaves no room for security and privacy considerations. Many of the provided security APIs allow for very detailed configurations, which seem to overwhelm the average developer and result in insecure/improper selection of security options. Developers are on the front line of the security battle and many of them are currently overburdened. However, user studies with developers [85] illustrate that platform developers could modify the current API design to achieve better security by making APIs more developer-friendly. We argue that it should become common practice to use developer studies to test and improve security and privacy APIs.

Our assessment (Programming-induced Leaks): Existing work on redesigning and simplifying usage of APIs and security-related tools should be extended and complemented by research on currently unexplored areas of developer usability.

2.4.5 Software Distribution

Software distribution in the appified world has changed from a decentralized to a centralized model.

Challenges

Android's ecosystem has piqued the interest into investigating the impact of its software distribution channels for the protection of end users against malicious apps. A second challenge is the protection of app developers against common problems such as piracy.

App Piracy and Malware Incentives Pioneering work investigated the incentives of malware developers and the state of malware for modern smartphone operating systems like iOS and Android [90]. The authors discovered that the most common malware activities were collecting user information and sending premium-rate SMS messages. This work predicted that in the future, with proliferation of the app markets and advertisement networks for mobile platforms, ad fraud will be a major incentive for malware authors. This prediction has been proven accurate by different follow-up studies [62, 101, 258, 290, 291, 295]. With the exception of a dedicated malware detection analysis [295], these studies focused on *re-packaged* (also noted as *cloned* [62, 101] or *piggybacked* [290]) apps, which have been identified as a major malware distribution method. The common bottom-line of all works (except one [295]) is that markets contain a noticeable number of re-packaged apps. Although all studies found trojan-like malware in the markets, the vast majority of re-packaged apps have been modified to siphon ad revenue from the original app authors (e.g., by exchanging the ad lib or ad identifier), thus suggesting that plagiarists of apps are fiscally motivated. Hence, this majority of re-packaged apps is not strictly malware in the sense that they harm the end user, but instead financially harm the affected app developers [101].

The implication of this research is that besides the known open challenge of protecting end users from malware distributed over markets, another pressing issue is the protection of app developers against plagiarism. Both are important factors in maintaining a healthy appified ecosystem, which needs to be achieved primarily by app markets. A particular challenge towards this goal is that plagiarism not only occurs within a market, but also across markets. To fight plagiarism, some alternative markets like Amazon's App Store require the app developers to participate in their DRM solutions—with limited success [150]. Moreover, the technical enabler for repackaging apps has to be considered: Android apps are signed by their developers and the signature is used to verify install-time integrity of the installation package and to implement a same-origin update policy. Thus, app developer certificates can be (and are, by default) self-signed certificates whose signature of app packages can be simply replaced with a new signature. This allows re-packaging of apps with a low technical knowledge and effort.

Application Signing Issues Recent work [80] brought up the central role of app markets in appified ecosystems as a new threat for their users. Due to their central role and power when distributing apps, app markets have enormous potential to cheat on their users by withholding apps or updates. A central security mechanism for software distribution is the prior mentioned app signing with self-signed certificates. Investigations [26, 47, 80] illustrate that the way app developers and

publishers handle the current app signing mechanism undermines the mechanism's intention: Many developers and publishers use one single key to sign up to 25,000 apps. Without having effective revocation mechanisms at hand, such practices are a serious threat to Android users. For instance, Android allows developers to define permissions that are only available to apps with the same origin (i.e., signing key) in order to establish secure ICC. This same-origin assumption (and with it secure ICC) is defeated by these inappropriate app certification practices.

Countermeasures

Different market-enabled solutions have been proposed to address the malware problem: *Meteor* [27] addresses security issues arising from multi-market environments by providing the same security semantics as for single-markets (e.g. kill switches and developer name consistency). *MAST* [47] ranks apps based on their attributes and helps targeting scarce malware analysis resources to apps with highest potential of being malicious. *Application Transparency* [80] addresses Android's application signing issues. It introduces different kinds of cryptographic proofs that allow users to verify the authenticity of apps offered on app markets.

Naturally, different analysis methods evolved to identify malware: *DNADroid* [62] is an approach to detect pirated apps in markets by applying program dependency graphs for methods in candidate apps. *RiskRanker* [113] proposes a proactive zero-day malware detection. *CHABADA* [108] takes a different approach from the prior malware detection tools by relying on anomaly detection: by grouping apps from same categories (e.g., games) by their protected API usage patterns, malicious apps stick out as outliers from those sets.

Ernst et al. [78] divert from the adversarial trust assumptions between app vendor and market operator in prior works by relying on a collaborative verification. Assuming that benign developers will co-operate by annotating their code such that it can be effectively verified, while malicious apps can be reliably rejected, this could enable high-assurance app stores.

AppInk [289] aims at deterring app repackaging through dynamic watermarking of apps. Through an IDE extension, app developers can encode watermarks as triggerable code in their app that can be checked dynamically by a companion app to confirm authorship.

Actors' Roles

Platform developers (**A1**.●, **R1**.●) are responsible for fixing key signing issues and allowing for secure distribution of apps in the ecosystem, for instance, distribution of encrypted application packages and full support for PKIs. Additionally, end users (**R8**.◐) could run malware detection software on their devices. However, this would require more effective support for malware detection from the platform developers (see Section 2.4.2).

App markets (**A3**.◐, **R3**.◐) with their central role in the software distribution process have an enormous impact on security. To prove their correct operations, they can add accountability features [80]. However, also app developers (**A5**.●, **R5**.●) and publishers (**A6**.●, **R6**.●) bear full responsibility for misusing app signing recommendations and have the potential to fix these issues in the future.

Lesson Learned

Appification has created an interesting paradigm shift here. Software distribution and installation have become highly centralized. Users typically go to a single app market to search for and install their apps. With their central role in the appified ecosystem, app markets' impact on overall security is enormous. They serve as a line of defense in the fight against malware and could also implement one or more of the many proposed app vetting technologies to protect their users against buggy apps. On the other hand, app markets can also serve as powerful attackers against their own users. They can act as malware distributors or withhold apps or updates.

Although app markets are in a very powerful position, not many of the security and privacy mechanisms proposed by researchers have been adopted by app markets as of today. However, when it comes to privacy, it is potentially not in the best interest of an app market to protect its users. App markets' major motive is monetization by selling apps to their users. As was shown in our systematization, particularly the solutions proposed by researchers to improve users' awareness and control of privacy issues often would require the app markets' cooperation. However, less installs and less lucrative advertising potential could potentially harm app markets' interests. Thus, one result of our work is that researchers should look for additional actors in the ecosystem that could assist in improving users privacy. In particular, app publishers and generators as a strongly emerging pattern for software distribution [80] have not yet received any attention, although their influence on the ecosystem can be considerable. It is unclear to which extent publishers and app generators are trustworthy or are harming the security of apps (e.g., following security best practices) and the privacy of users (e.g., adding tracking code).

Our assessment (Software Distribution): Centralizing software distribution has proven successful for protecting end users against malicious software and for fighting piracy, and should be retained. The threat of malicious app markets is manageable, with countermeasures (almost) ready to be deployed for market-scale application sets. Trustworthiness of app publishers and generators as emerging actors has to be evaluated and established.

2.4.6 Vendor Customization/Fragmentation

Fragmentation in appified ecosystems is a wide spread phenomenon since many hardware and software vendors compete for the customer base in the ecosystem.

Challenges

The Android ecosystem is fragmented at two different levels: First, Android devices are shipped with different OS versions customized by different vendors. Second, vendors ship their devices with custom system apps. Different works investigated the impact of vendor customizations on the permission enforcement on Android [1, 112, 160, 277, 283] that led to a large number of overprivileged system apps [277]. Moreover, vendor customization significantly increases the phone's attack surface. Vendors introduce higher-privileged apps that act as confused deputies [160] or misconfigurations at framework layer [1], both of which allow unprivileged apps access to protected functionality. Recently, the impact of vendor customizations of the device drivers [292] has been investigated and the study reports very similar results:

customizations of Android to fit the vendor-specific hardware have significantly increased the attack surface of the platform and provided attackers access to highly sensible functionality.

Countermeasures

As of today no research has been conducted to investigate countermeasures to challenges that stem from fragmented appified ecosystems.

Actors' Roles

Vendor customizations, and thus device vendors, are responsible for the security degradations caused by fragmentation and customization (**A2.●**, **R2.●**).

Lesson Learned

Android's open ecosystem, in contrast to tighter controlled ecosystems like Apple's iOS, allows vendor customization and fosters the fragmentation that comes along with such customizations. Hence, Android's ecosystem illustrates the potential security risks that such an open approach can induce and should be a warning to concurrent or future appified platforms.

Another lesson to be learned from Android is encouraging vendors to use (system) apps instead of OS patches to provide custom hardware support and force Android to become more modular. Forcing vendors to patch the OS was mainly driven by having only two different privilege levels for apps: system and third party. Eliminating the need for OS patches and allowing vendors to define more privilege levels to integrate customization purely at user space level could reduce fragmentation and drastically reduce the attack surface caused by OS modifications. Although prior works found that vendor app developers make the same mistakes as third-party developers, e.g., over-requesting permissions, bugs in more privileged vendor apps could be more efficiently fixed via the standard app update mechanism in contrast to OS updates. Since vendor app and third party app developers presumably make the same classes of errors, efforts to fix those error classes could be focused instead of having to fight two challenges—apps and OS patches.

Our assessment (Vendor Customization/Fragmentation): Allowing different vendors to customize their devices fueled the adoption process of Android as an appification platform. However, customizing the OS core raised new challenges for platform developers and device vendors. Hence, future fragmentation should focus on system apps rather than OS patches.

2.4.7 Software Update Mechanism

Due to centralization of software distribution, app updates are straight forward and can be pushed to millions of users simultaneously. However, fragmentation of the ecosystem makes OS updates very challenging.

Challenges

Application life-cycles are very fast paced and updates for actively maintained apps are published in high frequency to markets [258] from where automated update

mechanisms distribute them to end users. This is even pushed forward with centralizing updates of security critical libraries such as WebView. In contrast, the situation at OS and application framework level is rather bleak. Thomas et al. [247] present a field study of 20,400 Android devices to measure the prevalence of Android platform specific bugs in the wild. They define a metric to rank the performance of device manufacturers and network operators, based on their provision of updates and exposure to critical vulnerabilities. Their central finding is a significant variability in the timely delivery of security updates across different device manufacturers and network operators, since at least 87% of all investigated devices were vulnerable to at least 11 different vulnerabilities.

In addition, the complexity of upgrading the Android OS version induced problems in the permission management across OS versions [283]. This attack class is currently unique to permission-based mobile systems, such as Android, since the attacker does not corrupt the current system or update image, but instead strategically requests permissions and attributes that are available on the *future* OS version.

Countermeasures

No research has thus far investigated countermeasures for challenges that stem from software update mechanisms as implemented on Android. Apart from research, Google has with their latest Android versions changed their update strategy for their Nexus devices [36, 106]. It remains to be seen if other vendors adopt this strategy. Moreover, the *SecUp* [283] app can detect apps that exploit the above mentioned privilege-escalation attack through OS updates.

Actors' Roles

Providing OS updates is responsibility of device vendors (**A2.●**, **R2.●**). Platform developers (**A1.●**, **R1.●**) are responsible for introducing the upgrade privilege escalation attack.

Lesson Learned

Many researchers expect the platform developer to implement their countermeasures. However, even if that should happen—which is rare, as of today devices are not long-term and frequently maintained by vendors except Google—this expectation is causing slow adaption of new mechanisms and contributes to the fragmentation of the ecosystem [180, 247]. This also opens a large window of opportunity for attackers to compromise the system. Interestingly, appified platforms like Android already have a modularization of software at the application layer. This is inspired by classical high-assurance systems like EROS [222] and in fact, the Binder IPC of Android establishes something like a microkernel-like concept on top of the Linux kernel in userspace. We would like to see this modularization extended to allow modular updates of the system so that security updates can be deployed faster to the end user without requiring a full system update. This is an area where appified platforms are way behind traditional operating systems.

Our assessment (Software Update Mechanism): Since most proposed countermeasures rely on OS updates, and OS fragmentation make these very cumbersome, the platform developers should create better update mechanisms, so that security fixes and countermeasures can be more easily deployed.

2.5 Discussion

The central conclusion we draw from this systematization is that, like many new technologies, Android is a story of both victory and defeat. New security mechanisms were introduced without a clear understanding of how these applications would be developed and used, and well-established security mechanisms were reused to meet the expected security needs of the new general purpose computing platform. Some of the these techniques were a great success, while others failed almost entirely. We draw the following meta-conclusion:

Our meta-assessment: Some aspects worked out beautifully, e.g., centralizing software distribution helps to tackle critical security issues and makes fighting piracy and malware easier. Other approaches had initial difficulties, but are now more or less on track after research has helped to identify and bridge them. Examples comprise easier-to-use APIs that have started to replace hard-to-use but well-intended security APIs over the last few years, as well as the concept of Webification that has enabled more developers to produce their own apps. However, some approaches should be re-thought from the beginning and arguably abandoned for designs of future OSes: Permission dialogs for end users should be removed entirely, since they failed for the same reasons warning messages have failed since the dawn of computing.

In this chapter, we gave an in-depth analysis of the appified Android ecosystem, and its successes and drawbacks for security and privacy compared to conventional software ecosystems. We characterized the model's different actors and their capabilities to mitigate security and privacy threats. We systematized existing research, and classified attacker models and capabilities. We find that end users are limited in their ability to protect their own security and privacy, and that more responsibility lies "upstream", with app developers, API developers, device vendors and platform providers. In the following chapter, we explore empirically how app developers' security outcomes are influenced by documentation, the responsibility for which generally lies with API developers, or, in the case of official Android documentation, with the platform provider.

Chapter 3

On the Impact of Information Sources on Code Security

Disclaimer: *The contents of this chapter were previously published as part of the conference paper "You get where you're looking for: The impact of information sources on code security", presented at the 2016 IEEE Symposium on Security and Privacy. This research was conducted as a team with my co-authors Christian Stransky, Doowon Kim, Michelle L. Mazurek, Sascha Fahl, and Michael Backes; this chapter therefore uses the academic "we". Christian Stransky, Sascha Fahl, Michelle L. Mazurek and I designed, conducted and evaluated the online survey and controlled laboratory experiment. Doowon Kim helped in conducting the controlled experiment. Sascha Fahl and I designed the API call confirmation study. Christian Stransky and Sascha Fahl conducted and evaluated the API call search. Christian Stransky, Doowon Kim, Michelle L. Mazurek, Sascha Fahl and I co-wrote the paper. We are grateful to Sven Bugiel for assisting in the controlled experiment, and Marten Oltrogge for assisting in the API call search. This chapter builds on knowledge of common problems with Android at the time; the controlled experiment uses a late-2015 version of Android, as well as the resources available at the time.*

3.1 Motivation

Mobile devices in general and Android in particular are a rapidly growing market. Globally, mobile digital media has recently surpassed desktop and other media [244]; billions of users and devices with millions of apps installed attract many (new) developers. Previous research has found that many of these mobile apps have poorly implemented security mechanisms, potentially because developers are inexperienced, distracted or overwhelmed [19, 50, 53, 70, 74, 80, 82, 84, 85, 87, 93, 151, 175, 188, 233, 268, 277, 293]. Developers tend to request more permissions than actually needed, do not use TLS or cryptographic APIs correctly, often use insecure options for Inter Component Communication (ICC), and fail to store sensitive information in private areas.

Some previous work attempts to assess root causes for these programming errors. A frequent conclusion is that APIs are too complicated or insufficiently documented. Anecdotal reports indicate that developers use a search engine for help when they encounter an unfamiliar security issue. The search results often lead to official API documentation, blog posts, or Q&A forums such as Stack Overflow[1]. For example, Fahl et al. [82, 84, 85] interviewed developers whose use of pasted code snippets from Stack Overflow made them vulnerable to Man-In-The-Middle attacks.

[1] http://stackoverflow.com

These anecdotes set the stage for our work: While many developer issues have been identified in recent years, we know only very little about how these security issues make their way into apps, and most of what we know remains unsubstantiated. In this paper, we assess the validity of these anecdotes by exploring the following research questions:

- What do Android developers do when they encounter a security- or privacy-relevant issue?
- Which information sources do they use to look up security- or privacy-relevant questions?
- Does the use of Stack Overflow really lead to less secure code than the use of other resources?
- Is the official Android documentation really less usable, resulting in less functional code compared to other resources?

We are the first to address these questions systematically rather than anecdotally, shedding light on the root causes of security-related programming errors in Android apps. In order to understand these causes, we first conducted an online survey of 295 developers with apps listed in the Google Play marketplace, covering how they handle both general and security-specific programming challenges in their daily work. We found that most developers indeed use search engines and Stack Overflow to address security-related issues, with a sizable number also consulting the official API documentation and a few using books.

Based on the results of this study, we conducted a laboratory user study with 54 student and professional Android developers, assessing how they handle security challenges when given different resources for assistance. Participants were assigned to one of four study groups, in which we isolated *conditions*: free choice of resources, Stack Overflow only, official Android documentation only, and books only. Each participant was asked to complete four programming tasks that were drawn from common errors identified in previous work: A secure networking task, a secure storage task, an ICC task, and a least permissions task. We analyzed the correctness and security of the participants' code for each task as well as how they employ the resources we permitted them to use. Our results validate the prior anecdotal evidence: Participants using Stack Overflow were more likely to be functionally correct, but less likely to come up with a secure solution than participants in other study conditions.

To place these results in context, we also surveyed the quality of Stack Overflow Q&As. We first analyze the relevance and security implications of the 139 Stack Overflow threads accessed by our subjects. We found that many of the threads contain insecure code snippets, and that those threads are equally as popular as threads with only secure snippets.

To establish ground truth, we also apply static analysis to a random sample of 200,000 free apps from the Google Play market in order to investigate if the code written in the context of our laboratory study can be found in the wild. We find that our programming tasks were highly representative for the typical Android programmer, as 93.6% of all apps we analyzed used at least one of the API calls our participants generated during the study. Our analysis also finds that many of the security errors made by our participants when using these APIs also appear in the wild. For example, most of the custom hostname verifier implementations we found

in real-world apps implement insecure hostname verification, which is also true for the code written by our participants.

Taken together, our results confirm an important problem: Official API documentation is secure but hard to use, while informal documentation such as Stack Overflow is more accessible but often leads to insecurity. Interestingly, books (the only paid resource) perform well both for security and functionality. However, they are rarely used (in our study, one free choice participant used a book). Given time constraints and economic pressures, we can expect that Android developers will continue to choose those resources that appear easiest to use; therefore, our results firmly establish the need for secure-but-usable documentation.

The rest of this chapter proceeds as follows: In Section 3.2 we review related work. Section 3.3 describes our online survey of Android developers who have published in the Play market, Section 3.4 describes the design of our laboratory study, and Section 3.5 reports its results. In Section 3.6 we present our analysis of Stack Overflow posts and in Section 3.7 we present the ground truth from our static code analysis. Section 3.8 discusses some limitations of our work. Finally, in Section 3.9 we discuss our results and conclude.

3.2 Related Work

In this section, we discuss related work in three key areas: Security and privacy flaws in otherwise benign mobile apps, efforts to understand how mobile developers make security- and privacy-relevant decisions and prior research exploring online Q&A resources such as StackOverflow.

Security Flaws in Mobile Apps. Many researchers attempted to measure the incidence of security flaws in otherwise benign mobile apps. Fahl et al. found that 8% of 13,500 popular, free Android apps contained misconfigured TLS code vulnerable to Man-In-The-Middle attacks [82]. Common problems included accepting all certificates without verifying their validity and not checking whether the name of the server currently being accessed matches the hostname specified on the certificate it provides. In follow-up work, the same research team extended their analysis to iOS and found similar results: Using a Man-In-The-Middle attack, they were able to extract sensitive data from 20% of the apps [85]. Another examination of TLS code, this time in non-browser software more generally, found similar flaws in many Android and iOS applications and libraries [99]. In more recent work, Onwuzurike and De Cristofaro found that the same problems remain prevalent several years later, even in apps with more than 10 million downloads [178]. Oltrogge et al. [175] investigated the applicability of certificate pinning in Android apps. They came to the conclusion that pinning was not as widely applicable as commonly believed. However, there was still a huge gap between developers who actually implement pinning and apps that could use pinning.

Egele et al. examined the use of cryptography – including block ciphers and message authentication codes – in Android applications and found more than 10,000 apps misusing cryptographic primitives in insecure ways [70]. Examples included using constant keys and salts, using non-random seeds and initialization vectors, and using insecure modes for block ciphers.

Many problems also exist with the use and misuse of app permissions, device identifiers, and inter-application communication. Enck et al. analyzed 1,100 free Android apps and reported widespread issues, including the use of fine-grained

location information in potentially unexpected ways, using device IDs for fingerprinting and tracking (in concert with personal identifiable information (PII) and account registration), and transmitting device and location in plaintext [74]. Chin et al. characterized errors in inter-application communications (*intents*) that can lead to interception of private data, service hijacking, and control-flow attacks [53]. Felt et al. [87] analyzed how app developers use permissions and report that many request unnecessary permissions. The authors identify incomplete documentation for developers as one major root cause of this problem. Work by Poeplau et al. reported that almost 10% of analyzed apps load code via insecure channels (e.g., http or the SD card), which can allow attackers to inject malicious code into benign apps in order to steal data or create malware [188].

Enck et al. [73] presented TaintDroid – a tool that applies dynamic taint tracking to reveal how apps actually use permission-protected data. They uncovered a number of questionable privacy practices in apps and motivated enhancements to Android's original permission system and access control on inter-component communication.

In this chapter, we consider how the information sources developers use contribute to these kinds of errors and problems.

Understanding Developers. Many of the flaws discussed above arose from developer mistakes and misunderstandings. In interviews with developers who made mistakes in TLS code, Fahl et al. found that problems arose from several sources, including developers who disabled TLS functionality during testing and never reenabled it, developers who did not understand the purpose of TLS or the possible threat scenarios, and problems with default configurations in frameworks and libraries [85]. Georgiev et al. also reported that confusion about the many parameters, options, and defaults of TLS libraries contributed to developer errors [99]. Both papers noted that developer forums such as Stack Overflow contained many suggestions for avoiding TLS-related error messages by disabling TLS features, without warning about the potential security consequences. Many developers use these resources to solve security- and privacy-related problems [22]. Similarly, Egele et al. discussed how poor default configurations and confusing APIs, along with insufficient documentation, may contribute to errors using cryptographic primitives [70].

In a non-mobile context, Leon et al. found that many popular websites used invalid or misleading P3P compact policies, which are tokens used to summarize a website's privacy policy for automated parsing [145]. Their manual analysis suggested that while many mistakes likely resulted from developer error, others resulted from attempts to avoid Internet Explorer's cookie filtering mechanism, and appeared to rely on suggestions from forums like Stack Overflow for avoiding this filtering. While these works on TLS and compact policies observed problems related to Stack Overflow and similar sites, our work uses a controlled experiment to compare the impact of different information sources.

Other flaws, particularly those related to privacy, are caused when developers do not sufficiently consider the implications of their decisions. In interviews with mobile developers from companies of various sizes, Balebako et al. found that privacy policies are not considered important and that privacy concerns are frequently outweighed by concerns about revenue, time to market, and the potential for any data that can be collected to someday be useful [23]. In a follow-up survey, the same authors found that many developers are not aware of the privacy or security implications of third-party advertising and analytics libraries they use [22]. These

findings provide valuable insight into developers' perspectives; our work extends these perspectives with empirical observation of developer behavior.

Other researchers considered how to improve developers' decision making. Jain and Lindqvist propose a new location-request API designed to promote privacy-preserving choices by developers [126]. Fahl et al. suggested providing TLS as a service within a mobile OS that supports a separate development mode [85]. Similarly, Onwuzurike and De Cristofaro provided a code snippet for correctly using self-signed certificates during development but not production [178]. Our work extends Jain and Lindqvist's methodology to empirically evaluate developers' decisions.

Collectively, these findings suggest that helping well-meaning mobile developers to make better security- and privacy-relevant decisions could have a large impact on the overall mobile ecosystem. In this chapter, we expand on these findings by using a controlled lab study to quantify how documentary resources impact security and privacy outcomes.

Exploring Online Q&A Resources. The software engineering and machine learning communities explored how developers interact with Stack Overflow and other Q&A sites. Much of this research focused on what types of questions are asked, which are most likely to be answered, and who does the asking and answering [29, 30, 161, 251, 255, 263].

Other research considered the quality of questions and answers available on Q&A sites – including general sites not specifically targeting programming [24, 127, 190]. These works are generally intended to support automated identification and pruning of low-quality content. In contrast to the studies described above, our work does not describe or measure broad trends in how Stack Overflow is used; nor do we consider how to automatically classify content. Instead, we directly consider how existing Stack Overflow content affects the outputs of developers who rely on it.

Others have analyzed Q&A sites specifically in the context of mobile development. Linares-Vásquez et al. investigated how changes to Android APIs trigger activities on Stack Overflow and found that the frequency of questions increases when Android APIs change, particular in the case of method updates [146]. In two related works, Wang et al. mined Stack Overflow posts to identify mobile APIs (Android and iOS) that frequently give developers trouble. They proposed that this data can be used both to improve documentation for these "hotspots" and to help API providers improve the design of their APIs to better support developer needs [264, 265]. In a similar vein, Nadi et al. analyze Stack Overflow posts to identify difficulties that developers commonly have with Java cryptography APIs [166]. While these works used Stack Overflow to identify trouble spots within APIs, we instead start from known trouble spots in security and privacy and measure how information sources, including Stack Overflow, directly affect the code developers write.

3.3 Survey of Android Developers

To understand the challenges app developers face during the implementation of security-critical app components, we conducted an online survey of Android developers covering their experience, their programming habits, and the resources they use. Results from this survey helped motivate the design of our lab experiment (Section 3.4). In this section, we briefly discuss the design of this survey as well as the

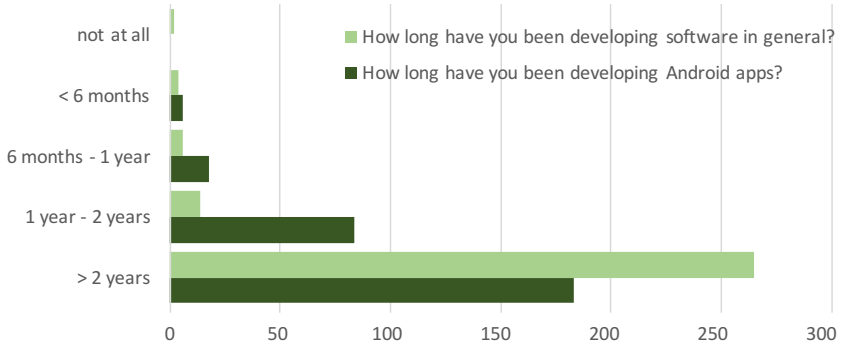

FIGURE 3.1: How long participants in our online survey have been developing software, both in general and specifically for Android.

results. The online study was approved by the University of Maryland Institutional Review Board.

We collected a random sample of 50,000 email addresses of Android application developers listed in Google Play (the official Android application market). We emailed these developers, introducing ourselves and asking them to take our online survey. A total of 302 people completed the survey between April 2015 and October 2015. Seven participants were removed for providing answers that were nonsensical, profane, or not in English. Results are presented for the remaining 295.

Education and Experience. Most participants (91.2%, 269) had been developing software for more than two years; 63.1% (186) had been developing Android apps specifically for more than two years, as shown in Figure 3.1. About half of them (48.7%, 147) had developed between two and five apps; however, 73.5% (218) of all participants reported that they do not develop Android apps as their primary job.

Almost half of the participants had formally studied programming at the undergraduate (27.8%, 82) or graduate level (18.6%, 55). Most of the remaining developers reported being self-taught (41.2%, 121). Most participants had never taken any classes or training related specifically to Android programming (81.3%, 239) or to computer or information security (56.6%, 167).

As we discuss in Section 3.5.1, these demographics have some similarity with our lab study participants; however, survey participants as a whole reported less formal education than lab participants.

Security and Permissions. We also asked participants about three security-related issues they might have encountered during app development: HTTPS/TLS, encryption, and Android permissions. These results provide some context for the security tasks used in our lab study.

About half of the developers (144) said that their Android app(s) use HTTPS to secure network connections; of those, 80.6% (116) had looked up information on HTTPS- or TLS-related topics at least once, but only 11.1% (16) did so more frequently than once per month. The most popular resources among these 116 were Stack Overflow (43.1%, 50) and a search engine such as Google (37.1%, 43); only 8.6% (10) mentioned the official Android documentation. Interestingly, a few (2.6%, 3) mentioned asking for help from certification-related companies such as certificate

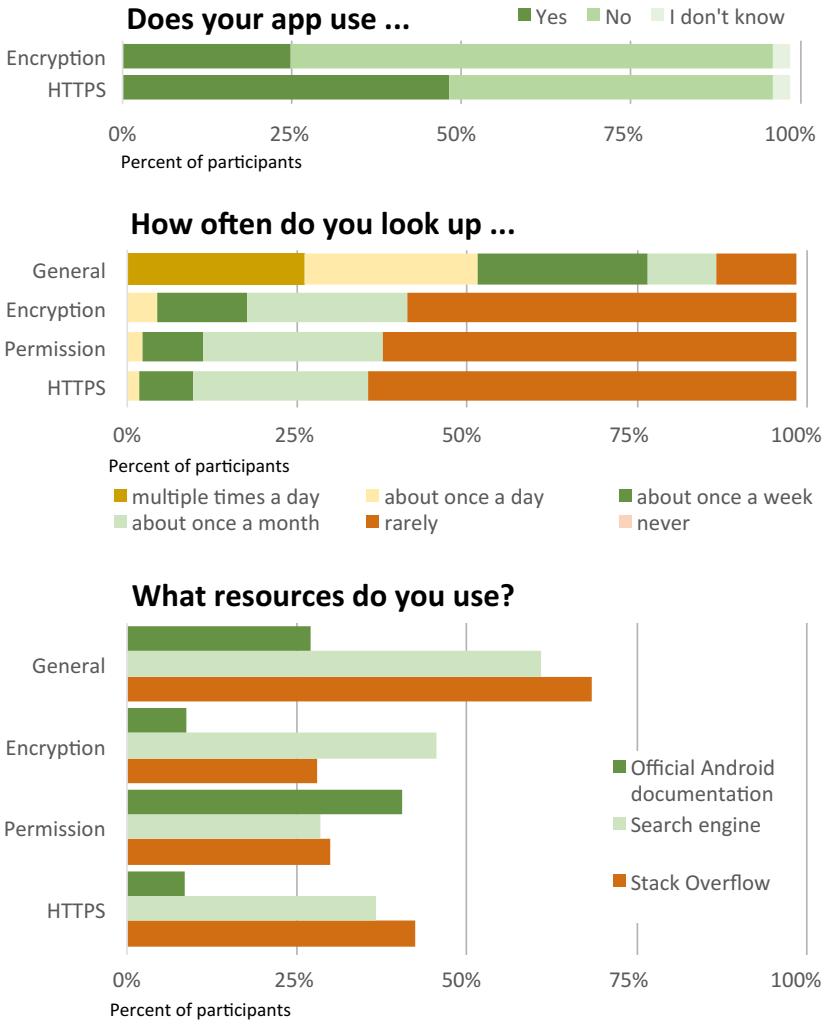

FIGURE 3.2: Highlights of resource questions from our online developer survey. How many participants work on apps that include encryption or HTTPS (top), how often participants look up information when solving general programming problems or security-related Android problems (middle), how many participants mentioned using each of the most popular resources for solving general programming problems or security-related Android problems (bottom).

vendors or hosting companies. A large majority of respondents (78.4%, 91) said they did not handle HTTPS or certificate problems differently from other problems.

Fewer participants (25.1%, 74) had used encryption to store files. Of these, almost all (90.5%, 67) had looked up encryption-related topics at least once, but again the vast majority did so once a month or less (82.1%, 55). The primary sources were once again search engines (mentioned by 31 participants, 46.3%) and Stack Overflow (28.4%, 19). Six of the 67 (9.0%) mentioned the official Android documentation, and two (3.0%) mentioned books. As with HTTPS, the majority (58, 86.6%) solved encryption problems similarly to other problems.

Responses to questions about Android permissions were somewhat different. As with HTTPS and encryption, most (74.9%, 221) reported they had looked up permissions information at least once, and a large majority of them did so once per month or less (84.2%, 186). However, participants who had looked up permission

information favored official documentation (41.2%, 91) over search engines (29.0%, 64) or Stack Overflow (30.3%, 67) on that topic. One participant wrote that "[I] don't have to Google. [I] go directly to Android developer resource" for authoritative information.

Development Resources More Generally. We also asked (free response) about the resources participants use when they encounter programming problems in general. The results are similar to those for security-specific problems. Large majorities mentioned Stack Overflow (69.5%, 205) and a search engine (62.0%, 183). Although this question did not specifically mention Android programming, 27.5% (81) also mentioned official Android documentation, including APIs and best practices guides.

In a separate question, we asked how frequently participants use any resources when programming for Android. More than half (52.2%, 154) reported looking up Android programming information at least once per day and another 25.4% said at least once per week. Among 35 participants (11.9%) who selected "rarely," 11 (31.4%) explicitly mentioned that while they rarely looked things up now, they had used resources or documentations for help many times a day when they were working on Android projects.

Figure 3.2 illustrates how participants used resources, both for security-related tasks and in general.

Discussion. Overall, these results indicate that many Android developers must deal with security or privacy issues periodically, but do not handle them consistently enough to become experts. This suggests that the quality of documentation is especially critical for these topics. Stack Overflow (and more generally, online search) is the default resource for certificate or encryption problems, as well as programming problems more generally. Permissions, however—perhaps because they are Android-specific and closely associated with the platform itself—are more frequently referenced from the official documentation. These findings validate both the need to understand the impact of the resources on security and privacy decisions generally, and our choice to compare Stack Overflow and the official documentation more specifically.

3.4 Android Developer Study

To examine how the resources developers access affect their security and privacy decision-making, we conducted a between-subjects laboratory study. We provided a skeleton Android app and asked participants to complete four programming tasks based on the skeleton, encompassing the storage of data, the use of HTTPS, the use of ICC and the use of permissions. Each participant was assigned to one of four conditions governing what resources they were allowed to access. We examined the resulting code for functional correctness as well as for security- or privacy-relevant decisions; we also used a think-aloud protocol and an exit interview to further examine how participants used resources and how this affected their programming.

The lab study was also approved by the University of Maryland Institutional Review Board.

3.4.1 Recruitment

We recruited participants who had taken at least one course in Android development or developed professionally or as a hobby for at least one year. Initially, participants were also asked to complete a short programming task to demonstrate competence with Android development. After receiving feedback that the qualification task required too great a time commitment for prospective participants, we instead required participants to correctly answer at least three of five multiple choice questions testing basic Android development knowledge. The bar for qualification was intentionally set low, as we wanted to compare the impact of programming resources for developers with different expertise levels. In addition, the usefulness of our results partially depended on our participants needing to look things up during the programming process.

Participants were recruited in and around one major city in the U.S., as well as in two university towns in Germany. We recruited participants by emailing undergraduate and graduate students (in computer science in general and specifically those who had taken mobile development courses), as well as by placing ads on Craigslist, emailing local hacker and developer groups, and using developer-specific websites such as meetup.com. Prospective participants who qualified were invited to complete the study at a university campus or at another public place (library, coffee shop) of their choice. No mention of security or privacy was made during recruitment. Participants were compensated with $30 in the U.S. or an €18 gift card in Germany.

3.4.2 Conditions and Study Setup

Participants were assigned round-robin to one of four conditions, as follows:

Official Only (official). Participants were only allowed to access websites within the official Android documentation [2].

Stack Overflow Only (SO). Participants were only allowed to access questions and answers within Stack Overflow, a popular crowd-sourced resource for asking and answering questions about programming in a variety of contexts.

Book Only (book). Participants were only allowed to use two books: Pro Android 4 [138] and Android Security Internals [71]. Participants were provided access to the PDF versions of the books, enabling text searching as well as use of indices and tables of contents.

Free Choice (free). Participants were allowed to use any web resources of their choice, and were also offered access to the two books used in condition book.

Conditions official and SO were enforced using whitelist-chrome[3], a Chrome browser plugin for limiting web access.

Participants were provided with AndroidStudio, pre-loaded with our skeleton app, and a software Android phone emulator. The skeleton app, which was designed to reduce participants' workload and simplify the programming tasks, was introduced as a location-tracking tool that would help users keep track of how much time they spent in various locations (at home, at work, etc.) each day.

[2] cf. `http://developer.android.com`

[3] cf. https://github.com/unindented/whitelist-chrome

After a brief introduction to the study and the skeleton app, participants were given four programming tasks in random order (detailed below), with approximately 20-30 minutes to complete each. (The first task was allowed to run longer as participants became acquainted with the skeleton app.) While the short time limit impeded some participants' performance, it also simulated the pressure of writing code on tight deadlines that many app developers face.

Security and privacy were not mentioned during the introduction to the study and skeleton app or in the directions for each task (the HTTPS task and password task do inherently imply some reference to security). We deliberately minimized security priming to account for the fact that security and privacy are generally secondary tasks compared to basic app functionality [23, 61, 85, 97]. Instead, we focus on whether developers – who in real-world scenarios may or may not be explicitly considering security – find and implement secure approaches. This is in line with prior studies examining security and privacy decisionmaking by developers, such as one by Jain and Lindqvist [126].

3.4.3 The Tasks

Each participant was assigned the same four tasks, but in a random order. We took care to implement baseline functionality so that the tasks could be done in any order and so that remaining tasks would still work, even if a previous task had not been successfully completed.

Drawing on related work (as discussed in Section 3.2), we selected four general areas that typically result in security or privacy errors on Android: (1) Mistakes in TLS and cryptographic API handling; (2) storing sensitive user data insecurely, such that it can be accessed by other (unauthorized) apps; (3) using inter-component communications (ICC) in a way that violates least privilege principles; and (4) requesting unneeded permissions. We designed four tasks, detailed below, to exercise these areas respectively.

Secure Networking Task. This task addressed correct usage of HTTPS and TLS in the presence of X.509 certificate errors. The skeleton app connected to a website via HTTP; participants were asked to convert the connection to HTTPS. This required making a minor adjustment to the connection object. More interestingly, we created a certificate for secure.location-tracker.org (a server we configured specifically for this study), but the participant was requested to connect to location-tracker.org, and a matching DNS entry for secure.location-tracker.org did not exist. As a result, participants received a HostnameVerifier exception indicating the certificate name and domain were mismatched. Secure solutions would include creating a custom HostnameVerifier to handle this case or pinning the certificate (although we expected pinning to be too time-consuming for most participants to implement in the study) [4]. We also accepted a participant arguing that the location tracker app should obtain a correct X.509 certificate rather than working around the problem as a secure solution. Insecure solutions that allow a connection to be established include using a HostnameVerifier that accepts all hostnames, or simply accepting all certificates without validation.

ICC Task. This task addressed secure inter-component communication. Participants were asked to modify a service within the skeleton app, in order to make the

[4]Implementing it correctly requires inspecting the server's certificate and using a third-party tool such as the OpenSSL command-line client to generate the pinning information

service callable by other apps. However, participants were asked to limit this access to apps created by the same developer. To accomplish this, participants needed to modify the Android Manifest. An insecure solution would expose the service to be called by any app; this could happen by setting the flag android:exported to true or by declaring intent filters, which set the exported flag to true by default. A secure solution for this task is to define an own permission with protection level 'signature' or 'signatureOrSystem' and assign it as required for the service. A second possible secure solution is to use a sharedUserId among all apps from the same developer, which allows the apps to share resources.

Secure Storage Task. This task focused on secure storage of the user's login ID and password for the remote server. The skeleton app contained empty store and load functions for the participant to fill in; the directions asked the participant to store the credentials persistently and locally on the device. A secure solution would be to limit access only to this app, for example using Android's shared preferences API in private mode. An insecure solution would make the data accessible to third parties, for example by storing it world-readable on the SD card.

Least Permissions Task. In this task, participants were asked to add functionality to dial a hard-coded customer support telephone number. The skeleton app contained a non-functional call button, to which the dialing functionality was to be applied. To solve this problem, the participant needed to use an intent to open the phone's dialing app. One option is to use the ACTION_DIAL intent, which requires no permissions; it opens the phone's dialer with a preset number but requires the user to actively initiate the call. Another option is to use the ACTION_CALL intent, which initiates the call automatically but requires the CALL_PHONE permission. We consider the second solution less appropriate because it requires unnecessary permissions, violating the principle of least privilege.

3.4.4 Exit Interview

After completing each task (or running out of time), participants were given a short exit interview about their experience. Using a five-point Likert scale, we asked whether each task was fun, difficult, and whether the participant was confident they got the right answer. We also asked whether the documentation and resources participants had access to were easy to use, helpful, and correct. We asked free-response questions about whether the participant had used that documentation source before and how they felt the documentation restriction (where applicable) and time crunch affected their performance. We also asked whether and how participants had considered security or privacy during each task. Finally, we asked a series of demographic and programming-experience questions that matched those in our initial developer survey (see Section 3.3).

3.4.5 Data Collection and Analysis

In addition to each participant's code, think-aloud responses, and exit interview responses, we collected browser activity during the session (for participants in all but the book condition) using the History Export[5] extension for Chrome, which stores all visited URLs in a JSON file.

[5] cf. https://chrome.google.com/webstore/detail/history-export/lpmoaclacdaofhlijejogfldmgkdlglj

Scoring the Programming Tasks. For each programming task, we assigned the participant a *functionality* score of 1 (if the participant's code compiled and completed the assigned task) or 0 (if not). To provide a security score for each task, we considered only those participants who had functional solutions to that task. We manually coded each participant's code to one of several possible strategies for solving the task, each of which was then labeled secure or insecure. Based on these categories, each participant who completed a task was assigned a security score of 0 (insecure approach) or 1 (secure approach) for that task. Manual coding was done by two independent coders, who then met to review the assigned codes and resolve any mismatches. All conflicts were resolved by discussions that resulted in agreement. Example secure and insecure approaches for each task are detailed in Table 3.1.

Prior to the conducting the lab study, we verified that functional and secure solutions for each task, such as those described in Table 3.1, were available in each of the official Android documentation, Stack Overflow, and the selected books. This ensured that it was possible (if not necessarily easy) for participants in all conditions to locate a correct and secure answer.

Statistical Methods. For ordinal and numeric data, we used the non-parametric Kruskal-Wallis test to compare multiple samples and Wilcoxon Signed-Rank test to compare two samples. For categorical data, we used Fisher's Exact test. In cases of multiple testing, we report tests as significant if the p-values are significant after applying the Bon Ferroni-Holm correction. To examine correlation between two sets of binary outcomes, we the use Cohen's κ measure of inter-rater reliability.

To examine functional correctness and security across tasks and conditions, while accounting for multiple tasks per participant, we used a cumulative-link (logit) mixed model (CLMM) [282]. As is standard, we include random effects to group each participant's tasks together. For the CLMM, we tested models with and without the participant's status as a professional developer as an explanatory factor, as well as with and without interactions among task, condition, and developer status. In each case, we selected the model with the lowest Akaike information criterion (AIC) [43].

3.5 Lab Study Results

In this section, we discuss our lab study results in terms of functional correctness, security, and participants' use of their assigned resources. We find that while Stack Overflow is easier to use and results in more functional correctness, it also results in less security than the less accessible official API documentation.

3.5.1 Participants

A total of 56 people participated in our lab study (13 in the U.S. and 43 in Germany). Two participants (one from the U.S. and one from Germany) were removed, one due to an error assigning the condition and one because of their refusal to work on the tasks. We report results for the remaining 54.

Our participants were aged between 18 and 40 (mean = 26, sd = 4.70), 85.2% were male (46 participants), and most of them (88.9%, 48) were students. Several were part-time students and part-time professional developers. More than half of participants said they grew up in Germany (51.9%, 28). The next most popular countries of origin were the U.S. (11.1%, 6) and India (9.3%, 5). Table 3.2 shows demographic information for the participants recruited in each country. Using Fisher's exact test, we did not find differences in gender ($p = 0.400$), occupation ($p = 1.00$) or country

Task	API	Details	Security Rating	Explanation
Secure Networking	`javax.net.ssl.HostnameVerifier.verify(host, session)`	`return true`	○	A custom hostname verifier with a correct domain check is rated as a secure solution. Hostname verifiers which accept all hostnames are rated insecure.
		e. g. `host.equals("secure.foo.com")`	●	
	`org.apache.http.conn.ssl.X509HostnameVerifier.verify(host, session)`	`return true`	○	
		e. g. `host.equals("secure.foo.com")`	●	
ICC	`<service>...</service>`	`android:exported=true`	○	A service that has the exported flag set to true, uses intent filters, or uses a normal or dangerous permission is rated as insecure. Services protected with a signature or signatureOrSystem permission are rated as secure.
		`<intent-filter>...</intent-filter>`	○	
		`android:permission`	○	
		`android:protectionLevel=signature`	●	
		`android:protectionLevel=signatureOrSystem`	●	
Secure Storage	`Environment.getExternalStoragePublicDirectory(type)`	-	○	We distinguish three different storage backends: SQLite databases, the file system, and Android's shared preferences. All three can have secure and insecure implementations. Secure implementations store information in an area local to an app; this is the default implementation for SQLite databases and shared preferences. However, both backends can be used to store data such that other apps can access it. The file-system API can be used to either store data locally or externally on a device's SD card. Implementations that store information in an externally accessible way are rated insecure. Implementations that store information locally are rated secure.
	`Context.getExternalFilesDir(type)`	-	○	
	`Context.getFilesDir()`	-	●	
	`Context.getCacheDir()`	-	●	
	`Context.openFileOutput(name, mode)`	`Context.MODE_PRIVATE`	●	
		`Context.MODE_WORLD_READABLE`	○	
		`Context.MODE_WORLD_WRITABLE`	○	
	`Context.getDir(name, mode)`	`Context.MODE_PRIVATE`	●	
		`Context.MODE_WORLD_READABLE`	○	
		`Context.MODE_WORLD_WRITABLE`	○	
	`PreferenceManager.getDefaultSharedPreferences()`	-	●	
	`PreferenceManager.getSharedPreferences(context)`	-	●	
	`Context.getSharedPreferences(name, mode)`	`Context.MODE_PRIVATE`	●	
		`Context.MODE_WORLD_READABLE`	○	
		`Context.MODE_WORLD_WRITABLE`	○	
	`Activity.getPreferences(mode)`	`Context.MODE_PRIVATE`	●	
		`Context.MODE_WORLD_READABLE`	○	
		`Context.MODE_WORLD_WRITABLE`	○	
	`Context.openOrCreateDatabase(name, mode,...)`	`Context.MODE_PRIVATE`	●	
		`Context.MODE_WORLD_READABLE`	○	
		`Context.MODE_WORLD_WRITABLE`	○	
Least Permissions	`new Intent(action, uri)`	`Intent.ACTION_DIAL`	●	Using action_dial is rated secure. However, using action_call and requesting the call_phone permission is rated insecure.
		`Intent.ACTION_CALL`	○	
		`android:name='android.permission.CALL_PHONE'`	○	

● = we rated this solution as secure, ○ = we rated this solution as insecure

TABLE 3.1: Task related API calls and their parameters. With security rating parameters help classify whether a solution is secure.

Assigned condition			
Official: 13	SO: 13	Book: 14	Free: 14
Location of Study			
United States: 12 (22.2%)			Germany: 42 (77.8%)
Gender			
Male: 46 (85.2%)			Female: 8 (14.8%)
Country of Origin			
United States: 6 (11.1%)			Germany: 28 (51.9%)
India: 5 (9.3%)			Others: 15 (27.8%)
Professional Android Experience			
Yes: 14			No: 40
Ages			
mean = 26.0	median = 25		sd = 4.7

TABLE 3.2: Participant Demographics.

of origin ($p = 0.81$) between the randomly assigned conditions. Using the Kruskal-Wallis test, we could not find a difference in ages across the randomly assigned conditions ($X^2 = 2.22$, $p = 0.528$). Both in the U.S. and in Germany, participants were distributed evenly across the four conditions.

Every lab study participant but one (98.1%) had been programming in general for more than two years; 51.9% (28) had been specifically developing Android apps for more than two years. About half of the participants (53.7%, 29) had developed between two and five Android apps, and 18.5% (10) had developed 10 or more apps. Most participants (85.2%, 46) were not developing Android apps as their primary job, but eight participants were employed as Android app programmers. Using the Kruskal-Wallis test, we did not find a difference in years of Android experience or in number of apps developed across the randomly assigned conditions ($X^2 = 5.06, 4.46$ and $p = 0.409, 0.485$ respectively). As shown in Figure 3.3, our lab-study participants had roughly similar experience to the developers in our online survey.

We also asked how participants learned to program (multiple answers allowed). Almost all (83.3%, 45/54) said they were at least partially self-taught, and 79.6% (43) had formally studied programming at the undergraduate or graduate level. More than half (63.0%, 34) had taken at least one security class, and slightly fewer than half (46.3%, 25) had taken a class in Android programming. Overall, our lab study participants had notably more education than the developers in our online survey.

3.5.2 Functional Correctness Results

Our results demonstrate that the assigned resource condition had a notable impact on participants' ability to complete the tasks functionally correctly; SO and book participants performed best, and official participants performed worst. SO participants solved 67.3% (35/52) of tasks correctly, compared to 66.1% (37/56) for book, 51.8% (29/56) for free, and 40.4% (21) for official. Figure 3.4 (top) provides more detail on the breakdown of correctness across tasks and conditions. The CLMM model (see Table 3.3) indicates that when controlling for task type, professional status, and multiple tasks per participant, participants in official were significantly less likely than baseline SO participants to functionally complete tasks.

Participants' perceptions of the tasks only partially dovetailed with these results. We asked participants, on a 5-point Likert scale, whether they were confident they

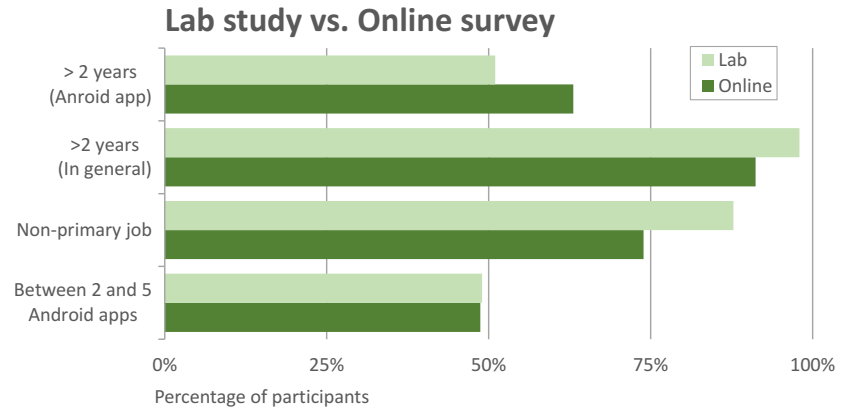

FIGURE 3.3: Comparison of programming experience for participants in our online survey and lab study.

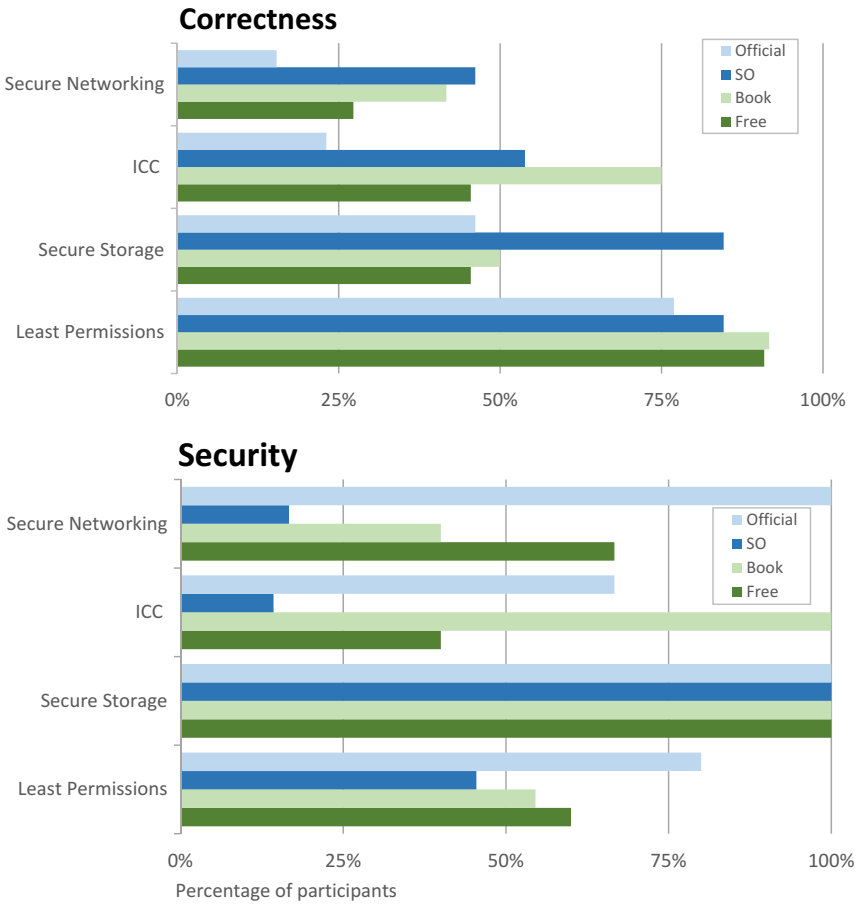

FIGURE 3.4: Top: Percentage of participants who produced functionally correct solutions, by task and condition. Bottom: Percentage of participants whose functionally correct solutions were scored as secure, by task and condition.

had gotten the right answer for each task.[6] Participants in condition free were most

[6]One book participant's confidence answer for the least permissions task was inadvertently not recorded; we exclude that participant from confidence analyses only.

confident: They agreed or strongly agreed they were confident for 55.4% of tasks. Participants in each of the other three conditions were confident for slightly fewer than half of tasks: 47.3% in book and 46.2% in both SO and official. Figure 3.5 illustrates these results.

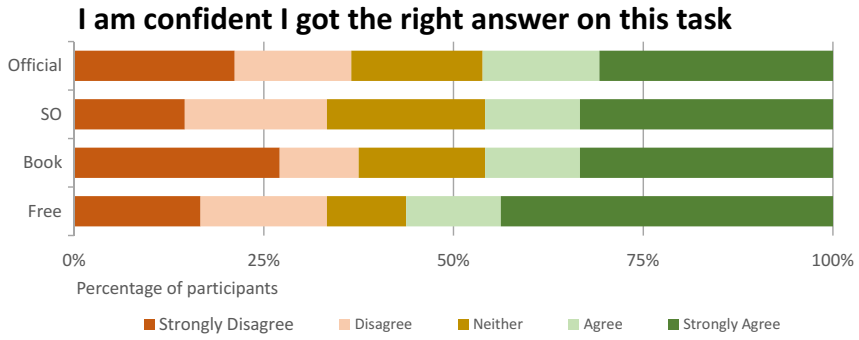

FIGURE 3.5: Participants' confidence in their tasks' correctness, by condition, on a 1-5 Likert scale (1 = Strongly disagree, 5 = Strongly agree).

Using Cohen's κ, we examined whether participants' self-reported confidence in their tasks' correctness (binned as strongly agree/agree and strongly disagree/disagree/neutral) matched with our functional correctness result. We found $\kappa = 0.55$, indicating that participants were assessing their functional correctness only somewhat effectively.

Correctness per Task. Observed correctness varied strongly among the four tasks, as shown in Figure 3.4 (top). In the least permissions task, 87.0% (47) of participants produced a functional solution; in the secure networking task only 33.3% (18) did. These results were mirrored in self-reported confidence: 81.1% of participants were confident about the least permissions task, compared to 53.7% for secure storage, 40.7% for ICC, and only 20.1% for secure networking. The CLMM analysis (Table 3.3) indicates that both the secure storage and least permissions tasks were significantly more likely to be functionally correct than the baseline secure networking task.

Factor	Coef.	Exp(coef)	SE	p-value
free	-1.054	0.349	0.613	0.085
official	-1.535	0.215	0.634	0.015*
book	-0.142	0.868	0.602	0.814
ICC	0.795	2.215	0.455	0.081
secure storage	1.280	3.597	0.468	0.006*
least permissions	3.299	27.092	0.632	< 0.001*
professional	1.004	2.728	0.501	0.045*

TABLE 3.3: CLMM regression table for functional correctness. The non-interaction model including professional status was selected. Non-significant values are greyed out; significant values are indicated with an asterisk. The baseline for condition is SO, and the baseline for task is secure networking.

3.5.3 Security Results

Our results suggest that choice of resources has the opposite effect on security than it did on functionality: SO participants performed worst on this metric. As described in Section 3.4.5, we scored tasks that had been solved correctly for security, privacy, and adherence to least-privilege principles. In the SO condition, only 51.4% (18/35) of functional solutions were graded as secure, compared to 65.5% (19/29) for free, 73.0% (27/37) for book, and 85.7% (18/21) for official. Figure 3.4 (bottom) illustrates these results. Using a CLMM that includes only tasks that were graded as functionally correct (Table 3.4), we find that both official and book produce significantly more secure results than SO. The difference between SO and free, in which many participants elected to use Stack Overflow for most of their tasks (see Section 3.5.4), was not significant.

Factor	Coef.	Exp(coef)	SE	p-value
free	1.112	3.040	0.623	0.074
official	2.218	9.184	0.796	0.005*
book	1.539	4.660	0.604	0.011*
ICC	0.763	2.144	0.666	0.252
least permissions	0.856	2.353	0.609	0.160

TABLE 3.4: CLMM regression table for security. Only tasks graded as functionally correct are included in the model. The non-interaction model without professional status was selected. Non-significant values are greyed out; significant values are indicated with an asterisk. The baseline for condition is SO, and the baseline for task is secure networking.

Security per Task. As with correctness, security results differed noticeably among tasks. For example, every participant who produced a functional solution to the storage task (31) produced a secure solution. On the other hand, only 38.9% (7/18) of participants who produced a functional solution to the networking task were scored as secure. This discrepancy is illustrated in Figure 3.4 (bottom). Our CLMM results (Table 3.4) indicate that neither the ICC nor least permissions task was significantly different from the networking task. Because all functional solutions to the storage task were graded as secure regardless of condition, the regression coefficient for that task approaches infinity, and the results of the model estimates for that task are not interpretable. We omit it from the table.

Considering Security while Programming. We were also interested in the extent to which participants thought about security while solving each task. We measured this in two ways. First, we considered the participants' think-aloud comments for each task, classifying them as having either explicitly mentioned security; mentioned security but later decided to ignore it for the task at hand; or not mentioned security at all. These classifications were independently coded by two coders who then reached agreement, as discussed in Section 3.4.5. We refer to this as *observed* security thinking. Second, we asked participants during the exit interview to self-report for each task whether or not they had considered security, as a yes/no question. We refer to this metric as *self-reported* security thinking. For both metrics, we considered all tasks, not just those that participants completed correctly.

In the observed metric, most participants did not mention security at all (79.2% of all tasks, 171). In the storage task, 16 participants (29.6%) mentioned security and all stuck with it; in the networking task 20 mentioned security (37.0%) but nine later abandoned it. In contrast, only five and four participants ever mentioned security or privacy in the least permissions and ICC tasks, respectively. Common reasons for abandoning security included that finding a secure solution proved too difficult, that the task was for a study rather than real, and that the participant was running short of time.

In the self-reported metric, more participants reported considering security: 60.2% of all tasks (130). Using this metric, security was most frequently considered for secure networking (79.6%), followed by ICC (70.4%) and secure storage (68.5%). Only 22.2% of participants reported considering security for the least permissions task. The higher rate of security thinking using this metric is most likely attributable to the participants being prompted.

To compare conditions, we assign each participant a separate score for each metric, corresponding to the number of tasks in which the participant considered security. In both metrics the average scores were highest in book (0.93, 2.86) and lowest in SO (0.69, 1.92), but neither difference was significant (Kruskal-Wallis, observed: $X^2 = 0.507$, $p = 0.917$, self-report: $X^2 = 4.728$, $p = 0.192$).

Comparing Professionals and Non-Professionals. Although the relatively small sample of professionals we were able to recruit makes comprehensive comparisons difficult, we examined differences in correctness and security between these two groups. For purpose of this analysis, we categorize 14 participants as professionals, including those who are currently or recently had been professional developers. The non-professional participants are primarily university students. The professionals were randomly distributed across conditions: five in free, three in SO, two in official and four in book.

Overall, professionals were slightly more likely to produce a functional solution, with a median three functionally correct tasks (mean = 2.79, sd = 0.70) compared to two functionally correct tasks (mean = 2.08, sd = 1.23) for non-professionals. We observed essentially no difference in security results: professionals' solutions were median 66.7% secure (mean = 69.0%, sd = 0.20), compared to 66.7% for non-professionals (mean = 66.2%, sd = 0.36). These observations fit with the CLMM results: professional status predicts a small but significant increase in functional correctness, but professional status is excluded from the best-fitting security model.

3.5.4 Use of Resources

During the tasks, we collected the search terms used and pages visited by all participants in non-book conditions. In addition, during the exit interview, we asked participants several questions about the resources they were assigned to use. In this section, we analyze participants' search and lookup behavior as well as their self-reported opinions.

Lookup Behavior Across Conditions.

We define "search queries" as submitting a search string to a search engine or to the search boxes provided by Stack Overflow and the official Android documentation. Participants in the SO condition made on average 22.8 queries across the four tasks, compared to 14.5 for the official condition. Participants in free were closer to SO than official, with an average of 21.1 queries. We also observed that participants in the official, free and SO conditions visited on average 35.4, 36.1, and 53.2

unique web pages across the four tasks. We offer two hypotheses for these results, based on on our qualitative observations: First, official participants were more likely to scroll through a table of contents or index and click topics that seemed relevant (as opposed to doing a keyword search) than those in other conditions, presumably because the official documentation is more structured. Second and perhaps more importantly, SO participants seemed to be more likely to visit pages that turned out to be unhelpful and restart their searches.

Participants in the free condition were given their choice of Internet resources to help them solve the programming tasks. Every free participant started every attempt to get help with a Google search. Undoubtedly this was partially influenced by Chrome using Google as the default start page as well as automatically using Google search for terms entered in the URL bar, but the complete unanimity (along with results from the online survey) suggests that many or most attempts would have started there anyway. From within their Google results, every participant selected at least one page within the official Android API documentation, and all but one visited Stack Overflow as well. A few visited blogs, and one visited an online book. These results are consistent with the online survey results reported in Section 3.3. In terms of frequency, official documentation was most popular, representing between 50 and 85% of non-google-search pages for all participants except one outlier who visited it 98% of the time. Most participants visited Stack Overflow for between 10 and 40% of their pages, with outliers at 0 and 2.4% as well as 50%. While participants in the group visited more official documentation pages than pages at Stack Overflow, their functionality and security results more closely resemble the group than the official group. This may be partially explained by a behavior pattern that we observed several times in the free condition: participants spent some time reading through the official documentation, but as the time limit approached used content (often a copied and pasted code snippet) from Stack Overflow.

Search Query Selection. We also examined the search query text chosen by participants. Queries were normalized for capitalization and spacing, and any queries within one string edit of each other were consolidated (to account for plurals and typos). Because few participants exactly duplicated one another's queries, in order to discern trends, one researcher manually coded similar terms into categories. For example, "restrict access developers," "restrict app access for same developer," and "restrict apps same developer" were categorized together. For the secure networking task, the most common queries involved hostname exceptions and HTTPS, together with just a few searches for certificates, certificate errors, and hostname verifiers. For the ICC task, the most popular searches included manifest, permissions, services, external access, and restricting access. A few more knowledgeable participants searched for intent filters, user IDs, and signatures. For secure storage, the most popular choices included storage, persistent storage, and shared preferences; for least permissions participants most frequently searched for call and phone call, with a few searching for dial. Only four participants searched for "secure" or "security," including two in free and one each in SO and official.

Participants' Opinions about Information Sources. We asked our non-free participants whether they had previously used their assigned resource. All 14 SO participants had previously used Stack Overflow, and most (10/13) official participants had used the official documentation. However, only six of 14 book participants had used books. We also asked participants to rate, on a five-point Likert scale, the extent to which the resources they used were easy to use, helpful, and correct. Results are

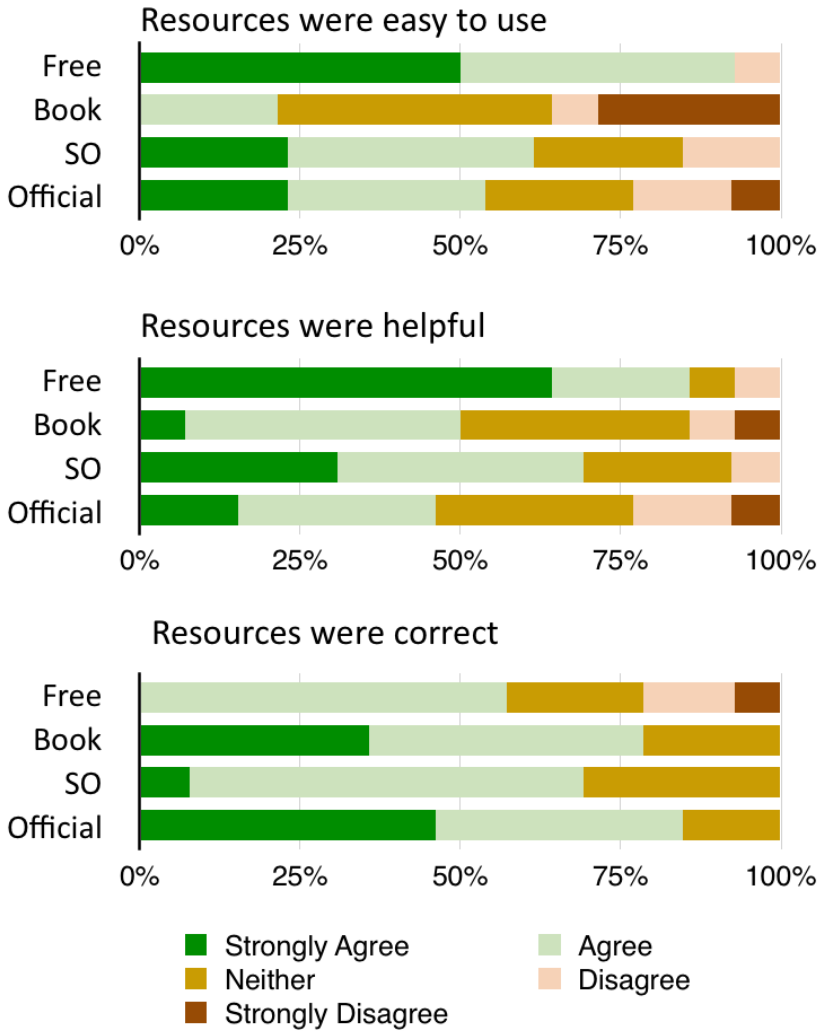

FIGURE 3.6: Participants' agreement (on a five-point Likert scale) with the statements that the resources they used were easy to use, helpful, and correct, by condition.

shown in Figure 3.6. As might be expected, participants found free choice easiest to use and books least easy; in contrast, they were most likely to consider books and the official documentation to be correct.

We also asked about the effect of participants' assigned resource on their performance. In every non-free condition, the large majority (official: 92.3% (12/13); book: 92.9% (13/14); SO: 78.6% (11/14)) said they would have performed better on the tasks if they had been allowed to use different resources. In particular, official and book participants said they would have liked to access Stack Overflow or search engines such as Google, so that they could search for their specific problems rather than reading background information. One book user mentioned the "danger that books could be outdated." On the other hand, many SO participants said they would have liked to access the official documentation to read up on background information for their problems.

Time constraints were also a concern for our participants. Most (61.1%, 33) said they would probably have performed better had they been given more time, while nine (16.7%) mentioned (unprompted) that more time would have allowed them to make their solutions more secure. One participant in official, for example, said

that "Twenty minutes is very limited to consider security." The remaining 38.9% said more time would not have helped, either because they solved the tasks to their satisfaction, or because they believed the resource they were using did not allow them to find a (better) solution.

3.6 Quality of Stack Overflow Responses

To better contextualize the performance of the participants – in both, the SO and free condition –, we examined in detail all Stack Overflow pages (threads) visited by our participants during the programming tasks. In particular, we were curious about whether these pages contained secure and/or insecure examples and code snippets, and whether the security implications were explained. As might be expected, we found many discouraging instances of insecure examples and few discussions of security implications.

3.6.1 Classification Methodology

We rated each thread on five different attributes, described below. All threads were independently coded by two researchers, who then reached consensus on any conflicts.

Task Relevance. We first checked whether the topic of the thread was actually relevant to solving the study task. If it would not help the participant in solving the task, it was flagged as off-topic and not looked at further.

Usefulness. We rated each on-topic thread as useful or not useful, based on how related answers were to the question. Threads with no answers, or no answers that responded to the original question, were rated as not useful. Threads with answers that discussed the question and gave helpful comments, links to other resources, or sample code were rated as useful.

Code Snippets. We examined all answers in each thread for ready-to-use code snippets. We rated a code snippet as ready-to-use if it was syntactically correct and a developer could copy and paste it into an app. Each thread in which at least one answer qualified was marked as containing a code snippet. Each code snippet was individually rated as secure or insecure relative to the programming tasks described in Section 3.4.3.

External Links. Within each thread, we looked for answers containing external links. We classified threads as containing links to GitHub, to other code repositories, to other Stack Overflow threads, or to anywhere else. Additionally, we classified the linked content as either secure or insecure.

Security Implications. We investigated whether any answer in the thread discussed security implications of possible solutions. For example, if two solutions existed and one included an extra permission request, we checked whether any of the answers discussed a violation of the least-privilege principle. If an answer contained a NullHostnameVerifier, we would check if at least one of the answers would advise that verification should not be disabled.

3.6.2 Classification Results

Overall, our participants accessed 139 threads on Stack Overflow. We categorized 41 threads as being on-topic. Table 3.5 summarizes the classification results for these 41 threads. Of these, 20 threads contained code snippets. Half of the threads containing code snippets contained only insecure snippets, such as instructions to use NullHostnameVerifiers and NullTrustManagers, which will accept all certificates regardless of validity. Among these 10 threads containing only insecure code snippets, only three described the security implications of using them. This creates the possibility for developers to simply copy and paste a "functional" solution that voids existing security measures, without realizing the consequences of their actions. More encouragingly, seven of the 10 remaining threads with code snippets contained only secure code snippets.

We next investigated how threads with different properties compared in terms of popularity (measured by total upvotes for the thread). Unsurprisingly, we found that threads with code snippets were more popular than those without ($W = 319.5$, $p = 0.00217$, $\alpha = 0.025$, Wilcoxon-Signed-Rank Test (B-H)). Discouragingly, we found no statistical difference between threads with secure and insecure code snippets ($W = 73$, $p = 0.188$). On the other hand, threads that discuss security implications are slightly more popular than those that don't ($W = 239.5$, $p = 0.0308$, $\alpha = 0.05$ (B-H)).

Although these results cover only a very small sample of Stack Overflow threads, they provide some insight into why our SO participants had lower security scores than those in the official condition.

Answers in the thread include ...	**Count**	
Useful answers	35	(85.4%)
Useless answers	6	(14.6%)
Discussion of security implications	12	(29.3%)
Working code examples	20	(48.8%)
Only secure code examples	7	(17.0%)
Only insecure code examples	10	(24.4%)
...but also discussion of security implications	3	(30.0%)
Secure links	23	(56.1%)
Insecure links	6	(14.6%)
Links to GitHub	4	(9.8%)
Links to other code repositories	1	(2.4%)
Links to other Stack Overflow threads	4	(9.8%)
Only secure code examples and secure links	3	(7.3%)

TABLE 3.5: Properties of the 41 on-topic Stack Overflow threads accessed during the lab study.

3.7 Programming Task Validity

To provide evidence for the validity of our lab study tasks and results, we examined how the APIs used in our programming tasks (cf. Table 3.1) are used in real-world apps. In particular, we were interested in how frequently these APIs are used in real Google Play apps, as well as whether secure or insecure solutions are more prevalent. Results of our analysis show that the APIs we examined are widely used; in

Stack Overflow Threads			
with code snippets		**without code snippets**	
mean	97.7	mean	3.9
median	12	median	2.5
sd	163.9	sd	4.4
W = 319.5, p = 0.00217, α = 0.025 (B-H)			
with secure code snippets		**with insecure code snippets**	
mean	204.3	mean	70.2
median	145	median	14
sd	209.3	sd	122.4
W = 73, p = 0.188			
with security implications		**without security implications**	
mean	135.2	mean	17.4
median	16	median	3
sd	207	sd	37
W = 239.5, p = 0.0308, α = 0.05 (B-H)			

TABLE 3.6: Popularity ratings for threads containing code snippets.

line with our lab study results, the secure networking and ICC APIs were frequently used in ways that suggest security concerns.

3.7.1 Analysis

To analyze real-world apps, we applied standard static code-analysis techniques: We decompiled Android APK files, constructed control flow graphs (CFGs), and applied backtracking to gather insights about how often real-world developers use APIs relevant to our programming tasks. Limitations of this approach are discussed in Section 3.8. Overall, we analyzed a random sample of 200,000 free Android apps from Google Play.

Secure Networking Task. For this task, we analyzed whether an app implements the `HostnameVerifier` interface (cf. Table 3.1). Hostname verification requires a developer to implement the `verify(String hostname, SSLSession session)` method. We checked if an implementation actually performs hostname verification by processing the `hostname` parameter or if it simply accepts every hostname (i. e. `return true;`).

ICC Task. For this task, we analyzed an app's Manifest file (cf. Table 3.1). We extracted `<service>` entries from the XML DOM, then checked for `<intent-filter>` child nodes to determine whether an intent filter was set. We also checked whether the `android:exported` flag, which indicates whether a service is made publicly available, was present and if it was set to `true`. Lastly, we extracted `android:permission` attributes to see if services were protected by permissions. We also extracted the `android:protectionLevel` attributes to learn whether signature or system permissions are required to use this service.

Secure Storage Task. To determine whether an app stores data persistently, we looked up relevant API calls in the call graph. We distinguished between three different targets: SQLite databases, the file system, and shared preferences (cf. Table 3.1).

To check for SQLite database usage, we looked up the `openOrCreateDatabase` API call in the CFG. Developers can use this API call in a way that keeps data local

to an app by explicitly setting the MODE_PRIVATE flag or using the default. Setting the MODE_WORLD_WRITEABLE or MODE_WORLD_READABLE flag stores the database outside an app's local storage and makes it available to other apps. We used backtracking to check which flags were set.

To analyze file-system access, we looked up API calls that return output- or inputstreams to a file handle. This includes the openFileOutput method and the mode flags. Additionally, we checked for use of methods that find the path of the external storage as well as the WRITE_EXTERNAL_STORAGE permission.

To check for shared preferences usage, we looked up the getSharedPreferences, getPreferences and getDefaultSharedPreferences API calls in the CFG. The MODE_PRIVATE, MODE_WORLD_WRITEABLE and MODE_WORLD_READABLE flags are used to distinguish between secure and insecure solutions.

Least Permissions Task. To examine use of dialing compared to calling, we analyzed the Manifest file for the occurrence of the CALL_PHONE permission request and searched for relevant API calls in the CFG. To initiate a phone call, a new Intent object must be created using a string parameter to specify the intended action. We used backtracking to obtain the respective action value and searched for ACTION_DIAL and ACTION_CALL values.

Apps that used an ACTION_DIAL intent were rated as adhering to least privilege since they use the system's dialer and do not require an additional permission. Apps that use an ACTION_CALL intent in combination with requesting the CALL_PHONE permission were rated as not adhering to least privilege.

3.7.2 Results

Table 3.7 summarizes the results of our real-world app analysis, which are further detailed below.

Secure Networking Task. We identified 19,734 apps that implement their own hostname verifier. Of those apps, 19,520 apps (98.9%) implement it in a way that accepts any hostname, i.e. they effectively turn off hostname verification and make their apps vulnerable to active Man-In-The-Middle attacks. Only the remaining 214 apps (0.1%) implement alternative hostname verification strategies. Although the limitations of static code analysis prevent us from assessing whether these implementations meet the programmers' expectations, we score them as secure compared to hostname verifiers that simply accept every hostname.

ICC Task. 42,193 apps implemented their own services. Of those, 15,857 (37.6%) configured a non-default access policy for their services by setting respective properties in the Manifest file. 11,929 (75.2%) of those apps use intent filters or set the exported=true flag, which weakens security. 3,928 (24.8%) of those apps configured their services to that an entity must have a permission in order to launch the service or bind to it. Only 101 apps required an entity to have a permission of the same developer or a system permission.

Secure Storage Task. 155,017 apps implemented file-system access. Of those, 34,183 (22.1%) access files on external storage or write to the internal storage with MODE_WORLD_READABLE/WRITEABLE. However, 120,834 (77.9%) only access files on internal storage with MODE_PRIVATE. Similar numbers can be seen with shared preferences, where 130,408 (88.0%) apps out of 148,256 use MODE_PRIVATE and 17,848 (12.0%) use a publicly accessible mode. SQLite databases are not very common

	secure	apps
Secure Networking Task		
broken hostname verifier	○	19,520
alternative hostname verification	●	214
ICC Task		
service	-	42,193
intent filter	○	8,133
exported=true	○	3,796
permission	◐	3,827
permission, signature	●	86
permission, signature or system	●	15
Secure Storage Task		
filesystem, private	●	120,834
filesystem, public	○	34,183
database, private	●	4,471
database, public	○	154
shared preferences, private	●	130,408
shared preferences, public	○	17,848
Least Permissions Task		
dial, permission	○	3,907
dial, no permission	●	48,832
call, permission	○	5,336
call, no permission	◐	6,157

● = secure; ○ = insecure

TABLE 3.7: Results of statically analysing a random sample of 200,000 Android apps.

among our dataset, but 4471 out of 4625 (96.7%) also use a private mode and only 154 (3.3%) a public mode.

Least Permissions Task. Overall we identified 64,232 apps that use intents to make phone calls. Of those apps 52,739 (82.1%) use the `ACTION_DIAL` action for that purpose. Interestingly 3,907 (7.4%) of those apps request the `CALL_PHONE` permission although `ACTION_DIAL` does not require a dedicated permission. The remaining 11,493 (17.9%) apps use the `ACTION_CALL` action which requires the `CALL_PHONE` permission to be requested by the developer. Of those apps, 6,157 (53.6%) do not request the `CALL_PHONE` permission and hence might crash if the `ACTION_CALL` intent is called.

3.7.3 Discussion

We found that 187,291 (93.6%) of the randomly chosen 200,000 apps we analyzed in our study used at least one of the APIs we used in our programming tasks, suggesting that our laboratory study includes programming tasks that real-world developers encounter. Interestingly, for the secure storage and least privilege tasks, most apps implement the more secure solutions. In contrast, for the secure networking and ICC tasks, we found more insecure solutions. This mirrors the results of our lab study (cf. Section 3.5). This analysis provides additional concrete evidence for the relevance and the results of our lab study.

3.8 Limitations

As with most studies of this type, our work has several limitations.

First, the response rate for our online developer survey was very low, as might be expected from sending unsolicited emails to prospective participants. This may introduce some self-selection bias, but we have no reason to believe a priori that those who responded differ meaningfully in terms of security knowledge or resource usage from those who did not.

Our lab study created an artificial scenario—working within a tight time limit, with unfamiliar starter code—which may have impacted participants' ability to complete tasks correctly and securely. Similarly, the artificial nature of study participation may have reduced participants' incentives to consider security. In addition, a majority of our lab participants were students rather than professional developers, and overall the lab participants were more formally educated than the developers in our online survey, which may limit the generalizability of our results somewhat. The professionals in the study performed slightly but not significantly better than the non-professionals in functional correctness, but not in security. All of these issues, however, were present across conditions, suggesting that comparisons among conditions are valid. We also hoped that the time limit would partially emulate the pressure professional developers feel to bring apps to market quickly rather than focus on writing the best possible software.

Our analysis of Stack Overflow threads is limited to only those accessed by our lab study participants; threads on other topics may exhibit different properties. In addition, our manual coding process was somewhat subjective. Nonetheless, we believe this analysis provides a useful glimpse into the broader characteristics of Stack Overflow as a resource.

The static code analysis we conducted has several limitations. Although we performed reachability analyses for all API calls, an inherent limitation of static code analysis is that we still might have included code paths that are not executed. For

the ICC task, it is possible that some services we marked as insecure were made publicly available deliberately rather than by mistake; however, the official Android documentation[7] discourages the use of intent filters for security reasons. Hence, while we may have some false positives, our results do suggest at minimum a violation of best practices. A similar limitation applies to the storage task: while some uses of external storage are necessary or deliberate, this also represents a risky violation of best practices[8] that can lead to unexpected disclosures of personal information [231].

3.9 Discussion

In the past, anecdotal evidence has suggested that the resources Android developers use when programming directly affect the security and privacy properties of the apps they make. In this paper, we present the first systematic investigation of this theory by approaching the problem of how programming resources affect Android developers' security- and privacy-relevant decisions from several different angles. We conducted a 295-person online survey about the resources developers use, both in general and specifically for security-relevant problems. Based on results from this survey, we then conducted a 54-person lab study directly exploring the impact of resource choice on both functional correctness and security. To provide context for these studies, we manually analyzed the security characteristics of the Stack Overflow posts our participants accessed and automatically analyzed how the APIs we tested in the lab are used in 200,000 randomly sampled apps from the Google Play market.

When combined, results from these varied analyses suggest several interesting conclusions:

- Real-world Android developers use Stack Overflow (and other Q&A communities) as a major resource for solving programming problems, including security- and privacy-relevant problems.
- Other resources, such as official Android API documentation, do not provide the same degree of quickly understandable, directly applicable assistance. Our results suggest that using Stack Overflow helps Android developers to arrive at functional solutions more quickly than with other resources.
- Participants who were given free choice of resources tended to visit both the official documentation and Stack Overflow, but their performance in both functional correctness and security was more similar to participants in the Stack Overflow condition.
- Because Stack Overflow contains many insecure answers, Android developers who rely on this resource are likely to create less secure code. Access to quick solutions via a Q&A community may also inhibit developers' security thinking or reduce their focus on security.
- Code relevant to the tasks we explored can be found in 93.6% of the apps we sampled. Many of these apps exhibit similar security patterns to those observed in our lab study.

[7] cf. `http://developer.android.com/guide/components/intents-filters.html`
[8] cf. `http://developer.android.com/guide/topics/data/data-storage.html`

Few participants in our study explicitly mentioned security or used it as a search term when accessing resources. While this may be partially a function of our artificial environment, when combined with prior research and anecdotal evidence, this suggests that security remains at best a secondary concern for many real-world developers [23, 85]. This underscores the need for both APIs and informational resources that promote security even when developers are not thinking about it directly.

Android developers are unlikely to give up using resources that help them quickly address their immediate problems. Therefore, it is critical to develop documentation and resources that combine the usefulness of forums like Stack Overflow with the security awareness of books or official API documents. One approach might involve rewriting API documents to be more usable, e.g. by adding secure and functional code examples. Another might be to develop a separate programming-answers site in which experts address popular questions, perhaps initially drawn from other forums, in a security-sensitive manner. Alternatively, Stack Overflow could add a mechanism for explicitly rating the security of provided answers and weighting those rated secure more heavily in search results and thread ordering. Further research is needed to develop and evaluate solutions to help prevent inexperienced or overwhelmed mobile developers from making critical mistakes that put their users at risk.

This chapter made clear that usable secure resources are an important factor for code security. Conducting the programming study in the lab limited us to developer populations that were locally available, which means that we recruited mostly students. For our next study, we aimed to investigate whether cryptographic APIs that are specifically developed to be usable help developers choose secure programming solutions. We moved our experiment online in order to access a more diverse population with more professional developers.

Chapter 4

Comparing the Usability of Cryptographic APIs

Disclaimer: *The contents of this chapter were previously published as part of the conference paper "Comparing the Usability of Cryptographic APIs", presented at the 2017 IEEE Symposium on Security and Privacy. This research was conducted as a team with my co-authors Christian Stransky, Doowon Kim, Michelle L. Mazurek, Simson Garfinkel, Sascha Fahl, and Michael Backes; this chapter therefore uses the academic "we". Christian Stransky, Doowon Kim and Sascha Fahl conducted the preliminary survey of libraries; based on critera all authors discussed, we chose the libraries. Christian Stransky, Michelle L. Mazurek, Sascha Fahl and I designed the study. Simson L. Garfinkel and I conducted the literature review. Michelle L. Mazurek and I designed the questionnaire. Christian Stransky and Sascha Fahl conducted the experiment. Christian Stransky, Doowon Kim, Michelle L. Mazurek, Sascha Fahl and I evaluated the experiment. Christian Stransky, Doowon Kim, Michelle L. Mazurek, Sascha Fahl and I co-wrote the paper. Simson L. Garfinkel, Michelle L. Mazurek and Sascha Fahl supervised throughout. The libraries and version of Python used throughout the study reflect the landscape in 2016.*

4.1 Motivation

Today's connected digital economy and culture run on a foundation of cryptography, which both authenticates remote parties to each other and secures private communications. Cryptographic errors can jeopardize people's finances, publicize their private information, and even put political activists at risk [12]. Despite this critical importance, cryptographic errors have been well documented for decades, in both production applications and widely used developer libraries [13, 70, 99, 205].

Many researchers have used static and dynamic analysis techniques to identify and investigate cryptographic errors in source code or binaries [13, 70, 83, 99, 205]. This approach is extremely valuable for illustrating the pervasiveness of cryptographic errors, and for identifying the kinds of errors seen most frequently in practice, but it cannot reveal root causes. Conventional wisdom in the security community suggests these errors proliferate in large part because cryptography is so difficult for non-experts to get right. In particular, libraries and Application Programming Interfaces (APIs) are widely seen as being complex, with many confusing options and poorly chosen defaults (e.g. [257]). Recently, cryptographers have created new libraries with the goal of addressing developer usability by simplifying the API and establishing secure defaults [32, 64]. To our knowledge, however, none of these libraries have been empirically evaluated for usability. To this end, we conduct a controlled experiment with real developers to investigate root causes and compare different cryptographic APIs. While it may seem obvious that simpler is

better, a more in-depth evaluation can be used to reveal where these libraries succeed at their objectives and where they fall short. Further, by understanding root causes of success and failure, we can develop a blueprint for future libraries.

This chapter presents the first empirical comparison of several cryptographic libraries. Using Python as common implementation language, we conducted a 256-person, between-subjects online study comparing five Python cryptographic libraries chosen to represent a range of popularity and usability: cryptography.io, Keyczar, PyNaCl, M2Crypto and PyCrypto. Open-source Python developers completed a short set of cryptographic programming tasks, using either symmetric or asymmetric primitives, and using one of the five libraries. We evaluate participants' code for functional correctness and security, and also collect their self-reported sentiment toward the usability of the library. Taken together, the resulting data allows us to compare the libraries for usability, broadly defined to include ability to create working code, effective security in practice (when used by primarily non-security-expert developers), and participant satisfaction. By using a controlled, random-assignment experiment, we can compare the libraries directly and identify root causes of errors, without confounds related to the many reasons particular developers may choose particular libraries for their real projects.

We find that simplicity of individual mechanisms in an API does not assure that the API is, in fact, usable. Instead, the stronger predictors of participants producing working code were the quality of documentation, and in particular whether examples of working code were available on the Internet, within or outside the provided documentation. Surprisingly, we also found that the participant's Python experience level, security background, and experience with their assigned library did not significantly predict the functionality of the code that they created. None of the libraries were rated as objectively highly usable, but PyCrypto, a complex API with relatively strong documentation, was rated significantly more usable than Keyczar, a simple API with poor documentation.

On the other hand, with some important caveats, simplified APIs did seem to promote better security results. As might be expected, reducing the number of choices developers must make (for example, key size or encryption mode of operation) also reduces their opportunity to choose incorrect parameters. Python experience level was not significantly correlated with security results, but participants with a security background were more likely to produce code that was, in fact, secure. Nevertheless, the overall security results were somewhat disappointing. A notable source of problems was APIs that did not easily support important auxiliary tasks, such as secure key storage. Perhaps of most concern, 20% of functional solutions were rated secure by the participant who developed them but insecure according to our evaluation; this suggests an important failure to communicate important security ideas or warn about insecure decisions.

4.2 Related Work

We discuss related work in four key areas: measuring cryptography problems in deployed code; investigating how developers interact with cryptographic APIs; attempts at developing more usable cryptographic libraries and related tools; and approaches to evaluating API usability more generally.

Cryptography problems in real code. Researchers have identified misuses of cryptography in deployed code. Egele et al. examined more than 11,000 deployed Android apps that use cryptography and found that nearly 90% contained at least

one of six common cryptography errors [70]. Fahl et al. and Onwuzurike et al. also analyzed Android apps, and found that a large number did not correctly implement the Trusted Layer Security (TLS) protocol, potentially leading to security vulnerabilities to Man-In-The-Middle (MITM) attacks [2, 82, 84, 175, 178, 184]. Likewise, a study examining Apple's iOS apps revealed that many were vulnerable to MITM attacks because of incorrect certificate validation during TLS connection establishment [85]. Other researchers specifically examined mobile banking applications and found a plethora of potentially exploitable cryptographic errors [205]. Lazar et al. examined cryptography-related vulnerabilities from the Common Vulnerabilities and Exposures (CVE) database and found more than 80% resulted from errors at the application level [143]. In all of these cases, weak ciphers and insufficient randomness were common problems; in this paper, we test the hypothesis that these problems are strongly affected by API design. Georgiev et al. identified many certificate-validation errors in applications and libraries; the authors attribute many of these vulnerabilities to poorly designed APIs and libraries with too many confusing options [99].

Interacting with cryptographic APIs. Others have investigated how developers interact with cryptographic APIs. Nadi et al. manually examined the top 100 Java cryptography posts on Stack Overflow and found that a majority of problems were related to API complexity rather than a lack of domain knowledge [166]. Follow-up surveys of some Stack Overflow users who had asked questions and of Java developers more generally confirmed that API complexity and poor documentation are common barriers in cryptographic API use. In this paper, we compare different APIs to measure their relative difficulty of use. Relatedly, Acar et al. examined how use of different documentation resources affects developers' security decisions, including decisions about certificate validation [4]; we compare different APIs rather than different sources of help. **Making cryptography more usable.** Several cryptographic APIs have been designed with usability in mind. The designers of NaCl (Networking and Cryptographic library, pronounced "salt") describe how their design decisions are intended to promote usability, in large part by reducing the number of decisions a developer must make, but do not empirically evaluate its usability [32]. In this work, we empirically compare NaCl to more traditional APIs, as well as to non-academic libraries that also claim usability benefits (e.g., cryptography.io [64]).

Rather than a new API, Arzt et al. present an Eclipse plugin that produces correct code templates based on high-level requirements identified by the developer [15]. This approach can make working with existing APIs easier; however, it is orthogonal to the question of how APIs do or do not encourage secure practices. Indela et al. suggest using design patterns to describe high-level *semantic APIs* for goals that require cryptography, such as establishing a secure connection or storing data securely [125]. This approach is complementary to improving cryptographic libraries that underlie such patterns.

Evaluating APIs, security and otherwise. Many software engineering researchers have examined what makes an API usable. Myers and Stylos provide a broad overview of how to evaluate API usability, with reference to Nielsen's general usability guidelines as well as the Cognitive Dimensions framework [57, 165, 173]. Henning and Bloch separately provide sets of maxims for improving API design [34, 120]. Smith and Green proposed similar high-level guidelines specific to security APIs [115]. We adapt guidelines from these various sources to evaluate the APIs we examine.

Concurrent with our work, Gorski and Iacono [111] use an extensive literature review to formulate high-level technical and usability criteria along which security-relevant APIs should be designed, calling for further work on evaluating adherence

to these principles. Also concurrent to our work, Wijayarathna et al. develop a set of questions about security APIs based on the above guidelines, resulting in a questionnaire similar to the one we developed and used in this work [273].

Oliveira et al. conducted a laboratory study to examine the security mindset of developers. They found that security is not a priority in the standard developer's mindset, but that detailed priming for security issues helps [174]. Wurster and Van Oorschot recommend assuming that developers will not prioritize security unless incentivized or forced to, and suggest mandating security tools, rewarding secure coding practices, and ensuring that secure tools and APIs are more usable and attractive than less secure ones [278]. Our work focuses on how choice of library affects developers who have already decided to interact with a cryptographic API and have been primed for the importance of security to their task.

Finifter, Wagner and Prechelt compared the security of two web applications built to the same specification but with different frameworks. They found that automatic framework-level support for mitigating certain vulnerabilities improved overall security, while manual framework supports were readily forgotten or neglected [94, 191].

Researchers have also conducted empirical studies of API usability in different domains, including comparing APIs for configuration [220], considering how assigning methods to classes affects usability [239], and analyzing the usability of the factory pattern [72]. Piccioni et al. examined the usability of a persistence library using a method similar to the one we use in this work, with exit interview questions structured around the Cognitive Dimensions framework [186]. They successfully identify usability failures of the examined API, and their results emphasize the critical importance of accurate, unambiguous and self-contained documentation to API usability. Burns et al. provide a preliminary survey of work evaluating APIs empirically [44].

4.3 Study Design

We designed an online, between-subjects study to compare how effectively developers could quickly write correct, secure code using different cryptographic libraries. We recruited developers with demonstrated Python experience (on GitHub) for an online study.

Participants were assigned to complete a short set of programming tasks using either symmetric- or asymmetric-key cryptography, using one of five Python cryptographic libraries. Assignment to one of the resulting 10 conditions was initially random, with counterbalancing to ensure roughly equivalent participant counts starting each condition. As the study progressed, however, it became clear that dropout rates varied widely by condition (see Section 4.4.3 for details), so we weighted the random assignment to favor conditions with higher dropout rates.

Within each condition, task order was randomized. Symmetric participants were either given a key generation, then an encryption/decryption task, or vice-versa. Asymmetric participants were assigned a key generation task, an encryption/decryption task, and a certificate validation task, according to a latin square ordering.

After finishing the tasks, participants completed a brief exit survey about the experience. We examined participants' submitted code for functional correctness and security. The study was approved by our institutions' ethics review boards.

4.3.1 Language Selection

We chose to use Python as the programming language for our experiment because it is widely used across many communities and has support for all kinds of security-related APIs, including cryptography. As a bonus, Python is easy to read and write and is widely used among both beginners and experienced programmers. Indeed, Python is the third most popular language on GitHub, trailing JavaScript and Java [103]. Therefore, we reasoned that there would be many Python developers to recruit for our study.

4.3.2 Cryptographic Library Identification

Next, we performed a series of Internet searches to identify possible cryptographic libraries that we could use in our study. We were agnostic to library implementation language, performance, and third-party certification: all that mattered was that the library could be called from Python language bindings. At this point, we decided to use the Python 2.7 programming language because several Python cryptographic libraries did not support Python 3.

We selected five Python libraries to empirically compare based on a combination of their popularity, their suitability for the range of tasks we were interested in, and our desire to compare libraries that were and were not designed with usability in mind. Table 4.1 lists details of these features for the libraries we examined.

We selected three libraries whose documentation claims they were designed for usability and that each handle (most of) the tasks we were interested in: cryptography.io, Keyczar, and PyNaCl. cryptography.io describes itself as “cryptography for humans” [64], Keyczar is “designed to make it easier and safer for developers to use cryptography" [275], and PyNaCl is a Python binding for NaCl, a crypto library designed to avoid “disaster" in part via simplified APIs [32]. pysodium is a potential alternative to PyNaCl; although pysodium is very slightly more popular, it is still beta and has no included documentation, so we selected PyNaCl.

For comparison, we also selected two libraries that do not make usability claims: PyCrypto and M2Crypto. PyCrypto is the most popular general-purpose Python crypto library we found, and the closest thing to a “default" Python crypto library that exists. M2Crypto is a Python binding for the venerable OpenSSL library, which is frequently criticized for its lack of usability. pyOpenSSL is both more popular than M2Crypto and the official OpenSSL [245] binding for Python; however, it lacks support for symmetric and asymmetric encryption, which was a major part of our study, so we opted for M2Crypto instead. We provide further details about the features and documentation of the libraries we selected in Section 4.3.6.

We excluded libraries that include few of the features we were interested in, or that have negligible popularity. We excluded PyCryptodome as a less popular replacement for PyCrypto, gnupg for its limited support for encryption (mainly in the context of email), pycryptopp as it was deprecated as of January 2016, and simplecrypt as it does not support asymmetric cryptography.

In tables and figures throughout the paper, we order the libraries as follows: PyCrypto first as the most popular, then M2Crypto as the other library without usability claims, then the three libraries with usability claims.

		Sym		Asym						
		Key generation	Encryption	Key generation	Encryption	KDF	Digital sig.	X.509	Usability claims	Downloads
PyCrypto	[148]	●	●	●	●	●	●	●	○	25 149 446
cryptography.io	[64]	●	●	●	●	●	●	●	●	10 481 277
M2Crypto	[46]	●	●	●	●	●	●	●	○	2 369 827
Keyczar	[136]	●	●	●	●	○	●	○	●	595 277
PyNaCl	[199]	●	●	●	●	○	●	○	●	46 013
pyOpenSSL	[200]	○	○	○	○	○	●	●	○	10 188 101
tlslite	[185]	○	○	○	○	○	○	●	○	641 488
bcrypt	[31]	○	○	○	○	○	○	○	○	536 851
gnupg	[104]	○	●	●	●	○	●	○	○	189 851
pycryptopp	[196]	●	●	●	○	○	●	○	○	140 703
scrypt	[221]	○	●	○	○	○	○	○	○	140 446
simple-crypt	[225]	●	●	○	○	●	○	○	●	112 254
pysodium	[201]	●	●	●	●	○	●	○	●	49 275
ed25519	[69]	○	○	●	○	○	●	○	○	29 670
pyaes	[193]	○	●	○	○	○	○	○	○	19 091
PyCryptodome	[195]	●	●	●	●	●	●	○	○	16 960
PyMe	[198]	○	○	●	●	○	●	○	○	2 489
pyDes	[197]	○	●	○	○	○	○	○	○	? †
tls	[194]	○	○	○	○	○	○	○	●	? †

● = applies; ○ = does not apply

TABLE 4.1: Cryptography-related Python libraries and their features, ordered by popularity. The top section includes the libraries we tested. Download counts as of May 2016 were taken from the PyPI ranking website (http://pypi-ranking.info). †No download statistics available.

4.3.3 Recruitment and Framing

To maintain ecological validity, we wanted to recruit developers who actively use Python. To find such developers, we conducted a systematic analysis of Python contributors on the popular GitHub collaborative source code management service.

We extracted all Python projects from the GitHub Archive database [102] between GitHub's launch in April 2008 and February 2016, giving us 749 609 projects in total. We randomly sampled 100 000 of these repositories and cloned them. Using this random sample, we extracted email addresses of 50 000 randomly chosen Python committers. These committers served as a source pool for our recruiting.

We emailed these developers in batches, asking them to participate in a study exploring how developers use Python libraries. We did not mention cryptography or security in the recruitment message. We mentioned that we would not be able to compensate them, but the email offered a link to learn more about the study and a link to remove the email address from any further communication about our research. Each contacted developer was assigned a unique pseudonymous identifier (ID) to allow us to correlate their study participation to their GitHub statistics separately from their email address.

Recipients who clicked the link to participate in the study were directed to a landing page containing a consent form. After affirming they were over 18, consented to the study, and were comfortable with participating in the study in English, they were introduced to the study framing. We asked participants to imagine they were developing code for an app called CitizenMeasure, "a new global monitoring system that will allow citizen-scientists to travel to remote locations and make measurements about such issues as water pollution, deforestation, child labor, and human trafficking. Please keep in mind that our citizen-scientists may be operating in locations that are potentially dangerous, collecting information that powerful interests want kept secret. Our citizen scientists may have their devices confiscated and hacked." We hoped that this framing would pique participants' interest and motivate them to make a strong effort to write secure code. We also provided brief instructions for using the study infrastructure, which we describe next.

4.3.4 Experimental Infrastructure

After reading the study introduction and framing, participants were redirected to the tasks themselves. Our aim was to conduct an online developer study in which real developers would write and test real cryptographic code in our environment. We wanted to capture the code that they typed and their program runs. We wanted to control the study environment (Python version, available libraries) and collect data about their progress in real time. To achieve this, we used Jupyter Notebook [133], which allowed participants to write and run Python code in their browser, using the Python installation from our server. We instrumented the notebook to frequently snapshot the participant's code, as well as to detect and store copy&paste events. All this information was stored on the server.

We configured Notebook (version 4.2.1) with Python 2.7.11 and all five tested cryptographic libraries. To prevent interference between participants, each participant was assigned to a Notebook running on a separate Amazon Web Service (AWS) instance. We maintained a pool of prepared instances so that each new participant could begin without waiting for an instance to boot. Instances were shut down when each participant finished, to avoid between-subjects contamination.

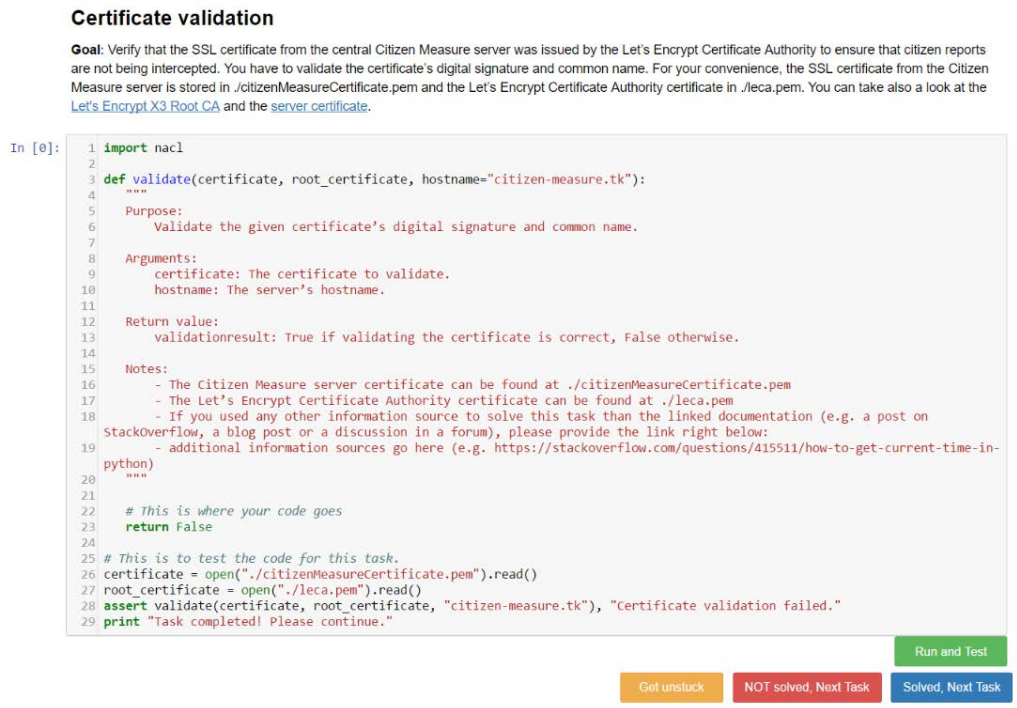

FIGURE 4.1: An example of the study's task interface.

Tasks were shown one at a time, with a progress indicator showing that the participant had completed, e.g., 1 of 3 tasks. For each task, participants were given buttons to "Run and test" their code, and to move on using "Solved, next task" or "Not solved, but next task." After each button press, we stored the participant's current code, along with metadata like timing, in a remote database. An example Notebook is shown in Figure 4.1.

Allowing participants to write and execute Python code presents serious security concerns. To mitigate this, we removed all unnecessary packages from the AWS image. We used the AWS firewall to restrict incoming traffic to port 80 and prevent outgoing traffic other than to our study database, which was password protected and restricted to sanitized insert commands. All instances were shut down within 4 hours of the last observed participant activity.

4.3.5 Task Design

We designed tasks that were short enough so that the uncompensated participants would be likely to complete them before losing interest, but still complex enough to be interesting and allow for some mistakes. Most importantly, we designed tasks to model real world problems that Python developers could reasonably be expected to encounter in their professional career. We chose two symmetric-encryption tasks: generating an encryption key and storing it securely in a password-protected file, and using the key to encrypt and decrypt text. We chose three asymmetric tasks: generating a key pair and storing the private key securely, using the public key to encrypt and the private key to decrypt, and validating an X.509 certificate.

Most of the libraries we chose support most of these tasks (Table 4.2). Unfortunately, task coverage by the libraries was not uniform: Keyczar and PyNaCl do not support secure key storage. The Keyczar documentation encourages generating keys at the command line; this can be worked around in the API, but it is not

Library	Current Version	Designed for Usability	Symmetric Key Generation	Symmetric Encryption/Decryption	Secure Symmetric Key Storage	Asymmetric Key Generation	Asymmetric Encryption/Decryption	Secure Asymmetric Key Storage	Certificate Validation
PyCrypto	2.6.1	○	●	●	●	●	●	●	○
M2Crypto	0.25.1	○	●	●	●	●	●	●	●
cryptography.io	1.4	●	●	●	●	●	●	●	●
Keyczar	0.716	●	◐	●	○	◐	●	○	◐
PyNaCl	1.0.1	●	●	●	○	●	●	○	◐

● = fully applies; ◐ = partly applies; ○ = does not apply

TABLE 4.2: Features and popularity for the five cryptography libraries we tested. Popularity data was updated as of Aug. 11, 2016.

straightforward to do so. Keyczar and PyNaCl do not support certificate validation directly, but it is possible to extract the public key and manually verify the signature. Finally, PyCrypto does not support certificate validation at all.

To account for cases where the library does not fully support the task, we offered participants the option to skip a task.

For each task, participants were provided with stub code and some commented instructions. These stubs were designed to make the task clear and ensure the results could be easily evaluated, without providing too much scaffolding. We also provided a main method pre-filled with code to test the provided stubs. This helped orient participants and saved time, but it did prevent us from learning how participants might have designed their own tests.

We also asked participants to please use only the included documentation for their assigned library, if at all possible, and to report (in comments) any additional documentation resources they consulted.

4.3.6 Python Cryptographic Libraries we Included

We briefly review the available features and documentation for each library we selected for our experiment (Table 4.2).

PyCrypto. The Python cryptographic toolkit PyCrypto [148] is Python's most popular cryptographic library. Developers can choose among several encryption and hashing algorithms and modes of operation, and may provide initialization vectors (IVs).

PyCrypto comes with primarily auto-generated documentation that includes minimal code examples. The documentation recommends the Advanced Encryption Standard (AES) and provides an example, but also describes the weaker Data Encryption Standard (DES) as cryptographically secure. The documentation warns

against weak exclusive-or (XOR) encryption. However, the documentation does not warn against using the default Electronic Code Book (ECB) mode, or the default empty IV, neither of which is secure.

M2Crypto. M2Crypto [46] is a binding to the well-known OpenSSL library that is more complete than alternative bindings such as pyOpenSSL. Although development on M2Crypto has largely ceased, the library is still widely used, and there is ample documentation and online usage examples, so we included it. M2Crypto supports all of the tasks we tested, including X.509 certificate handling. Developers are required to choose algorithms, modes of operation, and initialization vectors. M2Crypto comes with automatically generated documentation that includes no code examples or comments on the security of cryptographic algorithms and modes.

cryptography.io. cryptography.io has a stated goal of providing more usable security than other libraries by emphasizing secure algorithms, high-level methods, safe defaults, and good documentation [64]. It supports symmetric and asymmetric encryption as well as X.509 certificate handling. The documentation includes code examples that include secure options, with context for how they should be used. cryptography.io provides a high-level interface for some cryptographic tasks (such as symmetric key generation and encryption); this interface does not require developers to choose any security-sensitive parameters. The library also includes a lower-level interface, necessary for some asymmetric tasks and for encrypted key storage; this low-level interface does require developers to specify parameters such as algorithm and salt.

Keyczar. The library aims to make it easier to safely use cryptography, so that developers do not accidentally expose key material, use weak key lengths or deprecated algorithms, or improperly use cryptographic modes [275]. The documentation consists of an 11-page technical report that includes a few paragraphs regarding the program's design and a few abbreviated examples. Keyczar does not easily support X.509 certificate handling, encrypted key files, or password-based key derivation, but it does support digital signatures. There is no public API for key generation, but developers can generate keys by using an internal interface or by calling a provided command-line tool programmatically. Developers do not have to specify cryptographic algorithms, key sizes, or modes of operation.

PyNaCl. PyNaCl is a Python interface to libsodium [246], a cryptographic library designed with a focus on usability. The detailed documentation includes code examples with context for how to use them. PyNaCl supports both secure symmetric and asymmetric APIs without requiring the developer to choose cryptographic details, although the developer must provide a nonce. PyNaCl neither supports encrypted key storage nor password-based key derivation. X.509 certificate handling is also not supported directly; however, verifying digital signatures is supported.

4.3.7 Exit Survey

Once all tasks had been completed or abandoned, the participant was directed to a short exit survey. We asked for their opinions about the tasks they had completed and the library they used, including the standard System Usability Scale (SUS) [131] score for the library. We also collected their demographics and programming experience. The participant's code for each task was displayed (imported from our database) for their reference with each question about that task.

We were specifically interested in the participants' opinions about the usability of the API. To this end, we collected the SUS score, but we wanted to also investigate

in more depth. Prior work on API usability has suggested several concrete factors that affect an API's usability. We combined the cognitive dimensions framework [57] with usability suggestions from Nielsen and from Smith and Green [115, 173], and pulled out the factors that could most easily be evaluated via self-reporting from developers using the API. We transformed these factors into an 11-question scale (given in Appendix B.2) that focuses on the learnability of the API, the helpfulness of its documentation, the clarity of observed error messages, and other features. Our scale can be used to produce an overall score, as well as to target specific characteristics that impede the usability of each API. For this work, we treat this scale as exploratory; we correlate it with SUS and investigate its internal reliability in Section 4.4.6.

4.3.8 Evaluating Participant Solutions

We used the code submitted by our participants for each task, henceforth called a *solution*, as the basis for our analysis.

We evaluated each participant's solution to each task for both functional correctness and security. Every task was independently reviewed by two coders, using a codebook prepared ahead of time based on the capabilities of the libraries we evaluated. Differences between the two coders were adjudicated by a third coder, who updated the codebook accordingly. We briefly describe the codebook below.

Functionality. For each programming task, we assigned a participant a functionality score of 1 if the code ran without errors, passed the tests and completed the assigned task, or 0 if not.

Security. We assigned security scores only to those solutions which were graded as functional. To determine a security score, we considered several different security parameters. A participant's solution was marked secure (1) only if their solution was acceptable for every parameter; an error in any parameter resulted in a security score of 0.

Not all security parameters applied to all libraries, as some libraries do not allow users to make certain potentially insecure choices. Details of how the different security parameters applied to each library can be found in Table 4.3. Whenever a given library requires a developer to make a secure choice for a given parameter, we assign a full circle; if that parameter is not applicable in that library, we assign an empty circle. For example, for symmetric encryption, PyCrypto participants had to specify an encryption algorithm, mode of operation and an initialization vector (three full circles). However, PyNaCl participants did not have to care about these cryptographic details (three empty circles).

For key generation, we checked key size and proper source of randomness for the key material. We selected an appropriate key size for a particular algorithm (e.g., for RSA we required at least 2 048-bit keys [168]). For key storage we checked if encryption keys were actually encrypted and if a proper encryption key was derived from the password we provided. Depending on the library and task type, encrypting cryptographic key material requires the application of a key derivation function such as PBDKF2 [132]. For libraries in which developers had to pick parameters for PBKDF2 manually (cf. Table 4.3), we scored use of a static or empty salt, HMAC-SHA1 or below as the pseudorandom function, and less than 10 000 iterations as insecure [169]. For some libraries, participants had to select encryption parameters for one or more tasks; in these cases, we also scored the security of the chosen encryption algorithm, mode of operation, and initialization vector. For symmetric encryption, we scored ARC2, ARC4, Blowfish, (3)DES, and XOR as insecure,

Symmetric

	Key Generation	Key Storage				Key Derivation			Encryption		
	Size	Plain/ Encrypted	Algorithm	Mode	IV	Salt	PRF	Iterations	Algorithm	Mode	IV
PyCrypto	●	●	●	●	●	●	●	●	●	●	●
M2Crypto	●	●	●	●	●	●	●	●	●	●	●
cryptography.io	●	●	○	○	○	●	●	●	○	○	○
Keyczar	○	●	○	○	○	●*	●*	●*	○	○	○
PyNaCl	○	●	○	○	○	●*	●*	●*	○	○	○

Asymmetric

	Key Generation		Key Storage				Encryption		Certificate Validation			
	Type	Size	Plain/ Encrypted	Algorithm	Mode	IV	Padding	Nonce	Signature Verification	Hostname Check	CA Check	Date Check
PyCrypto	●	●	●	○	○	○	●	○	●	●	●	●
M2Crypto	●	●	●	○	○	○	●	○	●	●	●	●
cryptography.io	●	●	●	○	○	○	●	○	●	●	●	●
Keyczar	○	○	●	●*	●*	●*	○	○	●	●	●	●
PyNaCl	○	○	●	●*	●*	●*	○	●	●	●	●	●

TABLE 4.3: Security choices required by various libraries, as defined in our codebook. ● indicates the developer is required to make a secure choice, ○ indicates no such choice is required. Libraries that do not include a key derivation function, requiring the developer to fall back to Python's hashlib API, are indicated with *.

and AES as secure. We scored the ECB as an insecure mode of operation and scored Cipher Block Chaining (CBC), Counter Mode (CTR) and Cipher Feedback (CFB) as secure. Static, zero or empty initialization vectors were scored insecure. For asymmetric encryption we scored the use of OAEP/PKCS1 for padding as secure.

4.3.9 Limitations

As with any user study, our results should be interpreted in context. We chose an online study because it is difficult to recruit "real" developers (rather than students) for an in-person lab study at a reasonable cost. Choosing to conduct an online study allowed us less control over the study environment; however, it allowed us to recruit a geographically diverse sample. Because we targeted developers, we could not easily take advantage of services like Amazon's Mechanical Turk or survey sampling firms. Managing online study payments outside such infrastructures is very challenging; as a result, we did not offer compensation and instead asked participants to generously donate their time. As might be expected, the combination of unsolicited recruitment emails and no compensation led to a strong self-selection effect, and we expect that our results represent developers who are interested and motivated enough to participate. Comparing the full invited sample to the valid participants (see Section 4.4.1) suggests that indeed, more active GitHub users were more likely to participate. That said, these limitations apply across conditions, suggesting that comparisons between conditions are valid. Further, we found almost no results (Section 4.4) correlated with self-reported Python experience.

In any online study, some participants may not provide full effort, or may answer haphazardly. In this case, the lack of compensation reduces the motivation to answer in a manner that is not constructive; those who are not motivated will typically not participate. We attempt to remove any obviously low-quality data (e.g., responses that are entirely invective) before analysis, but cannot discriminate perfectly. Again, this limitation should apply across conditions without affecting condition comparisons.

Our study examines how developers use different cryptographic libraries. Developers who reach this point already recognize that they need encryption and have

chosen to use an existing library rather than trying to develop their own mechanism; these are important obstacles to secure code that cannot be addressed by better library design. Nonetheless, we believe that evaluating and improving cryptographic libraries is a valuable step toward more secure development.

Finally, we are comparing libraries overall: this includes their API design and implementation as well as their documentation. The quality of both varies significantly across the libraries. Our results provide insight into the contributions made by documentation and by API design to a library's overall success or failure, but future work is needed to further explore how the two operate independently.

4.4 Study results

Study participants experienced very different rates of task completion, functional success, and security success as a function of which library they were assigned and whether they were assigned the symmetric or asymmetric tasks. Overall, we find that completion rate, functional success, and self-reported usability satisfaction showed similar results: cryptography.io, PyCrypto and (to some extent) PyNaCl performed best on these metrics. The security results, however, were somewhat different. PyCrypto and M2Crypto were worst, while Keyczar performed best. PyNaCl also had strong security results; cryptography.io exhibited strong security for the symmetric tasks but poor security for asymmetric tasks. These results suggest that the relationship between "usable" design, developer satisfaction, and security outcomes is a complex one.

4.4.1 Participants

In total, we sent 52 448 email invitations. Of these, 5 918 (11.3%) bounced, and another 698 (1.3%) requested to be removed from our list, a request we honored.

A total of 1 571 people agreed to our consent form; 660 (42.0%) dropped out without taking any action, most likely because the initial task seemed too difficult or time-consuming. The other 911 proceeded through at least one task; of these, 337 proceeded to the exit survey, and 282 completed it with valid responses.[1] Of these, 26 were excluded for failing to use their assigned library. Unless otherwise noted, we report results for the remaining 256 participants, who proceeded through all tasks, used their assigned library, and completed the exit survey with valid responses.

An additional 61 participants attempted to reach the study but encountered technical errors in our infrastructure, mainly due to occasional AWS pool exhaustion during times of high demand.

Our 256 participants reported ages between 18 and 63 (mean 29.4, sd 7.9), and the vast majority of them reported being male (238, 93.0%). We successfully reached the professional developer demographic we targeted. Almost all (247, 96.5%) had been programming in general for more than two years, and 81.2% (208) had been programming in Python for more than two years. Most participants (196, 76.6%) reported programming as (part of) their primary job; of those, 147 (75.0%) used Python in their primary job. Most participants (195, 76.2%) said they had no IT-security background.

While the developers we invited represent a random sample from GitHub, our valid participants are a small, self-selected subset. Table 4.4 and Figure 4.2 detail

[1] We define invalid responses as providing straight-line answers to all questions or writing off-topic or abusive comments in free-text responses.

	Invited	Valid
Hireable	19.5%	37.9%
Company listed	28.0%	42.2%
URL to Blog	34.7%	55.6%
Biography added	8.1%	16.3%
Location provided	49.9%	75.8%
Public gists (median)	0	1
Public repositories (median)	12	20
Following (users, median)	1	2
Followers (users, median)	3	7
GitHub profile creation (days ago, median)	1 415	1 589
GitHub profile last update (days ago, median)	50	38

TABLE 4.4: GitHub demographics for the 50 000 invited users and for our 256 valid participants.

available GitHub demographics for both groups. Our participants appear to be slightly more active on GitHub than average: owning more public repositories, having more followers, having older accounts, and being more likely to provide optional profile information. This may correspond to their self-reported high levels of programming experience and professional status.

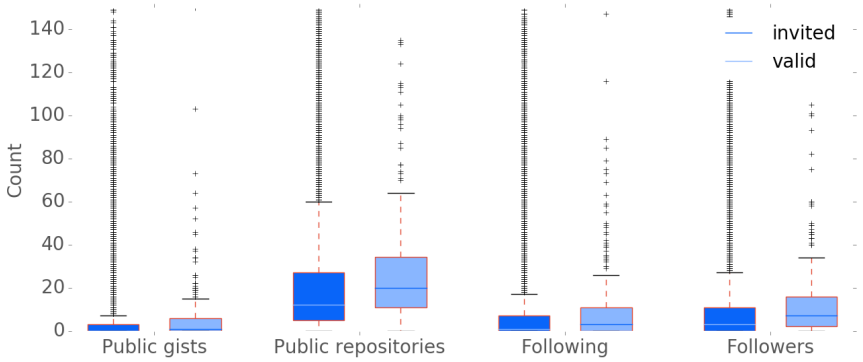

FIGURE 4.2: Boxplots comparing our invited participants (a random sample from GitHub) with those who provided valid participation. The center line indicates the median; the boxes indicate the first and third quartiles. The whiskers extend to ±1.5 times the interquartile range. Outliers greater than 150 were truncated for space.

4.4.2 Regression models

In the following subsections, we apply regression models to analyze our results in detail. To analyze binary outcomes (e.g., secure vs. insecure), we use logistic regression; to analyze numeric outcomes (e.g., SUS score), we use linear regression. When we consider results on a per-task rather than a per-participant basis (for security and functionality results, as well as perceived security), we use a mixed model that adds a random intercept to account for multiple tasks from the same participant.

For each regression analysis, we consider a set of candidate models and select the model with the lowest Akaike Information Criterion (AIC) [43]. The included factors are described in Table 4.5. We consider candidate models consisting of the required factors *library* and *encryption mode*, as well as (where applicable) the participant random intercept, plus every possible combination of the optional variables.

We report the outcome of our regressions in tables. Each row measures change in the analyzed outcome related to changing from the *baseline* value for a given factor to a different value for that factor (e.g., changing from asymmetric to symmetric encryption). Logistic regressions produce an odds ratio (O.R.) that measures change in likelihood of the targeted outcome; baseline factors by construction have O.R.=1. For example, Table 4.7 indicates that M2Crypto participants were $0.55\times$ as likely to complete all tasks as participants in the baseline PyCrypto condition. In constrast, linear regressions measure change in the absolute value of the outcome; baseline factors by construction have coef=0. In each row, we also provide a 95% confidence interval (C.I.) and a p-value indicating statistical significance.

For each regression, we set the library PyCrypto as the baseline, as it has the most download counts of all libraries we included in our study, and can therefore be considered as the most common "default" crypto library for Python. In addition, we used the set of symmetric tasks as the baseline, as these correspond to the simpler and more basic form of encryption. All baseline values are given in Table 4.5.

Factor	Description	Baseline
Required factors		
Library	The cryptographic library used.	PyCrypto
Encryption mode	Asymmetric or Symmetric	Symmetric
Optional factors		
Experienced	True if a programming in Python is part of participant's job, and/or if participant has been programming in Python for more than five years; otherwise false. Self-reported.	False
Security background	True or false, self-reported.	False
Library experience	Whether the participant has used the library before, seen code that used it but not used it themselves; or neither. Self-reported.	No experience
Copy-paste	Whether the participant pasted code during this task. Measured, per-task regressions only.	False
Library $\times$ Mode	Interaction between the library and encryption mode factors described above.	cryptography.io :asymmetric

TABLE 4.5: Factors used in regression models. Categorical factors are individually compared to the baseline. Final models were selected by minimum AIC; candidates were defined using all possible combinations of optional factors, with both required factors included in every candidate.

4.4.3 Dropouts

We first examine how library and encryption mode affected participants' dropout rates, as we believe that dropping out of the survey is a first (if crude and oversimplified) measure of how much effort was required to solve the assigned tasks with the assigned library. Table 4.6 details how many participants in each condition reached each stage of the study.

We test whether library and encryption mode affect dropout rate using a logistic regression model (see Section 4.4.2) examining whether each participant who consented proceeded through all tasks and started the exit survey. (We use the start of the survey here because dropping out at the survey stage seems orthogonal to library

Library	Mode	Consented	Started Survey	Total Valid
PyCrypto	sym	136	48	41
	asym	175	37	24
M2Crypto	sym	157	36	20
	asym	174	35	27
cryptography.io	sym	136	48	39
	asym	174	22	19
Keyczar	sym	136	26	20
	asym	173	24	17
PyNaCl	sym	136	34	29
	asym	174	27	20
Total		1 571	337	256

TABLE 4.6: The number of participants who progressed through each phase of the study, by condition. Each column is a subset of the previous columns.

type.) For this model, we include only the library-encryption mode interactions as an optional factor, because we do not have experience or security background data for the participants who dropped out.

The final model (see Table 4.7) indicates that asymmetric-encryption participants were only about half as likely to proceed through all tasks as participants assigned to symmetric encryption, which was statistically significant. Compared to the "default" choice of PyCrypto, participants assigned to M2Crypto and Keyczar were about half as likely to proceed through all tasks, which was also statistically significant. PyNaCl exhibits a higher dropout rate than PyCrypto; however, this trend was not significant. cryptography.io matches PyCrypto's dropout rate. Although the two-way interactions are included in the final model, none exhibits a significant result.

Overall, these results suggest that PyCrypto (approximate default) and cryptography.io (designed for usability, with relatively complete documentation) were least likely to drive participants away. Keyczar, also designed for usability, performed worst on this metric.

Factor	O.R.	C.I.	*p*-value
M2Crypto	0.55	[0.33, 0.91]	0.02*
cryptography.io	1.00	[0.61, 1.64]	1
Keyczar	0.43	[0.25, 0.75]	0.003*
PyNaCl	0.61	[0.36, 1.03]	0.065
asymmetric	0.49	[0.3, 0.81]	0.006*
M2Crypto:asymmetric	1.72	[0.83, 3.57]	0.144
cryptography.io:asymmetric	0.54	[0.25, 1.16]	0.112
Keyczar:asymmetric	1.39	[0.63, 3.05]	0.418
PyNaCl:asymmetric	1.12	[0.53, 2.39]	0.768

TABLE 4.7: Results of the final logistic regression model examining whether participants who consented proceeded through all tasks and continued to the survey. Odds ratios (O.R.) indicate relative likelihood of continuing. Statistically significant factors indicated with *. See Section 4.4.2 for further details.

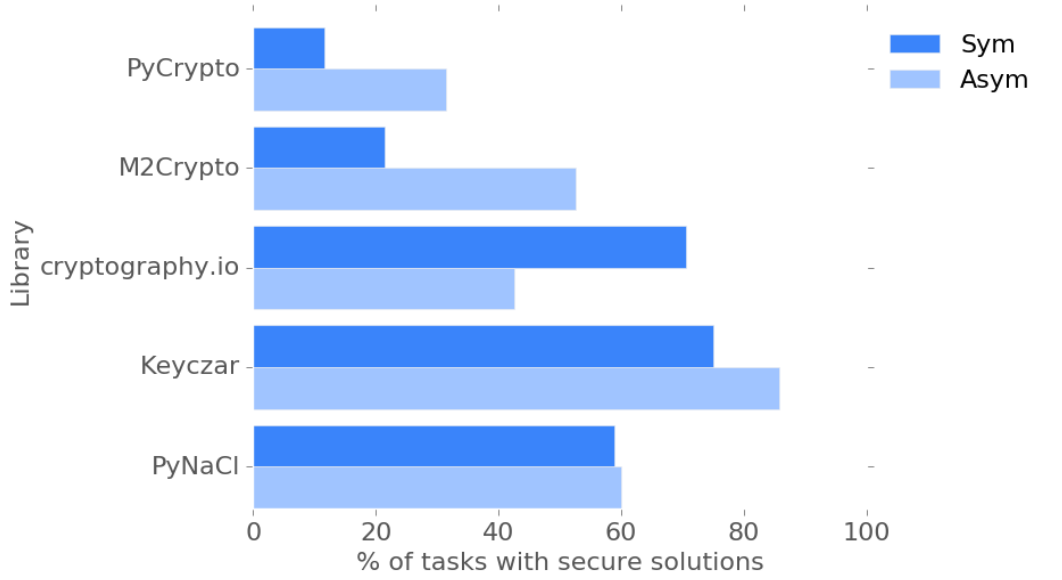

FIGURE 4.3: Percentage of tasks for which participants generated functional solutions, by condition.

4.4.4 Functionality results

We next discuss the extent to which participants were able to produce functional solutions—that is, solutions that produced a key or encrypted and decrypted some content without generating an exception.[2] We observed a wide variance in functional results across libraries and encryption types, ranging from asymmetric Keyczar (13.7% functional) to symmetric cryptography.io and symmetric PyNaCl (89.5% and 87.9% functional respectively).

Figure 4.3 illustrates these results.

To examine these results more precisely, we applied a logistic regression, as described in Section 4.4.2, to model the factors that affect whether or not each individual task was marked as functional. The final model (see Table 4.8) shows that M2Crypto and Keyczar are significantly worse for functionality than the baseline PyCrypto; cryptography.io and PyNaCl appear slightly better, but the difference is not significant. Most notably, Keyczar is estimated as only 10% as likely to produce a functional result. By comparing confidence intervals, we see that Keyczar is also significantly worse than PyNaCl and cryptography.io. The results also show that symmetric tasks were about 6× (0.16^{-1}) as likely as asymmetric tasks to have functional solutions, and that using code generated via copy-and-paste improves a task's odds of functionality about 3× (both significant). The participant's Python experience level, security background, and experience with their assigned library do not appear in the final model, suggesting they are not significant factors in the functionality results.

In general, the set of asymmetric cryptography tasks was harder to solve in a functionally correct way than the set of symmetric cryptography tasks. This seems to be largely because we included X.509 certificate handling in the set of asymmetric cryptography tasks. Two of the libraries specifically designed to be easy to use (Keyczar and PyNaCl) do not support X.509 certificate handling out of the box, so these tasks had to be done via workarounds or could not be solved at all. On the

[2]Participants who skipped a task are counted as functionally incorrect for that task.

other hand, the low-level X.509 certificate APIs of M2Crypto and PyCrypto require developers to deal with many cryptographic details (e.g., root certificate stores and certificate details such as the Common Name or Subject Alternative Name), which might have an impact on functionality in addition to security.

The only significant interaction in the final model is between M2Crypto and asymmetric tasks: these tasks were about 8× more likely than expected to be marked functional. Indeed, M2Crypto is the only library (see Figure 4.3) for which symmetric tasks were (slightly) less functional than asymmetric tasks. We hypothesize that this is caused by the requirement that developers have to choose many cryptographic details for both symmetric and asymmetric encryption in M2Crypto.

Factor	O.R.	C.I.	*p*-value
M2Crypto	0.26	[0.09, 0.69]	0.007*
cryptography.io	1.68	[0.61, 4.61]	0.311
Keyczar	0.10	[0.04, 0.26]	< 0.001*
PyNaCl	1.58	[0.55, 4.56]	0.394
asymmetric	0.16	[0.07, 0.38]	< 0.001*
copy-paste	3.29	[1.97, 5.49]	< 0.001*
M2Crypto:asymmetric	8.14	[2.29, 28.95]	0.001*
cryptography.io:asymmetric	1.53	[0.4, 5.75]	0.532
Keyczar:asymmetric	1.50	[0.36, 6.22]	0.578
PyNaCl:asymmetric	0.49	[0.13, 1.86]	0.293

TABLE 4.8: Results of the final logistic regression mixed model examining which factors correlate with task functionality. Odds ratios indicate relative likelihood of a task being functionally correct. Statistically significant values indicated with *. See Section 4.4.2 for further details.

4.4.5 Security results

Next, we consider whether participants whose code was functional also produced secure solutions. As with functionality, we observed a broad range of results (see Figure 4.4). Overall, Keyczar was notably secure (for a small sample) and PyCrypto and to a lesser extent M2Crypto were notably insecure.

We again apply logistic regression (Section 4.4.2) to investigate the factors that influence security; we include only functional task solutions in this analysis. The results are shown in Table 4.9. The final model shows that compared to the baseline PyCrypto, every library appears to produce better security; all of these except M2Crypto are significant. At the extreme, Keyczar is estimated almost 25× as likely to produce a secure solution. This is particularly notable because Keyczar was so difficult: only 16 and seven participant tasks, respectively, exhibited functional symmetric and asymmetric solutions, but 12 and six of these respectively were secure, the highest per-capita of any library. The regression results also show that at baseline, asymmetric tasks were about 3× more likely to exhibit secure code than symmetric tasks. The final model also indicates that tasks from participants with a security background were about 1.5× more likely to be secure; Python experience level and experience directly with the assigned library do not seem to affect security noticeably, as they do not appear in the final model. The only significant interaction term is between cryptography.io and asymmetric: cryptography.io is the only library for which asymmetric performed less securely. We hypothesize that this

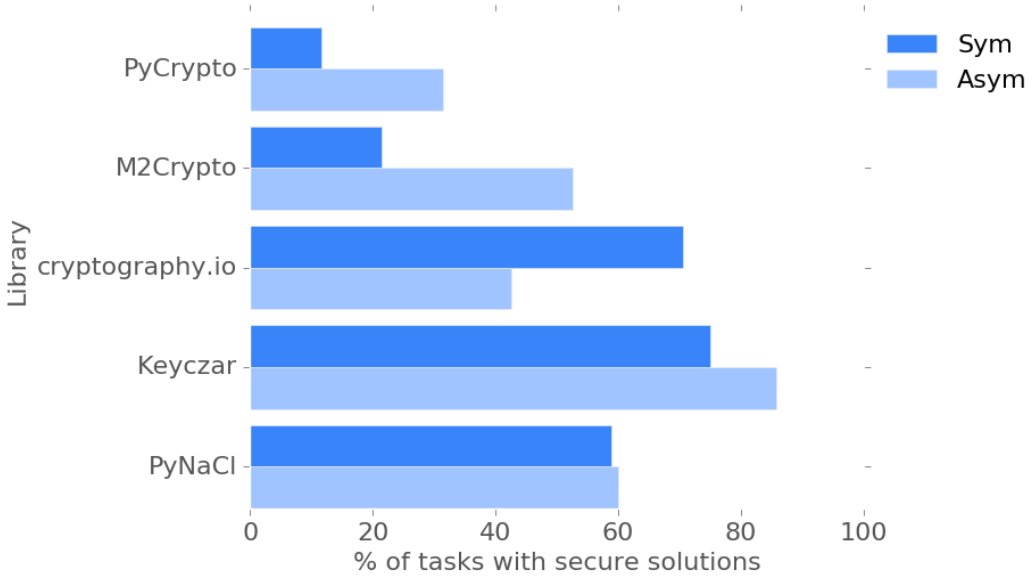

FIGURE 4.4: Percentage of tasks with secure solutions, considering only tasks with functional solutions, by condition.

Factor	O.R.	C.I.	*p*-value
M2Crypto	2.20	[0.68, 7.11]	0.186
cryptography.io	19.34	[7.78, 48.03]	<0.001*
Keyczar	24.54	[6.31, 95.43]	<0.001*
PyNaCl	11.29	[4.46, 28.61]	<0.001*
asymmetric	3.58	[1.28, 10.03]	0.015*
sec. bkgrd.	1.57	[0.94, 2.61]	0.083
M2Crypto:asymmetric	1.09	[0.25, 4.73]	0.909
cryptography.io:asymmetric	0.08	[0.02, 0.31]	<0.001*
Keyczar:asymmetric	0.54	[0.04, 7.37]	0.642
PyNaCl:asymmetric	0.29	[0.07, 1.2]	0.088

TABLE 4.9: Results of the final logistic regression mixed model examining which factors correlate with task security, among only tasks that were functional. Odds ratios indicate relative likelihood of a solution being secure. Statistically significant values indicated with *. See Section 4.4.2 for further details.

is because the symmetric tasks could be completed using the library's high-level "recipes" layer, while the asymmetric tasks required the participant to work with the low-level "hazmat" layer.

Security perception. In the exit survey, we showed participants the code they had written to solve each task and asked them (on a five-point Likert scale from Strongly Agree to Strongly Disagree) whether they thought their solution was secure. We did not define security, as we wanted to know whether our participants were satisfied with the security properties of their code in general, rather than meeting a specific threat model. Across all libraries, the majority of our participants were convinced that their solution was secure. The median (excluding 10% of tasks for which participants answered "I don't know") was no lower than "neutral" across all combinations of libraries and encryption modes; security confidence was highest for cryptography.io and PyNaCl (both encryption modes), as well as PyCrypto and

Factor	O.R.	C.I.	*p*-value
M2Crypto	0.59	[0.25, 1.38]	0.221
cryptography.io	0.58	[0.27, 1.27]	0.176
Keyczar	0.25	[0.05, 1.3]	0.099
PyNaCl	0.62	[0.27, 1.46]	0.277
asymmetric	1.32	[0.72, 2.42]	0.373
sec. bkgrd.	1.65	[0.86, 3.14]	0.13

TABLE 4.10: Results of the final logistic regression mixed model examining factors correlating with erroneous belief that a task is secure. Odds ratios indicate relative likelihood of this belief. Some trends are observable, but no results are statistically significant. See Section 4.4.2 for further details.

Keyczar (asymmetric), all of which had median value “agree."

In considering these answers, we are most interested in tasks for which we rated the solution insecure, but the participant agreed or strongly agreed that their solution for that task was secure. These situations are potentially dangerous, as the developer mistakenly believes they have achieved security. Overall, 78 of 396 tasks (19.7%) fell into this category, a disappointingly high number. To examine factors that correlate with this situation, we applied a mixed-model logistic regression, as described in Section 4.4.2, with outcome *dangerous error* or *not* per task. The results are shown in Table 4.10. Although some trends are observable, the final model finds no significant results; this suggests that at least at this sample size, no particular factors were significantly associated with a higher likelihood of erroneous belief.

4.4.6 Participant opinions

Our self-reported usability metrics reveal large differences between the libraries. Table 4.11 lists the average SUS scores by condition. Overall, PyNaCl and cryptography.io performed best, while M2Crypto and Keyczar performed worst. Overall, these SUS scores are quite low; a score of 68 is considered average for end-user products and systems [131], and even our best-performing condition does not reach this standard. This suggests that even the most usable libraries we tested have considerable room for improvement.

Using a linear regression model (see Section 4.4.2), we analyzed the impact of library and encryption mode, shown in Table 4.12. We find that M2Crypto and Keyczar are significantly less usable than the baseline PyCrypto; PyNaCl is significantly more usable. Unsurprisingly, symmetric-condition participants reported significantly more usability than asymmetric-condition participants. The final model indicates that security background and having seen the assigned library before were both associated with a significant increase in usability. Having used the library before was associated with an increase relative to no familiarity, but this trend was not significant, probably because of the very small sample size: only 18 participants reported having used their assigned library before. Python experience was included in the final model but was not a signficiant covariate; the final model did not include any interactions between library and encryption mode.

We compiled our additional usability questions, drawn from prior work as described in Section 4.3.7, into a score out of 100 points. The results were similar to the SUS, and in fact, the two scores were significantly correlated (Kendall's τ=0.65,

Library	Mode	Mean SUS	Mean API Scale
PyCrypto	sym	63.9	64.2
	asym	47.8	52.5
M2Crypto	sym	33.9	32.5
	asym	36.4	35.6
cryptography.io	sym	67.2	67.7
	asym	52.3	61.6
Keyczar	sym	40.8	40.9
	asym	32.5	26.9
PyNaCl	sym	67.2	66.8
	asym	59.5	57.1

TABLE 4.11: Mean SUS scores and scores on our new API usability scale, by condition.

Factor	Coef.	C.I.	p-value
M2Crypto	-20.57	[-27.62, -13.52]	<0.001*
cryptography.io	5.04	[-1.52, 11.61]	0.131
Keyczar	-18.07	[-25.85, -10.3]	<0.001*
PyNaCl	7.56	[0.48, 14.64]	0.036*
asymmetric	-9.60	[-14.13, -5.08]	<0.001*
experienced	3.79	[-1.33, 8.91]	0.146
sec. bkgrd.	6.22	[0.98, 11.46]	0.02*
seen lib	6.62	[0.39, 12.85]	0.037*
used lib	3.33	[-5.95, 12.6]	0.481

TABLE 4.12: Linear regression model examining SUS scores. The coefficient indicates the average difference in score between the listed factor and the base case. Significant values indicated with *. $R^2 = 0.376$. See Section 4.4.2 for further details.

$p < 0.001$). Using Cronbach's alpha, we determined that the scale's internal reliability was high ($\alpha = 0.98$).

Table 4.13 shows the results of a linear regression examining score on our scale. As before, M2Crypto and Keyczar are significantly worse than PyCrypto. Using this measure, cryptography.io is significantly better than PyCrypto, while PyNaCl is better than PyCrypto but not significantly so. Also as before, significantly higher scores were correlated with symmetric tasks and with having seen the assigned library before. Having used the library before was again correlated with higher scores, but not significantly so, probably due to sample size. Security background was included in the final model but not significant; Python experience and interactions between library and encryption mode were not included in the final model.

The answers to questions about the API documentation indicate that Keyczar and M2Crypto have a sizable problem with their documentation: Our participants consistently answered that they found neither helpful explanations nor helpful code examples in the documentation, and that they had to spend a lot of time reading the documentation before they could solve the tasks. Altogether, they found the documentation for Keyczar and M2Crypto not helpful. This corresponded to responses saying that the tasks were not straightforward to implement for these two libraries. Interestingly, for cryptography.io, the perceived effort that had to be invested into understanding the library in order to be able to work on the tasks was the lowest.

Factor	Coef.	C.I.	*p*-value
M2Crypto	-22.44	[-28.54, -16.35]	<0.001*
cryptography.io	7.21	[1.45, 12.97]	0.014*
Keyczar	-21.59	[-28.41, -14.77]	<0.001*
PyNaCl	5.66	[-0.5, 11.82]	0.072
asymmetric	-8.00	[-11.99, -4.02]	<0.001*
sec. bkgrd.	3.94	[-0.66, 8.54]	0.093
seen lib	6.60	[1.12, 12.09]	0.019*
used lib	6.74	[-1.41, 14.88]	0.104

TABLE 4.13: Linear regression model examining scores on our cognitive-dimension-based scale. The coefficient indicates the average difference in score between the listed factor and the base case (PyCrypto and symmetric, respectively). Significant values indicated with *. $R^2 = 0.466$. See Section 4.4.2 for further details.

For cryptography.io, PyNaCl, and PyCrypto, the developers felt that after having used the library to solve the tasks, they had a pretty good understanding of how the library worked.

For color, we include a few exemplar quotes from our participants who chose to comment on the documentation. One participant said the Keyczar documentation was "awful and doesn't seem to document its Python API at all." A second said, "I don't understand why you have an API with no search feature and functional descriptions. This is insane," and a third commented that "The linked document is so unkind that I must read the code." A third Keyczar participant left an ASCII-art comment spelling out "Your documentation is bad and you should feel bad."

One participant assigned to M2Crypto called the documentation "solidly awful," "just terrible," and "completely unusable." The same participant inquired whether our request to use this library was "a joke" or "part of the study." Other M2Crypto participants said "the linked documentation is wildly insufficient" and M2Crypto's "interface is arcane and documentation hard to understand." Several participants assigned to this library commented that they had to revert to Stack Overflow posts or blog entries found via search engines to be able to work on the tasks at all.

In contrast, one participant working with cryptography.io called a tutorial contained in the documentation "amazing!" while stating that "The comparable OpenSSL docs make one want to jump off a cliff." Another said the documentation "was confusing at first, but later I got the hang of it."

4.4.7 Examining individual tasks

Success in solving the tasks varied not only across libraries, but also across individual tasks, as illustrated in Figure 4.5. We analyze these results for trends, rather than statistical significance, to avoid diluting our statistical power by testing the same results in multiple ways.

Encryption proved easiest. Symmetric participants achieved 85.2% functional success, with 70.1% of those rated secure; 72.0% of asymmetric encryption tasks were functional, with 78.8% of those rated secure. In contrast, the hardest task to solve overall dealt with certificate validation. Only 22.4% of asymmetric participants were able to provide a functional solution, and not a single one was secure. Key generation tasks fell in the middle.

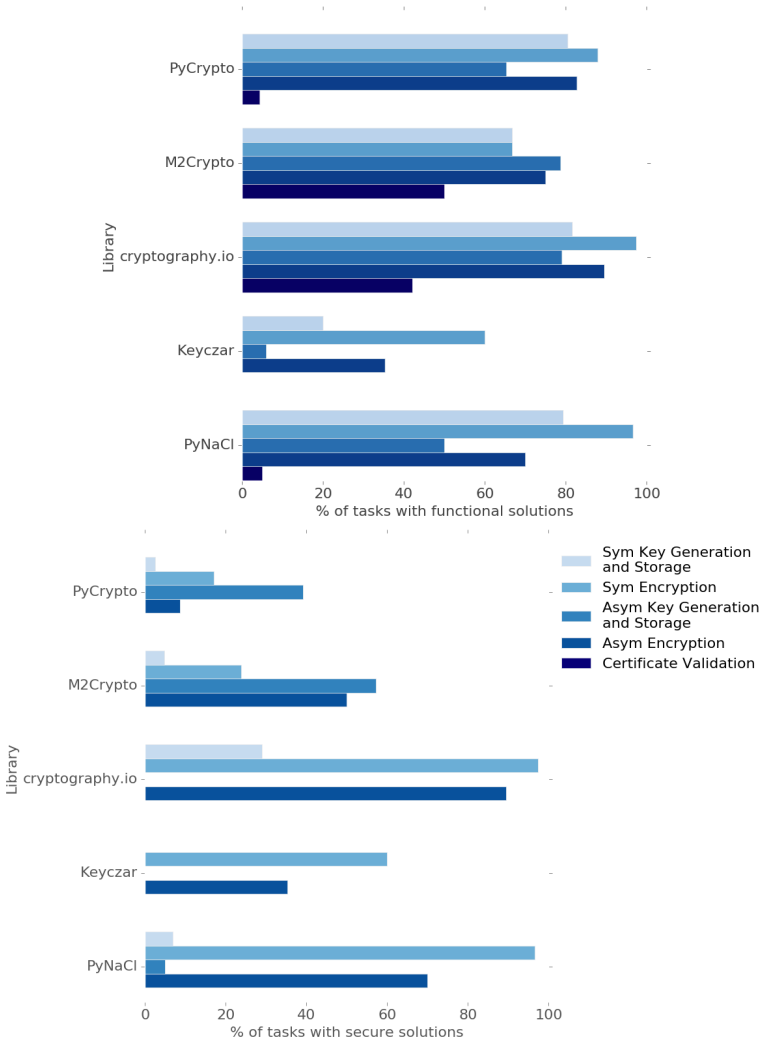

FIGURE 4.5: Percentage of tasks with functionally correct solutions (left), and percentage of functional solutions that were rated secure (right), organized by library and task type.

Investigating security errors. We also examined trends in the types of security errors made by our participants. (For a full accounting, see Table B.1 in Appendix B.1.)

We first consider symmetric cryptography, and in particular situations where participants were allowed to make security choices. Only M2Crypto and PyCrypto allow developers to choose an encryption algorithm; interestingly, all 11 PyCrypto participants selected DES (insecure), but no M2Crypto participants chose an insecure algorithm. While M2Crypto's official API documentation does not provide code examples, the first results on Google when searching "m2crypto encryption" provide code snippets that use AES. The PyCrypto documentation does provide code examples for symmetric encryption and discourages the use of DES as a weak encryption algorithm. However, the first Google results when searching "pycrypto encryption" provide code examples that use DES. Nine of the 11 participants who used DES mentioned specific blog posts and Stack Overflow posts that we later determined to have insecure code snippets. Similarly, allowing developers to pick modes of operation resulted in relatively many vulnerabilities. PyCrypto participants chose the insecure ECB as mode of operation explicitly or did not provide a

mode of operation parameter at all (ECB is the default). As with selecting an encryption algorithm, affected participants reported using blog posts and Stack Overflow posts containing insecure snippets as information sources. PyCrypto participants chose static IVs more frequently than those using other libraries; interestingly, this corresponds to not mentioning the importance of a truly random IV in the documentation. Relatedly, requiring developers to pick key sizes manually frequently resulted in too-small keys, across libraries.

Interestingly, PyCrypto participants were most likely to fail to use any key derivation function, possibly because the documentation uses a plain string for an encryption key. PyNaCl and PyCrypto participants used an insecure custom key derivation function more frequently than participants in other conditions: they frequently used a simple hash function for key stretching. cryptography.io participants, in contrast, performed exceedingly well on this task, likely because the included PBKDF2 function is well documented and close to the symmetric encryption example. On the negative side, cryptography.io users picked static salts for PBKDF2 more frequently than others, even though the code example in the API documentation uses a random salt; however, no explanation on the importance of using a random value is given. Storing encryption keys in plaintext rather than encrypted was also common across all libraries.

Generating and storing asymmetric keys was significantly less vulnerable to weak cryptographic choices. Only PyCrypto and M2Crypto participants failed to pick sufficiently secure RSA key sizes, potentially due again to insecure code examples (mentioning 1024-bit keys) among the top Google search results. Since all libraries but Keyczar and PyNaCl provide a private-key export function that offers encryption, asymmetric private-key storage had comparably few insecurities. However, PyNaCl users had to manually encrypt their private key and ran into similar security problems as the symmetric-encryption users mentioned above. Asymmetric encryption produced relatively few security errors.

Certificate validation was the most challenging task. Across all libraries, participants had trouble properly implementing signature validation, hostname verification, CA checks, and validity checks. This may be caused by task complexity and insufficient API support.

4.5 Discussion

Our results suggest that usability and security are deeply interconnected in sometimes surprising ways. We distill some high-level findings derived from our individual results and suggest future directions for library design and further research.

Simplicity does promote security (to a point). In general, the simplified libraries we tested produced more secure results than the comprehensive libraries, validating the belief that simplicity is better. Further, cryptography.io proved secure for the symmetric tasks (primarily doable via the simplified "recipes" layer) but not for the asymmetric tasks (primarily requiring use of the complex "hazmat" layer). This reinforces both the idea that simplicity promotes security and the need for simplified libraries to offer a broader range of features.

However, even simplified libraries did not entirely solve the security problem; in all but one condition, the rate of security success was below 80%. These security errors were frequently caused by missing features (discussed next). Worse, for 20% of functional solutions, participants rated their code as secure when it was not; this indicates a dangerous gap in recognition of potential security problems.

Features and documentation matter for security. Several of the libraries we selected did not (or not well) support tasks auxiliary to encryption and decryption, such as secure key storage and password-based key generation. These missing features caused many of the insecure results in the otherwise-successful simplified libraries. We argue that to be usably secure, a cryptographic API must support such auxiliary tasks, rather than relying on the developer to recognize the potential for danger and identify a secure alternate solution. Further, we suggest that cryptographic APIs should be designed to support a reasonably broad range of use cases; requiring developers to learn and use new APIs for closely related tasks seems likely to drive them back to comprehensive libraries like PyCrypto or M2Crypto, which pose security risks.

Documentation is also critical. PyCrypto, for example, contains symmetric encryption examples that use AES in ECB mode, which is prima facie insecure. Participants who left the PyCrypto documentation to search for help on Stack Overflow and blogs often ended up with insecure solutions; this suggests the importance of creating official documentation that is useful enough to keep developers from searching out unvetted, potentially insecure alternatives. Many participants copied these examples in their solutions. In contrast, the excellent code examples for PyNaCl and in the cryptography.io “recipes" layer appear to have contributed to high rates of security success.

What do we mean by usable? Despite claims of usability and a simplified API, Keyczar proved the most difficult to use of our chosen libraries. This was caused primarily by two issues: poor documentation (as measured by our API usability scale) and the lack of documented support for key generation in code, rather than requiring interaction at the command line. Those few participants who successfully achieved functional code had very high rates of security, but in practice developers who give up on a library because they cannot make it work for the desired task will not be able to take advantage of potential security benefits. For example, developers who have difficulty with Keyczar might turn to PyCrypto, which participants preferred but which showed poor security results.

A blueprint for future libraries. Taken together, our results suggest several important considerations for designers of future cryptographic libraries. First, the recent emphasis on simplifying APIs (and choosing secure defaults) has provided improvement; we endorse continuing in this direction. We suggest, however, that library designers go further, by treating documentation quality as a first-class requirement, with particular emphasis on secure code examples. We also recommend that library designers consider a broad range of potential tasks users might need to accomplish cryptographic goals, and build support for each of them into a more comprehensive whole.

Our results suggest that supporting holistic, application-level tasks with ready-to-use APIs is the best option. That said, we acknowledge that it may be difficult or impossible to predict all tasks API users may want or need. Therefore, where lower-level features are necessary, they should be intentionally designed to make combining them into more complex tasks securely as easy as possible.

Looking forward, further research is needed to design and evaluate libraries that meet these goals. Some changes can also be made within existing libraries—for example, improving documentation, changing insecure defaults to secure defaults, or even adding compiletime or runtime warnings for insecure parameters. These changes should be evaluated involving future users both before they are deployed and longitudinally to see how they affect outcomes within real-world code. We also hope to refine and expand the usability scale developed in this chapter to create

an evaluation framework for security APIs generally, providing both feedback and guidance for improvement.

This chapter showed that studies with developers can be used to research API usability in a programming study. Our experiment helped us to create meaningful guidelines to improve the usable security of cryptographic libraries. For this, we experimented with a then-novel group of developers: those we recruited from GitHub. In a follow-up work, we explore this sample in depth. One main result of Chapters 3 and 4 was that documentation and other advice sources significantly impact security outcomes. In Chapter 6, we therefore explore online advice for secure programming.

Chapter 5

Exploring a GitHub Sample for Security Developer Studies

Disclaimer: *The contents of this chapter were previously published as part of the conference paper "Security Developer Studies with GitHub Users: Exploring a Convenience Sample", presented at the 2017 Symposium on Usable Privacy and Security. This research was conducted as a team with my co-authors Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl; this chapter therefore uses the academic "we". Christian Stransky, Michelle Mazurek, Sascha Fahl, and I designed the study. Christian Stransky, Sascha Fahl, and I conducted the study. All authors evaluated the study and co-wrote the paper. The libraries and version of Python used throughout the study reflect the landscape in 2016.*

5.1 Motivation

The usable security community is increasingly considering how to improve security decision-making not only for end users, but for information technology professionals, including system administrators and software developers [3, 4, 85, 95, 285]. By focusing on the needs and practices of these communities, we can develop guidelines and tools and even redesign ecosystems to promote secure outcomes in practice, even when administrators or developers are not security experts and must balance competing priorities.

One common approach in usable security and privacy research is to conduct an experiment, which can allow researchers to investigate causal relationships (e.g., [37, 83, 118, 253]). Other non-field-study mechanisms, such as surveys and interview studies, are also common. For research concerned with the general population of end users, recruitment for these studies can be fairly straightforward, via online recruitment platforms such as Amazon Mechanical Turk or via local methods such as posting flyers and advertising on email lists or classified-ad services like Craigslist. These approaches generally yield acceptable sample sizes at an affordable cost.

Recruiting processes for security developer studies, however, are less well established. For in-lab studies, professional developers may be hard to contact (relative to the general public), may not be locally available outside of tech-hub regions, may have demanding schedules, or may be unwilling to participate when research compensation is considerably lower than their typical hourly rate. For these reasons, studies involving developers tend to have small samples and/or to rely heavily on university computer-science students [4, 22, 126, 248, 249, 285]. To our knowledge, very few researchers have attempted large-scale online security developer studies [3, 22].

To date, however, it is not well understood how these different recruitment approaches affect research outcomes in usable security and privacy studies. The empirical software engineering community has a long tradition of conducting experiments with students instead of professional developers [226] and has found that under certain circumstances, such as similar level of expertise in the task at hand, students can be acceptable substitutes [215]. These studies, however, do not consider a security and privacy context; we argue that this matters, because security and privacy tasks differ from general programming tasks in several potentially important ways. First, because security and privacy are generally secondary tasks, it can be dangerous to assume they exhibit similar characteristics as general programming tasks. For example, relative to many general programming tasks, it can be especially difficult for a developer to directly test that security is working. (For example, how does one observe that a message is correctly encrypted?) Second, a portion of professional developers are self-taught, so their exposure to security and privacy education may differ importantly from university students' [235].

The question of how to recruit for security studies of developers in order to maximize validity is complex but important. In this study, we take a first step toward answering it: We report on an experiment (n=307) comparing GitHub contributors completing the same security-relevant tasks. For this experiment, we take as a case study the approach (which we used in prior work [3], Chapter 4) of recruiting active developers from GitHub for an online study. All participants completed three Python-programming tasks spanning four security-relevant concepts, which were manually scored for functionality and security. We found that participants across all programming experience levels were similarly inexperienced in security, and that professional developers reported more programming experience than university students. Being a professional did not increase a participant's likelihood of writing functional or secure code statistically significantly. Similarly, self-reported security background had no statistical effect on the results. Python experience was the only factor that significantly increased the likelihood of writing both functional and secure code. Further work is needed to understand how participants from GitHub compare to those recruited more traditionally (e.g., students recruited using flyers and campus e-mail lists, or developers recruited using meetup websites or researchers' corporate contacts). Nonetheless, our findings provide preliminary evidence that at least in this context, similarly experienced university students can be a valid option for studying professionals developers' security behaviors.

5.2 Related Work

We discuss related work in two key areas: user studies with software developers and IT professionals focusing on security-relevant topics, and user studies with software developers and IT professionals that do not focus on security but do discuss the impact of participants' level of professionalism on the study's validity.

Studies with Security Focus. In [4] (Chapter 3) we present a laboratory study on the impact of information sources such as online blogs, search engines, official API documentation and StackOverflow on code security. We recruited both computer science students (40) and professional Android developers (14). We found that software development experience in Android, as bucketed in the study, had no impact on code security, but previous participation in security classes had a significant impact. That study briefly compares students to professionals, finding that professionals were more likely to produce functional code but no more likely to produce

secure code; however, that work does not deeply interrogate differences between the populations and the resulting implications for validity. The study is however limited by bucketing experience years, and by the small sample size, as well as the limited representation by professional developers. In [3], we conducted an online experiment with GitHub users to compare the usability of cryptographic APIs; that work does not distinguish different groups of GitHub users.

Many studies with a security focus rely primarily on students. Yakdan et al. conducted a user study to measure the quality of decompilers for malware analysis [285]. Participants included 22 computer-science students who had completed an online bootcamp as well as 9 professional malware analysts.

Scandariato conduct a controlled experiment with 9 graduate students, all of whom had taken a security class, to investigate whether static code analysis or penetration testing was more successful for finding security vulnerabilities in code [218]. Layman et al. conducted a controlled experiment with 18 computer-science students to explore what factors are used by developers to decide whether or not to address a fault when notified by an automated fault detection tool [142].

Jain and Lindqvist conducted a laboratory study with 25 computer-science students (5 graduate; 20 undergraduate) to investigate a new, more privacy-friendly location API for Android application developers and found that, when given the choice, developers prefer the more privacy-preserving API [126]. Barik et al. conducted an eye-tracking study with undergraduate and graduate university students to investigate whether developers read and understand compiler warning messages in integrated development environments [25]. Studies that use professional developers are frequently qualitative in nature, and as such can effectively make use of relatively small sample sizes. Johnson et al. [130] conducted interviews with 20 real developers to investigate why software developers do not use static analysis tools to find bugs in software, while Xie et al. [281] conducted 15 semi-structured interviews with professional software developers to understand their perceptions and behaviors related to software security. Thomas et al. [248] conducted a laboratory study with 28 computer-science students to investigate interactive code annotations for access control vulnerabilities. As follow up, Thomas et al. [249] conducted an interview and observation-based study with professional software developers using snowball sampling. They were able to recruit 13 participants, paying each a $25 gift card, to examine how well developers understand the researchers' static code analysis tool ASIDE.

Johnson et al. [129] describe a qualitative study with 26 participants including undergraduate and graduate students as well as professional developers. Smith et al. [230] conducted an exploratory study with five students and five professional software developers to study the questions developers encounter when using static analysis tools. To investigate why developers make cryptography mistakes, Nadi et al. [166]surveyed 11 Stack Overflow posters who had asked relevant questions. A follow-up survey recruited 37 Java developers via snowball sampling, social media, and email addresses drawn from GitHub commits. This work does not address demographic differences, nor whether participants were professional software developers, students, or something else.

A few online studies of developers have reached larger samples, but generally for short surveys rather than experimental tasks. Balebako et al. [22] studied the privacy and security behaviors of smartphone application developers; they conducted 13 interviews with application developers and an online survey with 228 application developers. They compensated the interviewees with $20 each, and the online survey participants with a $5 Amazon gift card. Witschey et al. [276] survey hundreds

of developers from multiple companies (snowball sampling) and from mailing lists to learn their reasons for or against the use of security tools.

Overall, these studies suggest that reaching large numbers of professional developers can be challenging. As such, understanding the sample properties of participants who are more readily available (students, online samples, convenience samples) is an aspect of contextualizing the valuable results of these studies. In this paper, we take a first step in this direction by examining in detail an online sample from GitHub.

Studies without Security Focus. In the field of Empirical Software Engineering, the question whether or not students can be used as substitutes for developers when experimenting is of strong interest. Salman et al. [215] compared students and developers for several (non-security-related) tasks, and found that the code they write can be compared if they are equally inexperienced in the subject they are working on. When professionals are more experienced than students, their code is better across several metrics. Hoest et al. [124] compare students and developers across assessment (not coding) tasks and find that under certain conditions, e.g., that students be in the final stretches of a Master's program, students can be used as substitutes for developers. Carver et al. [45] give instructions on how to design studies that use students as coding subjects. McMeekin et al. [159] find that different experience levels between students and professionals have a strong influence on their abilities to find flaws in code. Sjoeberg et al. [226] systematically analyze a decade's worth of studies performed in Empirical Software Engineering, finding that eighty-seven percent of all subjects were students and nine percent were professionals. They question the relevance for industry of results obtained in studies based exclusively on student recruits. Smith et al. [229] perform post-hoc analysis on previously conducted surveys with developers to identify several factors software researchers can use to increase participation rates in developer studies. Murphy-Hill et al. [162] enumerate dimensions which software engineering researchers can use to generalize their findings.

5.3 Methods

We designed an online, between-subjects study to compare how effectively developers could quickly write correct, secure code using Python. We recruited participants, all with Python experience, who had published source code at GitHub.

Participants were assigned to complete a set of three short programming tasks using Python: an encryption task, a task to store login credentials in an SQLite database, and a task to write a routine for a URL shortener service. Each participant was assigned the tasks in a random order (no task depended on completing a prior task). We selected these tasks to provide a range of security-relevant operations while keeping participants' workloads manageable.

After finishing the tasks, participants completed an exit survey about the code they wrote during the study, as well as their educational background and programming experience. Two researchers coded participants' submitted code for functional correctness and security.

All study procedures were approved by the Ethics Review Board of Saarland University, the Institutional Review Board of the University of Maryland and the NIST Human Subjects Protection Office.

5.3.1 Language Selection

We elected to use Python as the programming language for our experiment, as it is widely used across many communities and offers support for all kinds of security-related APIs, including cryptography. As a bonus, Python is easy to read and write, is widely used among both beginners and experienced programmers, and is regularly taught in universities. Python is the third most popular programming language on GitHub, trailing JavaScript and Java [103]. Therefore, we reasoned that we would be able to recruit sufficient professional Python developers and computer science students for our study.

5.3.2 Recruitment

As a first step to understanding security-study behavior of GitHub committers, we recruited broadly from GitHub, the popular source-code management service. To do this, we extracted all Python projects from the GitHub Archive database [102] between GitHub's launch in April 2008 and December 2016, yielding 798,839 projects in total. We randomly sampled 100,000 of these repositories and cloned them. Using this random sample, we extracted email addresses of 80,000 randomly chosen Python committers. These committers served as a source pool for our recruitment.

We emailed these GitHub users in batches, asking them to participate in a study exploring how developers use Python. We did not mention security or privacy in the recruitment message. We mentioned that we would not be able to compensate them, but the email offered a link to learn more about the study and a link to remove the email address from any further communication about our research. Each contacted GitHub users was assigned a unique pseudonymous identifier (ID) to allow us to correlate their study participation to their GitHub statistics separately from their email address.

Recipients who clicked the link to participate in the study were directed to a landing page containing a consent form. After affirming that they were over 18, consented to the study, and were comfortable with participating in the study in English, they were introduced to the study framing. We did not restrict participation to those with security expertise because we were interested in the behavior of non-security-experts encountering security as a portion of their task.

To explore the characteristics of this sample, the exit questionnaire included questions about whether they were currently enrolled in an undergraduate or graduate university program and whether they were working in a job that mainly involved Python programming. We also asked about years of experience writing Python code, as well as whether the participant had a background in computer security.

5.3.3 Experimental Infrastructure

For this study, we used an experimental infrastructure we developed, which is described in detail in our previous work [3, 238](Chapter 4).

We designed the experimental infrastructure with certain important features in mind:

- A controlled study environment that would be the same across all participants, including having pre-installed all needed libraries.
- The ability to capture all code typed by our participants, capture all program runs and attendant error messages, measure time spent working on tasks, and recognize whether or not code was copied and pasted.

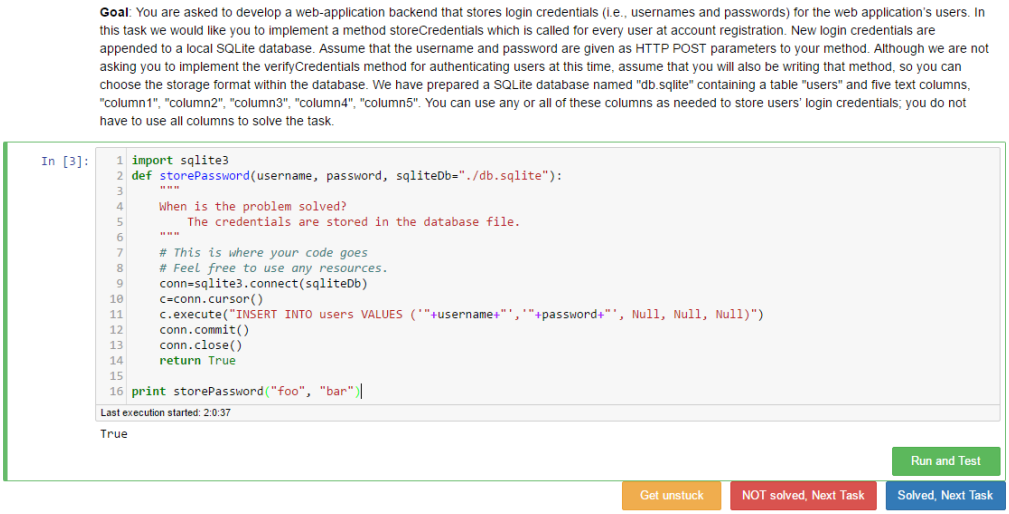

FIGURE 5.1: An example of the study's task interface.

- Allowing participants to skip tasks and continue on to the remaining tasks, while providing information on why they decided to skip the task.

To achieve these goals, the infrastructure uses Jupyter Notebooks (version 4.2.1) [133], which allow our participants to write, run, and debug their code in the browser, without having to download or upload anything. The code runs on our server, using our standardized Python environment (Python 2.7.11). This setup also allows us to frequently snapshot participants' progress and capture copy-paste events. To prevent interference between participants, each participant was assigned to a separate virtual machine running on Amazon's EC2 service. Figure 5.1 shows an example Notebook.

We pre-installed many popular Python libraries for accessing an SQLite database, dealing with string manipulation, storing user credentials, and cryptography. Table C.2 in Appendix C.3 lists all libraries we provided. We tried to include as many relevant libraries as possible, so that every participant could work on the tasks using their favorite libraries.

The tasks were shown one at a time, with a progress indicator showing how many tasks remained. For each task, participants were given buttons to "Run and test" their code, and to move on using "Solved, next task" or "Not solved, but next task." A "Get unstuck" button was also provided in case the participant accidentally sent Python into an infinite loop or otherwise crashed the Python interpreter running in the Notebook. After completing (or skipping) all tasks, the participant was redirected to the exit survey.

5.3.4 Exit Survey

Once all tasks had been completed or abandoned, the participants were directed to a short exit survey (cf. Appendix C.1). We asked for their opinions about the tasks they had completed: Did they think they had solved them? How did they perceive the tasks' difficulty? Did they think their solution was secure? We also were interested in whether they thought about security or privacy when working on the tasks. Finally, we wanted to know whether our participants had worked on similar programming problems in the past. For these task-specific questions, we used our infrastructure to display the participant's code for the corresponding task

for their reference. We also asked several questions about demographic information and programming experience, to allow us to distinguish categories of participants.

5.3.5 Task Design

We designed tasks that were short enough so that the uncompensated participants would be likely to complete them before losing interest, but still complex enough to be interesting and allow for some mistakes. Most importantly, we designed the tasks to model real-world security and privacy problems that Python developers could reasonably be expected to encounter. While these tasks of course do not represent all possible security tasks, we think they provide an interesting variety for analysis.

URL Shortener We asked our participants to write code for a method that could be part of a URL shortening service such as bit.ly [1] or Google's URL shortener goo.gl. [2]

> **URL Shortener Task:**
> You are asked to develop code for a URL shortening service similar to `https://goo.gl/`. Users of this service will provide you URLs such as `https://en.wikipedia.org/wiki/History_of_the_Internet`. As a result your service should return a shortened URL such as `http://foo.bar/1xf2`. In this task we would like you to implement a method `shortenURL` that is called for every input URL. The output of this method is a shortened URL for the input URL.
>
> ***When is the problem solved?***
> A shortened URL is returned. Please print the output of the method to the console.

While this task does not directly have security implications, we were mainly interested in whether, e.g., they used well established approaches such as message digest or random number generators to generate a short URL, or whether they invented their own algorithm.

[1] cf. `https://bitly.com/`
[2] cf. `https://goo.gl/`

Credential Storage We asked our participants to store login credentials, namely a username and password, in a database backend.

> **Credential Storage Task:**
> You are asked to develop a web-application backend that stores login credentials (i.e., usernames and passwords) for the web application's users. In this task, we would like you to implement a method `storeCredentials` that is called for every user at account registration. New login credentials are appended to a local SQLite database. Assume that the username and password are given as HTTP POST parameters to your method. Although we are not asking you to implement the `verifyCredentials` method for authenticating users at this time, assume that you will also be writing that method, so you can choose the storage format within the database. We have prepared a SQLite database named "db.sqlite" containing a table "users" and five text columns, "column1", "column2", "column3", "column4", "column5". You can use any or all of these columns as needed to store users' login credentials; you do not have to use all columns to solve the task.
>
> ***When is the problem solved?***
> The credentials are stored in the database file.

This task has direct security implications: we were mainly interested in whether participants followed security best practices. Best practices to store user credentials in a database include hashing and salting the password instead of storing it in plaintext and using some sort of input sanitization (e.g. parameterized instead of raw SQL queries) for SQL queries to prevent SQL injection attacks.

String Encryption We asked participants to write code to encrypt and decrypt a string.

> **String Encryption Task:**
> You are asked to write code that is able to encrypt and decrypt a string.
>
> ***When is the problem solved?***
> The input string is encrypted and decrypted afterwards. You should see the encrypted and decrypted string in the console.

In this task we were mainly interested in whether participants wrote secure cryptographic code, e.g., choosing secure algorithms, strong key sizes, and secure modes of operation.

For each task, we provided stub code and some comments with instructions about how to work on the task. The code stubs were intended to make the programming task as clear as possible and to ensure that we would later easily be able to run automated unit tests to examine functionality. The code stubs also helped to orient participants to the tasks.

We told participants that "you are welcome to use any resources you normally would" (such as documentation or programming websites) to work on the tasks. We asked participants to note any such resources as comments to the code, for our reference, prompting them to do so when we detected that they had pasted text and/or code into the Notebook.

5.3.6 Evaluating Participant Solutions

We used the code submitted by our participants for each task, henceforth called a *solution*, as the basis for our analysis. We evaluated each participant's solution to each task for both functional correctness and security. Every task was independently reviewed by two coders, using a content analysis approach [140] with a codebook based on our knowledge of the tasks and best practices. Differences between the two coders were resolved by discussion. We briefly describe the codebook below.

Functionality. For each programming task, we assigned a participant a functionality score of 1 if the code ran without errors, passed the unit tests and completed the assigned task, or 0 if not.

Security. We assigned security scores only to those solutions which were graded as functional. To determine a security score, we considered several different security parameters. A participant's solution was marked secure (1) only if their solution was acceptable for every parameter; an error in any parameter resulted in a security score of 0.

URL Shortener For the URL shortening task, we checked how participants generated a short URL for a given long URL. We were mainly interested in whether participants relied on well-established mechanisms such as message digest algorithms (e.g. the SHA1 or SHA2 family) or random number generators, or if they implemented their own algorithms. The idea behind this evaluation criterion is the general recommendation to rely on well-established solutions instead of reinventing the wheel. While adhering to this best practice is advisable in software development in general, it is particularly crucial for writing security- or privacy-relevant code (e.g., use established implementations of cryptographic algorithms instead of re-implementing them from scratch). We also considered the reversibility of the short URL as a security parameter (reversible was considered insecure). We did not incorporate whether solutions were likely to produce collisions (i. e. produce the same short URL for different input URLs) or the space of the URL-shortening algorithm (i. e. how many long URLs the solution could deal with) as security parameters: we felt that given the limited time frame, asking for an optimal solution here was asking too much.

Credential Storage For the credential storage task, we split the security score in two. One score (password storage) considered how participants stored users' passwords. Here, we were mainly interested whether our participants followed security best practices for storing passwords. Hence, we scored the plain text storage of a password as insecure. Additionally, applying a simple hash algorithm such as MD5, SHA1 or SHA2 was considered insecure, since those solutions are vulnerable to rainbow table attacks. Secure solutions were expected to use a salt in combination with a hash function; however, the salt needed to be random (but not necessarily secret) for each password to withstand rainbow table attacks. Therefore, using the same salt for every password was considered insecure. We also considered the correct use of HMACs [139] and PBKDF [132] as secure.

The second security score (SQL input) considered how participants interacted with the SQLite database we provided. For this evaluation, we were mainly interested whether the code was vulnerable to SQL injection attacks. We scored code

that used raw SQL queries without further input sanitization as insecure, while we considered using prepared statements secure.[3]

String Encryption For string encryption, we checked the selected algorithm, key size and proper source of randomness for the key material, initialization vector and, if applicable, mode of operation. For symmetric encryption, we considered ARC2, ARC4, Blowfish, (3)DES and XOR as insecure and AES as secure. We considered ECB as an insecure mode of operation and scored Cipher Block Chaining (CBC), Counter Mode (CTR) and Cipher Feedback (CFB) as secure. For symmetric key size, we considered 128 and 256 bits as secure, while 64 or 32 bits were considered insecure. Static, zero or empty initialization vectors were considered insecure. For asymmetric encryption, we considered the use of OAEP/PKCS1 for padding as secure. For asymmetric encryption using RSA, we scored keys larger than or equal to 2048 bits as secure.

5.3.7 Limitations

As with any user study, our results should be interpreted within the context of our limitations.

Choosing an online rather than an in-person laboratory study allowed us less control over the study environment and the participants' behavior. However, it allowed us to recruit a diverse set of developers we would not have been able to obtain for an in-person study.

Recruiting using conventional recruitment strategies, such as posts at university campuses, on Craigslist, in software development forums or in particular companies would likely have limited the number and variety of our participants. As a result, we limited ourselves to active GitHub users. We believe that this resulted in a reasonably diverse sample, but of course GitHub users are not necessarily representative of developers more broadly, and in particular students and professionals who are active on GitHub may not be representative of students and professionals overall. The small response rate compared to the large number of developers invited also suggests a strong opt-in bias. Comparing the set of invited GitHub users to the valid participants suggests that more active GitHub users were more likely to participate, potentially widening this gap. As a result, our results may not generalize beyond the GitHub sample. However, all the above limitations apply equally across different properties of our participants, suggesting that comparisons between the groups are valid.

Because we could not rely on a general recruitment service such as Amazon's Mechanical Turk, managing online payment to developers would have been very challenging; further, we would not have been able to pay at an hourly rate commensurate with typical developer salaries. As a result, we did not offer our participants compensation, instead asking them to generously donate their time for our research.

We took great care to email each potential participant only once, to provide an option for an email address to opt out of receiving any future communication from us, and to respond promptly to comments, questions, or complaints from potential participants. Nonetheless, we did receive a small number of complaints from people who were upset about receiving unsolicited email.[4]

[3] While participants could have manually sanitized their SQL queries, we did not find a single solution that did that.

[4] Overall, we received 13 complaints.

Some participants may not provide full effort or many answer haphazardly; this is a particular risk of all online studies. Because we did not offer any compensation, we expect that few participants would be motivated to attempt to “cheat" the study rather than simply dropping out if they were uninterested or did not have time to participate fully. We screened all results and attempted to remove any obviously low-quality results (e.g., those where the participant wrote negative comments in lieu of real code) before analysis, but cannot discriminate with perfect accuracy. Further, our infrastructure based on Jupyter Notebooks allowed us to control, to an extent, the environment used by participants; however, some participants might have performed better had we allowed them to use the tools and environments they typically prefer. However, these limitations are also expected to apply across all participants.

5.4 Results

We were primarily interested in comparing the performances of different categories of participants in terms of functional and secure solutions. Overall, we found that students and professionals report differences in experience (as might be expected), but we did not find significant differences between them in terms of solving our tasks functionally or securely.

5.4.1 Statistical Testing

In the following subsections, we analyze our results using regression models as well as non-parametric statistical testing. For non-regression tests, we primarily use the Mann-Whitney-U test (MWU) to compare two groups with numeric outcomes, and X^2 tests of independence to compare categorical outcomes. When expected values per field are too small, we use Fisher’s exact test instead of X^2.

Here, we explain the regression models in more detail. The results we are interested in have binary outcomes; therefore, we use *logistic* regression models to analyze those results. the consideration whether an insecure task counts as *dangerous*, i.e. whether it is functional, insecure and the programmer thinks it is secure, is also binary and therefore analyzed analogously. As we consider results on a per-task basis, we use a mixed model with a random intercept; this accounts for multiple measures per participant. For the regression analyses, we select among a set of candidate models with respect to the Akaike Information Criterion (AIC) [43]. All candidate models include which task is being considered, as well as the random intercept, along with combinations of optional factors including years of Python experience, student and professional status, whether or not the participant reported having a security background, and interaction effects among these various factors. These factors are summarized in Table 5.1. For all regressions, we selected as final the model with the lowest AIC.

The regression outcomes are reported in tables; each row measures change in the dependent variable (functionality, security, or security perception) related to changing from the *baseline* value for a given factor to a different value for the same factor (e.g., changing from the encryption task to the URL shortening task). The regressions output odds ratios (O.R.) that report on change in likelihood of the targeted outcome. By construction, O.R.=1 for baseline values. For example, Table 5.2 indicates that the URL shortening task was $0.45\times$ as likely to be functional as the baseline string encryption task. In each row, we also report a 95% confidence interval (C.I.)

and a p-value; statistical significance is assumed for p$\leq$.05, which we indicate with an asterisk (*). For both regressions, we set the encryption task to be the baseline, as it was used similarly in previous work [3].

5.4.2 Participants

We sent 23,661 email invitations in total. Of these, 3,890 (16.4%) bounced and another 447 (1.9%) invitees requested to be removed from our list, a request we honored. 16 invitees tried to reach the study but failed due to technical problems in our infrastructure, either because of a large-scale Amazon outage[5] during collection or because our AWS pool was exhausted during times of high demand.

A total of 825 people agreed to our consent form; 93 (11.3%) dropped out without taking any action, we assume because the study seemed too time-consuming. The remaining 732 participants clicked on the begin button after a short introduction; of these, 440 (60.1%) completed at least one task and 360 of those (81.8%) proceeded to the exit survey. A total of 315 participants completed all programming tasks and the exit survey. We excluded eight for providing obviously invalid results. From now on, unless otherwise specified, we report results for the remaining 307 valid participants, who completed all tasks and the exit survey.

We classified these 307 participants into students and professionals according to their self-reported data. If a participant reported that they work at a job that mainly requires writing code, we classified them as a professional. If a participant reported being an undergraduate or graduate student, we classified them as a student. It was possible to be classified as either only a professional, only a student, both, or neither. The 307 valid participants includes 254 total professionals, 25 undergraduates, and 49 graduate students. 53 participants were both students and professionals; 32 participants were neither students nor developers. Due to the small sample size, we treated undergraduates and graduate students as one group for further analysis.

The 307 valid participants reported ages between 18 and 81 years (mean: 31.6; sd: 7.7) [Student: 19-37, mean: 25.3, sd: 5.2 - Professional: 18-54, mean: 32.9, sd: 6.7], and most of them reported being male (296 - Student: 21 - Professional 194). All but one of our participants (306) had been programming in general for more than two years and 277 (Student: 18, Professional: 186) had been programming in Python for more than two years. The majority (288 - Student: 20, Professional: 188) said they had no IT-security background nor had taken any security classes.

We compared students to non-students and professionals to non-professionals for security background and years of Python experience. (We compared them separately because some participants are both students and professionals, or are neither.) In both cases, there was no difference in security background (due to small cell counts, we used Fisher's exact test; both with $p \approx 1$). Professionals had significantly more experience in Python than non-professionals, with a median 7 years of experience compared to 5 (MWU, $W = 5040$, $p = 0.004$). Students reported significantly less experience than non-students, with median 5 years compared to 7 years (MWU, $W = 10963$, $p < 0.001$).

The people we invited represent a random sample of GitHub users — however, our participants are a small, self-selected subset of those. We were able to retrieve metadata for 192 participants; for the remainder, GitHub returned a 404 error, which most likely means that the account was deleted or set to private after the commit

[5]Some participants were affected by this Amazon EC2 outage: `https://www.recode.net/2017/3/2/14792636/amazon-aws-internet-outage-cause-human-error-incorrect-command`.

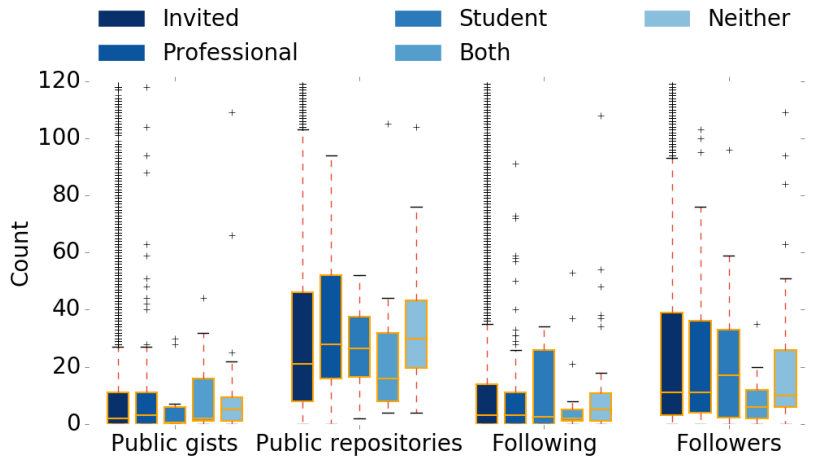

FIGURE 5.2: Boxplots comparing our invited participants (a random sample from GitHub) with those who provided valid participation. The center line indicates the median; the boxes indicate the first and third quartiles. The whiskers extend to ±1.5 times the interquartile range. Outliers greater than 150 were truncated for space.

we crawled was pushed to GitHub. We compare these 192 participants to the 12117 invited participants for whom we were able to obtain GitHub metadata.

Figure 5.2 illustrates GitHub statistics for all groups (for more detail, see Table C.1 in the Appendix). Our participants are slightly more active than the average GitHub user: They have a median of 3 public gists compared to 2 for invited GitHub committers (MWU, W = 1045300, p = 0.01305); they have a median of 28 public repositories compared to 21 for invited participants (MWU, W = 1001200, p < 0.001); they all follow a median of 3 committers (MWU, W = 1142100, p = 0.66); and they are followed by a similar number of committers (10 for participants, 11 for invited; MWU, W = 1146100, p = 0.73).

5.4.3 Functionality

We evaluated the functionality of the code our participants wrote while working on the programming tasks. Figure 5.3 illustrates the distribution of functionally correct solutions between tasks and across professional developers and university students. Overall, professionals got 720 of 804 tasks correct (89.6%), while students got 71 of 84 correct (84.5%); participants who were both students and professionals got 181 of 212 (85.4%) correct, while participants who were neither succeeded in 114 of 128 (89.1%) cases.

Table 5.2 shows the results of the regression model for functionality. The final model does not include developer or student status, security background, or any interaction effects, suggesting that these factors are not important predictors of functional success. Python experience, on the other hand, did produce a statistically significant effect: each additional year of experience corresponds to on average a 10% increase in likelihood of a correct solution. Comparing tasks, the password storage task proved most difficult: participants were only 0.45× as likely to complete it as to complete the baseline string encryption task. Results for the URL shortening task were comparable to the baseline.

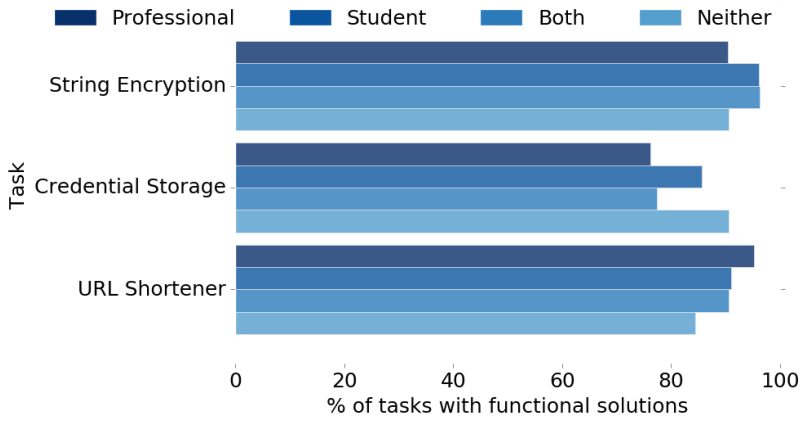

FIGURE 5.3: Functionality results per task, split by students vs. professional developers.

To assess the fit of our regression model, we use Nagelkerke's method [167] to compute a pseudo-R^2 value, somewhat analogous to the standard coefficient of determination commonly used with ordinary linear regression. We find that, relative to a null model that includes only the random (per-participant) effect, our model produces a pseudo-R^2 of 0.07; this is not a particularly strong fit, reflecting the fact that there are potentially many unmeasured covariates, such as the specifics of a participant's prior programming experience and education.

5.4.4 Security

We evaluated the security of the code based on the codebook described in Section 5.3.6. In this section, we talk about four tasks instead of three, as the credentials storage task had two security relevant components that we account for individually: secure password digest and SQL input validation (see Section 5.3.6 for details).

Figure 5.4 illustrates the distribution of secure solutions between tasks and across professional developers vs. university students. Altogether, professionals got 493 of 720 tasks correct (68.5%), while students got 48 of 71 correct (67.6%); participants who were both students and professionals got 119 of 181 (65.7%) correct, while participants who were neither succeeded in 77 of 114 (67.5%) cases.

Table 5.3 lists the results of the final security regression model. This model had Nagelkerke pseudo-R^2 of 0.183, which is a fairly strong fit for an uncontrolled experiment with potential unmeasured factors.

As with the functionality results, none of developer status, student status, security background, nor any interactions, appear in the final model. This again suggests that these factors do not meaningfully predict security success. As before, more Python experience is associated with more success: this time, each year of additional experience adds about 5% to the likelihood of a secure solution. Comparing tasks, string encryption proved significantly more difficult to complete securely than any other task. Password storage was associated with about 2× higher likelihood of success. Both these tasks were significantly harder than SQL input validation and URL shortening. (The non-overlapping confidence intervals indicate significant difference from password storage as well as from the baseline string encryption task). SQL input validation and URL shortening were each about 8× easier to secure than string encryption.

Factor	Description	Baseline
Required		
Task	The performed tasks	String encryption
Participant	Random effect accounting for repeated measures	n/a
Optional		
Python experience	Python programming experience in years, self-reported.	n/a
Security background	True or false, self-reported.	False
Developer	True or false, self-reported.	False
Student	True or false, self-reported.	False
Python experience × task		False:String encryption
Python experience × developer		False:False
Python experience × student		False:False
Developer × task		False:String encryption
Student × task		False:String encryption

TABLE 5.1: Factors used in regression models. Categorical factors are individually compared to the baseline. Final models were selected by minimum AIC; candidates were defined using all possible combinations of optional factors, with the required factors included in every candidate.

Factor	O.R.	C.I.	p-value
URL shortener	0.45	[0.22, 0.89]	0.022*
Credentials storage	0.22	[0.11, 0.42]	<0.001*
Python experience	1.10	[1.02, 1.19]	0.014*

TABLE 5.2: Results of the final logistic regression model examining functionality of tasks for participants. Odds ratios (O.R.) indicate relative likelihood of succeeding. Statistically significant factors indicated with *. See Table 5.1 for further details.

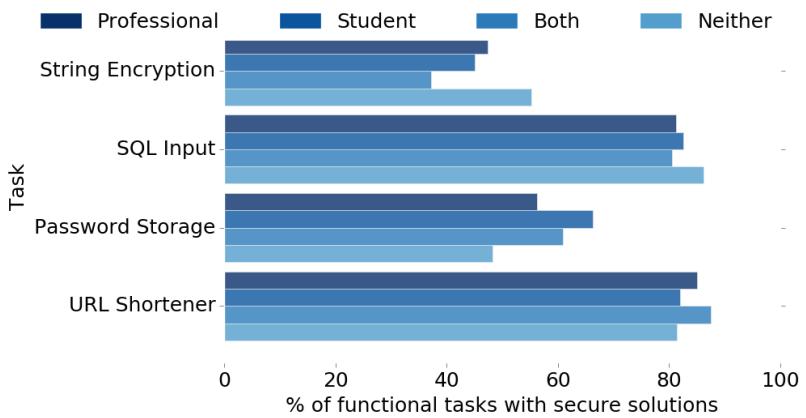

FIGURE 5.4: Security results per task, split by students vs. professional developers.

Factor	O.R.	C.I.	p-value
URL shortener	8.03	[5.14, 12.53]	<0.001*
Password storage	2.34	[1.6, 3.43]	<0.001*
SQL input	7.69	[4.89, 12.09]	<0.001*
Python experience	1.05	[1.01, 1.1]	0.020*

TABLE 5.3: Results of the final logistic regression model examining security of tasks for participants. Odds ratios (O.R.) indicate relative likelihood of succeeding. Statistically significant factors indicated with *. See Table 5.1 for further details.

Category	Encryption	Password Storage	URL shortener	SQL input	Total
Dangerous (Perception Secure & Scoring Insecure)	41 (13.4%)	57 (18.6%)	17 (5.5%)	39 (12.7%)	154
Harmless Misperception (Perception Insecure & Scoring Secure)	49 (16.0%)	31 (10.1%)	156 (50.8%)	64 (20.8%)	300
True Positives (Perception Secure & Scoring Secure)	82 (26.7%)	131 (42.7%)	75 (24.4%)	149 (48.5%)	437
True Negatives (Perception Insecure & Scoring Insecure)	135 (44.0%)	88 (28.7%)	59 (19.2%)	55 (17.9%)	337

TABLE 5.4: Detailed distribution of perceived and actual security within functional solutions, broken out per task. Percentages are as a function of each task; for example, 13.4% of all encryption solutions were categorized as dangerous.

Security Perception

We asked participants, for each task, whether they believed their result was secure. In this section, we analyze the incidence of what we call *dangerous* solutions: solutions that are functionally correct and where the participant believes the result is secure, but our analysis indicates that it is not. In a sense, this represents a worst-case scenario, where a developer may confidently release insecure code unwittingly.

Table 5.4 details how perceptions of security connect to evaluated security. Across tasks, 154 of 1228 (12.5%) solutions were classified as dangerous; dangerous solutions were least common of the four classes, but this rate is still higher than we might hope.

Table 5.5 reports on a regression model with whether or not a solution is classified as dangerous as the binary outcome. The final model contains no optional factors at all. This indicates that none of Python experience, security background, professional status, or student status is a good predictor of a dangerous outcome. Indeed, the Nagelkerke pseudo-R^2 for this model is only 0.049, which reflects that we did not measure important additional factors.

Our regression model suggests that string encryption, which was most difficult to secure, was (unsurprisingly) also associated with significantly higher likelihood of dangerous solutions than the SQL input and URL shortening tasks. Encryption, however, was comparable to password digests, which also have a cryptographic component. In a prior experiment, we found that about 20% of cryptographic tasks fell into this dangerous category [4].

Factor	O.R.	C.I.	p.value
URL shortener	0.25	[0.12, 0.52]	<0.001*
Password storage	1.16	[0.7, 1.93]	0.565
SQL input	0.53	[0.29, 0.97]	0.038*

TABLE 5.5: Results of the final logistic regression model examining perceived security and actual security. Odds ratios (O.R.) indicate relative likelihood of being insecure. Statistically significant factors indicated with *. See Table 5.1 for further details.

Investigating Security Errors

We also examined patterns in the types of security errors made by our participants across tasks. Note that these patterns reflect only functional but insecure solutions. In all cases, the same solution may have more than one security error, so percentages generally total to more than 100%.

	Plain password	MD5 hash	SHA1 hash	No salt	Static salt	Raw SQL	Not stored
Professionals	24 (14.0%)	3 (1.7%)	4 (2.3%)	40 (23.3%)	15 (8.7%)	29 (16.9%)	1 (0.6%)
Student	4 (25.0%)	0 (0.0%)	0 (0.0%)	6 (37.5%)	1 (6.2%)	3 (18.8%)	0 (0.0%)
Both	8 (19.5%)	2 (4.9%)	2 (4.9%)	14 (34.1%)	2 (4.9%)	8 (19.5%)	0 (0.0%)
Neither	9 (31.0%)	2 (6.9%)	0 (0.0%)	14 (48.3%)	1 (3.4%)	4 (13.8%)	0 (0.0%)
Total	45 (17.4%)	7 (2.7%)	6 (2.3%)	74 (28.7%)	19 (7.4%)	44 (17.1%)	1 (0.4%)

TABLE 5.6: Types of security errors found in functional solutions (and their percentages) by professional, student, both or neither for the password storage task. See Subsection 5.3.5 for task details and Subsection 5.3.6 for codebook details.

URL Shortening First, we consider the URL shortening task. The most common security error (11 cases, 23.0%) was participants who implemented their URL shortening feature using an algorithm that allows an attacker to easily predict the long URL for a given short URL. An example is the use of Base 64 to derive a "short" URL from a given long URL. Although we did not consider keyspace as a security parameter, we briefly review the keyspace generated by participants with functional solutions to this task. 104 participants (37.4%) selected a shortening approach with an unlimited keyspace. The remaining 174 solutions had an average keyspace of 74.1 bits (median 48, standard deviation 6.1). The average for professionals (82.0 bits, median 48) was higher than for students (62.5 bits, median 48), participants who were both students and professionals (58.5 bits, median 36) and participants who were neither (60.6 bits, median 36).

Password Storage Next, we consider insecure password storage. Here the most common error was hashing the password without using a proper salt, leaving the stored password vulnerable to rainbow-table attacks (74 cases, 77.1%). The second most common error was storing the plain password (45 cases, 46.9%). A total of 19 (19.8%) participants used a static salt instead of a random salt. Seven (7.3%) participants used MD5, while six (6.3%) used SHA-1 family hashes. Instead of using a one way hash function, four (4.2%) used encryption to secure the password. This is highly discouraged, since an attacker who can gain access to the decryption key is able to recover plain text passwords. These results are detailed in Table 5.6.

Library	Used	Weak Algo	Weak Mode	Static IV
Professionals				
No library	44 (22.8%)	42 (21.8%)	0 (0.0%)	0 (0.0%)
cryptography.io	71 (36.8%)	0 (0.0%)	0 (0.0%)	10 (5.2%)
pyCrypto	65 (33.7%)	5 (2.6%)	9 (4.7%)	23 (11.9%)
PyNaCl	10 (5.2%)	0 (0.0%)	0 (0.0%)	1 (0.5%)
Other	3 (1.6%)	3 (1.6%)	0 (0.0%)	0 (0.0%)
Student				
No library	8 (42.1%)	8 (42.1%)	0 (0.0%)	0 (0.0%)
cryptography.io	5 (26.3%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
pyCrypto	6 (31.6%)	1 (5.3%)	0 (0.0%)	1 (5.3%)
Both				
No library	17 (33.3%)	17 (33.3%)	0 (0.0%)	0 (0.0%)
cryptography.io	16 (31.4%)	0 (0.0%)	0 (0.0%)	3 (5.9%)
pyCrypto	15 (29.4%)	1 (2.0%)	2 (3.9%)	5 (9.8%)
PyNaCl	1 (2.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
pySodium	1 (2.0%)	1 (2.0%)	0 (0.0%)	0 (0.0%)
Other	1 (2.0%)	1 (2.0%)	0 (0.0%)	0 (0.0%)
Neither				
No library	6 (20.7%)	6 (20.7%)	0 (0.0%)	0 (0.0%)
cryptography.io	11 (37.9%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
pyCrypto	7 (24.1%)	0 (0.0%)	0 (0.0%)	2 (6.9%)
PyNaCl	4 (13.8%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
Other	1 (3.4%)	1 (3.4%)	0 (0.0%)	0 (0.0%)
Total				
No library	75 (25.7%)	73 (25.0%)	0 (0.0%)	0 (0.0%)
cryptography.io	103 (35.3%)	0 (0.0%)	0 (0.0%)	13 (4.5%)
pyCrypto	93 (31.8%)	7 (2.4%)	11 (3.8%)	31 (10.6%)
PyNaCl	15 (5.1%)	0 (0.0%)	0 (0.0%)	1 (0.3%)
pySodium	1 (0.3%)	1 (0.3%)	0 (0.0%)	0 (0.0%)
Other	5 (1.7%)	5 (1.7%)	0 (0.0%)	0 (0.0%)

TABLE 5.7: Types of security errors found in functional solutions (and their percentages) by professional, student, both or neither for the string encryption task. Participant categories are subdivided by the cryptographic library they opted to use. See Subsection 5.3.5 for task details and Subsection 5.3.6 for codebook details.

SQL Query For the SQL query task, 44 (97.8%) of the participants used raw SQL queries instead of prepared statements, leaving their implementation vulnerable to SQL injection attacks. Interestingly, no participant tried to implement their own SQL query sanitization solution.

Encryption For the string encryption task, one important decision participants made was the choice of cryptographic library (cf. Table C.2 for the libraries that came pre-installed). 118 (40.4% of all functional solutions) of the participants used a cryptographic library that was designed with usability in mind, which reduces the necessity to select (and potential make an error with) parameters like algorithm, mode of operation, key size, initialization vector, and padding scheme (cryptography.io: 103, PyNacl: 15, PySodium: 1). 93 participants (31.8% of all functional solutions) chose a more conventional library (PyCrypto: 93), and 73 (25.0% of all functional solutions) used no third-party library at all.

Overall, 15 (12.7%) of participants who applied a usable library made a security error, while 49 (52.7%) of the participants who used a conventional library made a security error. All participants but one who used usable libraries used secure algorithms, modes of operation, and key sizes; the other 14 who made an error used a

```
def encrypt(plainText):
    return ''.join([chr(ord(c) + n % 5) for n, c in enumerate(plainText)])

def decrypt(cipherText):
    return ''.join([chr(ord(c) - n % 5) for n, c in enumerate(cipherText)
    ])

stringToEncrypt = "ThisIsAnExample"
encryptedString = encrypt(stringToEncrypt)
print encryptedString
decryptedString = decrypt(encryptedString)
print decryptedString
```

LISTING 5.1: Substitution cipher solution as written by a professional developer participant.

static initialization vector. Users of conventional cryptographic libraries mostly also used a static initialization vector (31 cases, 63.3% of error cases), used an insecure mode of operation (11, 22.4% of error cases), or chose an insecure algorithm (7, 14.3% of error cases). These results indicate that usable libraries do reduce errors, and they are in line with the errors we identified in a prior experiment [3] (Chapter 4). These results are detailed in Table 5.7.

Among participants who did not apply cryptography effectively, 20 used Base64 to encode their plaintext instead of encrypting it, and 43 implemented a very basic substitution cipher like Rot13. An example is shown in Listing 5.1.

5.5 Discussion

In our online quasi-experiment with 307 GitHub participants, we measured functionality and security outcomes across Python programming tasks. We came into the experiment hypothesizing that whether or not a participant wrote code professionally or as a student would impact at least the functional correctness of their code. However, we found that neither student nor professional status (self-reported) was a significant factor for functionality, security, or security perception. We were also surprised to learn that self-reported security background was equally unimportant. Note that only small numbers of participants reported that they were exclusively students or that they had a security background, which may affect these results.

We did, however, find a significant effect for Python experience: Each year of experience corresponded to 10% more likelihood of getting a functional result and a 5% better chance of getting a secure result. Differences in experience across students and professionals were significant: Students reported a median of 5 years of experience, compared to 7 for professionals. On the other hand, experience did not appear to matter for security perception. This accords well with previous results within the empirical software engineering community (cf. Section 5.4), which suggest that student and professional developer participants' expertise should be similar to produce similar results. While expertise with Python in our study differs significantly between students and professional developers, their security and privacy expertise is similar (in both cases quite low). At least within GitHub then, it seems that students and professionals can be equally useful for studying usable security and privacy problems, particularly if overall experience is controlled for.

In addition to the small sample size, we speculate that the very similar results across students and professional developers can be accounted for in part because

writing security-related code is not something the average software developer deals with on a regular basis, nor is security education a strong focus at many universities teaching computer science. We hypothesize, therefore, that overall the result (that experience matters somewhat, but professional status on its own does not) would continue to hold for student and professional populations recruited more traditionally, at local universities and via professional networks. We suspect, however, that typically local university students may have less experience than students recruited from GitHub. Further research is needed to validate these hypotheses.

We found the recruitment strategy of emailing GitHub developers to be convenient in many ways: We were able to recruit many experienced professionals quickly and at a low cost. In addition, many participants expressed to us how much they enjoyed the challenge of our tasks and the opportunity to contribute to our research. However, it does have important drawbacks: we received complaints about unsolicited email from 13 invited GitHub committers and were generally subject to a small opt-in rate. We also found that our participants were slightly more active and therefore not quite representative of the GitHub population; representativeness for professionals (or students) in general is considerably less likely. Overall, the practice of sending unsolicited emails was not ideal, and is unlikely to be sustainable over many future studies. Instead, we plan in the future to develop a GitHub application that would allow developers who are interested in contributing to research to opt in to study recruitment requests, which would benefit both these developers and the research community.

This chapter addressed the methodological challenge of recruiting developers to participate in programming studies. While Chapter 2 identified challenged for developers through a systematic review, problems with resources can be found in developer studies. The next chapter collects information about online advice sources on secure development that we could review independently from recruiting developers into our research.

Chapter 6

A Survey of Security Advice for Software Developers

Disclaimer: *The contents of this chapter were previously published as part of the conference paper "Developers need support, too: A survey of security advice for software developers", presented at the 2017 IEEE Cybersecurity Development Conference. This research was conducted as a team with my co-authors Christian Stransky, Dominik Wermke, Charles Weir, Michelle L. Mazurek, and Sascha Fahl; this chapter therefore uses the academic "we". Christian Stransky, Dominik Wermke, Michelle L. Mazurek, Sascha Fahl and I designed the study. Christian Stransky, Dominik Wermke and Charles Weir conducted the coding. All authors evaluated and co-wrote the paper. The resources gathered and evaluated in the study reflect the landscape in 2017.*

6.1 Motivation

The last two decades have seen explosive growth in the creation and usage of software, as we now use computers and digital devices to manage almost every aspect of our lives: to communicate, to plan, to manage our finances, to do our shopping, and to remember all our security information. Software holds sensitive information about us, controls our financial transactions, enables our personal communication and social networking, and holds the intimate details of our lives. This growth has led to a commensurate rise in the number of people working as software developers. In 1997, the Bureau of Labor Statistics estimated just over 500,000 computer programmers in the United States; by 2016, this category had been revised into four sub-categories and more than tripled, to 1.6 million employees [42].

The increasing ubiquity of software has also led to the increasing ubiquity of security bugs and associated attacks [203]. An important question, therefore, is how to help the increasing population of developers adopt effective security practices and write more secure code.[1]

Balebako et al. surveyed and interviewed more than 200 app developers and concluded most approached security issues using web search, or by consulting peers [23]. A survey by many of the current authors concluded the same, and also determined experimentally the surprising result that programmers using digital books achieved better security than those using web search [4] (Chapter 3). Nadi et al. found that Java developers perceived cryptography APIs as too low-level and preferred more task-based solutions in documentations [166]. Further work from several of the current authors concluded that documentation is critical to security outcomes when developers use unfamiliar cryptography APIs [3] (Chapter 4).

[1] For simplicity, throughout this paper we refer to *security and privacy* as *security*.

When developers search the internet for guidance, some of the most popular results will often point to Stack Overflow.[2] Oftentimes, the developer's search will lead to a code snippet on Stack Overflow, and the developer can be tempted to copy and paste the snippet into their own code. This behaviour has been shown to often lead to operational but insecure code, widespread across hundreds of thousands of apps [3, 4, 81, 85, 95, 281]. As Stack Overflow's effect on code security has been investigated in depth, we focus our analysis elsewhere. Similarly, code completion systems found in IDEs often are not evaluated for the evolving context found in real-world situations [192].

Beyond crowdsourcing application-specific problems and documenting particular APIs, the web also contains many general resources about security and secure programming. While we would hope that these resources are helpful, to our knowledge few if any have been empirically tested for effectiveness. Moreover, the continuing prevalence of security bugs suggests that this guidance ecosystem is fundamentally broken: either effective guidance is not available (if it is even possible), or it is not reaching the developers who need it. We speculate that this web-based guidance is particularly important for developers working outside of large mainstream corporations, who do not have access to professional security teams or well-developed toolchains and frameworks supporting secure programming (e.g., Google's Tricorder [214]).

In this paper, we take a first step toward understanding and improving this guidance ecosystem. We have identified and analyzed 19 guidance websites, to understand what currently available guidance says; what it does and does not cover; and how it is structured. We found that, overall, this general-purpose guidance does not often provide concrete examples, tutorials, or exercises to help developers practice the concepts being described. We also found that while some critical topics like secure networking, user input validation, and software testing are well represented, other important concepts like program analysis tools, data minimization, and social engineering are rarely mentioned. These results provide a foundation for future research to fill gaps as well as to empirically evaluate existing guidance.

6.2 Selecting Online Resources

We collected online developer resources relevant to security by searching for variations of "developer," "security" and "guide" on various online search engines. This gives us a collection of links to online resources, some of them collections of other online resources, which we exclude from our analysis in favour of directly investigating the linked resources themselves.

We found 19 sites designed to help developers with security related questions. We excluded forum posts and Q&A platforms such as Stack Overflow, and included guides such as blog posts from authoritative sources and official guidelines from software providers and non-profit organizations such as OWASP.[3]

We focused on those types of guidelines since they carry authority for developers who feel the need to look up security-related questions and concepts. The guidelines we found covered mobile application security (Android, iOS and Windows Mobile), web security, and general secure programming. We focused on these three areas since they cover the vast majority of today's software development; additionally,

[2] cf. https://stackoverflow.com/

[3] cf. https://www.owasp.org

ID	Title	Organization	Description
1.	Safeguard your code: 17 security tips for developers	InfoWorld, an online magazine for IT and business professionals	Article dated 2013
2.	Best Practices for Security & Privacy	Google	Online Android training materials from Google.
3.	Secure Coding Practice Guidelines	UC Berkeley	Guidance on ways to satisfy application software security requirements, mainly in the form of links to other resources.
4.	Start with Security: A Guide for Business	US Government Federal Trade Commission	Paper with guidelines on corporate IS security
5.	Android Secure Coding Standard	Software Engineering Institute, CMU	Wiki with extensive guidelines.
6.	Mobile Application Security: 15 Best Practices for App Developers	Checkmarx, a development tool vendor	Short blog article
7.	Top 10 Secure Coding Practices	Software Engineering Institute, CMU	Short summary of 10 general secure coding principles
8.	Secure Coding Guide	Apple	Extensive online handbook covering technical aspects of iOS security
9.	Secure Mobile Development Guide	NowSecure, company specializing in app security testing and support	Online handbook covering many aspects of mobile app security.
10.	Secure Coding Practices Quick Reference Guide Project	OWASP, a not-for-profit dedicated to application security	Checklist of around 100 short bullet-point entries around general coding
11.	Secure Coding Cheat Sheet	OWASP	More detailed handbook describing principles of general coding.
12.	Security Guidance for Applications	Microsoft	Handbook with security guidance for web applications (outdated)
13.	Security Checklist for Software Developers	CERN, a research organization	Site with general guidelines and tips for specific languages.
14.	Website security	Mozilla	Extensive training article on web application security.
15.	Web Fundamentals: Security and Identity	Google	Several tips on web app security, with emphasis on Google tools.
16.	Developer How To Guide	SANS, a not-for-profit specializing in software security training	Article on how to avoid three common web security vulnerabilities.
17.	8 Tips for Better Mobile Application Security	UpWork, a developer recruitment site	Blog article
18.	TOP 25 Most Dangerous Software Errors	SANS with MITRE, a non-profit research company	Handbook, exploring types of security errors
19.	Intro to secure Windows app development	Microsoft	Extensive handbook to secure programming in the MS Windows environment.

FIGURE 6.1: The 19 security guides that we identified and analyzed.

mobile and web security issues have received a lot of attention from the security and privacy research community in the past.

Table 6.1 shows the resources we analyzed. We use the term 'handbook' to describe structured websites containing relatively large amounts of information; 'article' to describe smaller sites, mainly arranged linearly.

6.3 Evaluating Resources

We evaluated the 19 guides using a content-analysis-based manual coding approach [140]. We first defined a code book that listed desirable features that might be present in a secure-development guide. These features are listed as the heading to Table 6.4. We identified several main categories:

6.3.1 Source

The source category describes the author of a guide: we distinguish between official guidelines written by a framework/platform company, e.g., Google for Android or Apple for iOS, and a third-party organization (for-profit or non-profit, e.g., OWASP) providing a guide as a community service.

6.3.2 Content Organization

This category distinguishes different features that describe the content organization of a guide. We distinguish two different types of guides: brief, single-section articles and more detailed, multi-section handbooks. We noted whether a guide contained

a tutorial, defined as a step-by-step walkthrough of a specific real-world example, or any exercises to help developers become familiar with a certain security mechanism in a risk-free way. Whenever we found code examples, we checked if they were ready to use to solve certain small tasks, e.g., encrypting a file or establishing a secure HTTPS connection. We thought this to be important as ready-to-use code snippets can be copied directly into the programmer's source code without major changes. We distinguish these ready-to-use snippets from examples that present or describe specific API calls one at a time rather than in usage context. In addition, we checked whether a guide contained references to code repositories such as GitHub or BitBucket. We noted when a guide referenced external articles, blog or Wikipedia posts, or other external information sources. We noted the last time that the guide was updated as a measure of whether it was outdated or maintained. Finally, we analyzed how easy it was to find specific information in a guide: this included checking whether a guide had multiple hierarchical sections to avoid extensive scrolling, whether multiple levels of information granularity were provided (e.g., for novices, experienced programmers, and experts) and whether the guide was easily searchable.

6.3.3 Covered Topics

We next analyzed the different topics that were covered in the guides we examined. We started from an initial list of topics we thought were important, and added others as we encountered them in different guides. The final list of topics included:

- **Obfuscation**: Motivation for code obfuscation obfuscation techniques and tools.
- **Cryptography**: Encrypted data storage, secure key generation, etc.
- **Secure networking**: TLS/SSL, HTTPS, etc.
- **Storage management**: Backups, secure deletion, and proper management of shared storage.
- **Privileges**: Responsible use of privileges, principle of least privilege.
- **User input**: Proper validation of user input to avoid, e.g., SQL injection, cross-site scripting, and memory errors.
- **Use of tools**: Using of automated security-test tools including linters and other program-analysis techniques.
- **Mobile security**: Mobile-specific topics including privacy implications of mobile device tracking and geolocation.
- **Library use**: Using trusted libraries to avoid reinventing the security wheel, validating that selected libraries do not introduce malicious behavior.
- **Testing**: Measures for examining finished or deployed systems, including code review, fuzzing and penetration testing.
- **Data minimization**: Limiting the collection, storage, and transmission of personal information to protect users' privacy.
- **Regulations**: Security and privacy laws and regulations in various jurisdictions.

			Content Organization									Covered Topics													
Resource ID	Source[3]	Last Update[1]	Type[2]	Ready to use code	API examples	Exercise	Tutorials	Repository	Citations	Layered Information	Searchable	Sectioned	Obfuscation	Cryptography	Secure Networking	Storage Management	Privileges	User Input	Use of Tools	Mobile Security	Library Use	Testing	Data Minimization	Regulations	Threat Modelling	Logging	Password Advice	Social Engineering
ID 1 (InfoWorld)	t	2013-02-04	a	○	○	○	○	○	○	○	●	○	○	●	○	○	○	●	○	○	○	○	●	○	●	○	●	○
ID 2 (Google Android)	v	?	h	○	●	○	○	○	○	●	●	○	○	●	●	●	●	●	○	●	○	○	○	○	●	○	●	○
ID 3 (UC Berkeley)	o	?	a	○	○	○	○	○	●	●	●	●	○	●	●	●	●	●	●	○	●	●	○	○	●	○	●	○
ID 4 (US FTC)	o	2015-06	a	○	○	○	○	○	○	●	●	●	○	○	●	●	●	○	○	●	○	○	●	○	●	●	●	○
ID 5 (SEI, CMU)	o	2016-07-11	h	●	○	○	○	○	○	●	●	●	●	●	●	●	●	●	●	●	○	○	○	○	●	●	○	○
ID 6 (Checkmarx)	t	2015-08-19	a	○	○	○	○	○	○	●	●	●	○	●	●	○	●	○	●	●	○	●	●	●	○	○	○	●
ID 7 (SEI, CMU, Top 10)	o	2011-03-01	a	○	○	○	○	○	○	●	●	●	○	○	○	○	●	●	●	○	○	●	○	○	●	○	○	○
ID 8 (Apple)	v	2016-09-13	h	○	●	○	○	○	○	●	●	○	○	●	●	●	●	●	●	●	●	●	○	○	○	○	○	●
ID 9 (NowSecure)	t	2017-03-05	h	○	●	○	○	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	○	○	○	●	○	●
ID 10 (OWASP, reference)	n	2010-08-11	h	○	○	○	○	○	○	○	●	○	○	●	●	●	●	●	○	○	●	●	●	○	●	●	●	○
ID 11 (OWASP, cheat sheet)	n	2017-04-18	h	○	○	○	○	○	●	●	●	○	○	●	●	○	●	●	○	○	●	●	●	○	○	●	●	○
ID 12 (Microsoft)	v	2003-07-01	h	●	○	○	○	○	●	●	●	○	○	●	●	●	●	●	○	○	○	●	●	○	●	●	○	○
ID 13 (CERN)	o	?	a	○	○	○	○	○	●	●	●	○	○	●	○	○	●	●	●	○	●	●	○	○	●	●	○	○
ID 14 (Mozilla)	n	2017-05-24	h	○	○	○	○	○	○	●	●	○	○	○	●	○	○	●	○	○	○	○	●	○	○	○	●	○
ID 15 (Google)	v	2017-05-22	h	●	○	○	○	○	●	●	●	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	●
ID 16 (SANS)	n	?	h	●	○	○	○	○	○	●	●	○	○	○	●	○	○	●	○	○	○	○	○	○	○	○	○	○
ID 17 (UpWork)	t	2017-01-15	a	○	○	○	○	○	○	●	○	○	●	●	●	○	○	○	○	○	○	●	○	○	○	○	○	○
ID 18 (SANS, MITRE)	n	2011-07-27	h	○	○	○	○	○	○	●	●	●	○	●	●	○	●	●	○	○	○	○	○	○	○	○	○	○
ID 19 (Microsoft)	v	2017-02-08	h	●	○	○	●	●	●	●	●	●	○	●	●	●	●	○	○	●	○	○	○	●	○	○	○	○

[1] ?: date not specified [2] a: short article, h: detailed handbook
[3] t: third-party, v: vendor, o: organization, n: non-profit

TABLE 6.1: Features of the guides, as determined by content analysis. A shaded circle (●) indicates a guide exhibits this feature; an empty circle (○) indicates it does not.

- **Threat modeling**: Design-level analysis of security requirements.
- **Logging**: Keeping records to enable post-hoc auditing.
- **Password advice**: Guidelines for the secure creation and storage of passwords (e.g., salting and hashing stored passwords).
- **Social engineering**: Tricking people into making security errors, such as giving away secret data improperly.

6.4 Results

Table 6.4 provides an overview of our results.

Sources and Organization. We found that the majority (>55%) of the resources we analyzed were written by companies. All of these companies are in some way involved in secure software development or benefit from it. In 16% of cases, the guides were published as part of the official developer documentation written by the vendor of a development framework such as Apple for iOS, Google for Android, and Microsoft for Windows Mobile. Interestingly, about a third of all guides were written by non-profit organizations like OWASP.

Most of the guides for which we could identify update times (10 of 15) were last updated within the last two years; one (Microsoft web-application guidance, ID 12)

could be considered entirely obsolete, as it dated from 2003. These findings suggest that when developers search for secure-programming resources, they will frequently but not always encounter up-to-date ones.

Overall, these resources did not tend to contain concrete, low-level guidance. We found only five guides (IDs 5, 12, 15, 16, 19) that contained ready-to-use code snippets; most of these offered help on secure network connections using TLS. An additional three guides included API examples that do not rise to the level of snippets: Google's Android security and privacy guide (ID 2) exhibits the Android cryptography API, Apple's Secure Coding Guide (ID 8) gives examples for platform-specific security mechanisms, and NowSecure (ID 9) contains negative examples that demonstrate how not to use their SSL API. Interestingly, all these API examples come from mobile application guides.

Sadly, none of the guidelines we checked contained exercises that would have helped developers to learn security APIs or frameworks. Similarly, only the Microsoft Windows handbook (ID 19) contained a tutorial introducing developers to secure web programming techniques or referenced an external GitHub code repository with example code demonstrating specific security features. Only six guides referenced external information sources; a lack of such citations potentially undermines a reader's confidence in the guide's accuracy and inhibits further reading and learning. Overall, these results provide evidence of an important guidance gap noted in our prior work [4]: official documents and corporate guidelines do not provide the same level of detail and focus on utility as, for example, Q&A sites.

On the positive side, we were heartened to observe that most guides provide layers of advice that can target readers with different skill levels and are easily searchable.

Coverage of Topics. We found significant variation in coverage across topics. Some important topics are well covered: cryptography, secure networking, privilege management, and user input validation were all covered in at least 14 guides, and testing was covered in 10. On the other hand, there are several potentially critical topics with little to no coverage. Social engineering, which is a source of many security horror stories from phishing to recklessness with USB drives, is covered by only four guides. Logging, which is critical for being able to audit a system and understand any potential problems, is covered by only seven. Despite years of intensive research and commercial development creating and improving program-analysis tools, these tools are also only mentioned in seven guides. Another topic mentioned in only seven guides is data minimization, a critical modern concept in a world of exploding data mining. Some of the guides we analyzed may be too old to recognize the critical importance of data minimization; nonetheless it is concerning that developers searching for security help may not encounter it. Using and validating trusted libraries is an important and well-regarded practice, but it is mentioned by only six guides, perhaps because it is assumed to be implied. Finally, we note that laws and regulations are mentioned in only two guides. One might assume this is an issue for lawyers and executives rather than for software developers, but independent developers and small companies (the kind of developers most likely to be searching for guidance on the internet rather than talking to a company's dedicated security team) may not have separate compliance departments either. Further, developers who have some knowledge of legal requirements are less likely to make accidental errors that violate regulations.

We speculate that these coverage limitations arise from the nature of the authors creating these guides. The coverage suggests a predominantly traditional approach to security, based around technical support for developers. Looking at the results,

we may conclude that in future better results may come from having a cross discipline team create the documentation, with representation from test, support, product management, and legal experts.

6.5 Discussion

Our brief survey of general security guidance available on the web provides some insight into what developers—especially those without formal security training and/or without corporate security support—may encounter when they search for information about how to write secure code. They will find accessible information, appropriately layered and searchable, with good coverage of cryptography, secure networking and the handling of user input and privileges. However there are significant areas of concern: some readily available advice is outdated; most of this general-purpose guidance does not provide concrete examples or exercises; and some critical topics like program analysis tools, logging/auditing, and data minimization are not well represented. To remedy this would require a rather different team of authors from traditional security writers: pedagogical experts to generate exercises and tutorials, and human-centred security experts and legal experts to deal with social engineering and regulations.

Our prior work found that “official" guidance (from Google and from books) could promote stronger security outcomes than community-based guidance from Stack Overflow [4] (Chapter 3). The results of this survey, however, underscore another conclusion from that work: these “official" documents may not necessarily provide the content and format that developers want or need in practice.

In this work, we take the preliminary step of identifying and classifying a multitude of information sources. Further work is needed to assess their quality, and to understand how developers use these guides as well as how to increase their effectiveness. Therefore, the question of how best to organize online security help, such as guides, crowdsourcing sites, aggregators that combine various sources of security advice, as well as how to ensure that the quality of such advice remains high, remains open.

Thus, our results suggest two paths for understanding and improving the security-guidance ecosystem. First, we must examine whether and which of the gaps we have identified here—both in content and in format—represent serious omissions, and which are filled by other resources outside the scope of this paper. Second, we must continue to empirically evaluate the existing guidance, to understand which approaches do and don’t prove to be effective and why. By understanding what current guidance is missing, where it succeeds, and where it fails, we can hope to provide a framework for developing better guidance, both now and as secure programming continues to evolve.

Chapter 7

Conclusions and Future Work

This concluding part of the dissertation builds on, and sometimes uses verbatim, conclusions and future work ideas from the previously published papers that contributed to this dissertation, and for better coherence, are also contained in Chapters 1, 2, 3, 4, 5, and 6. As detailed in the individual chapters, this research was collaborative in nature and therefore uses the academic "we".

Throughout this dissertation, we have demonstrated that human factors are a major factor in secure software development.

First, we systematized research on appified ecosystems, finding that, like many new technologies, Android is a story of both victory and defeat. New security mechanisms were introduced without a clear understanding of how these applications would be developed and used, and well-established security mechanisms were reused to meet the expected security needs of the new general purpose computing platform. Some of the these techniques were a great success, while others failed almost entirely. Some aspects worked out beautifully, e.g., centralizing software distribution helps to tackle critical security issues and makes fighting piracy and malware easier. Other approaches had initial difficulties, but are now more or less on track after research has helped to identify and bridge them. Examples comprise easier-to-use APIs that have started to replace hard-to-use but well-intended security APIs over the last few years, as well as the concept of Webification that has enabled more developers to produce their own apps. However, some approaches should be rethought from the beginning and arguably abandoned for designs of future OSes: Permission dialogs for end users should be removed entirely, since they failed for the same reasons warning messages have failed since the dawn of computing.

Second, we followed this research up with empirical research on how decisions made by API developers, namely on documentation, can impact security outcomes. We started out with anecdotal evidence that suggested that copying and pasting code snippets from the internet can directly lead to insecure code outcomes. In Chapter 3, we conducted the first systematic investigation of this theory, by empirically investigating the question of how programming resources affect Android developers' security- and privacy-relevant decisions from several different angles, including an online survey that helped us scope resource use and realistic programming problems, a controlled programming experiment with isolated conditions for resource use and a control condition that represents real-world free resource use, a manual analysis of relevant Stack Overflow threads, and a large scale analysis of Android apps. Our results suggest that developers use both peer-created resources that are not moderated for security, like Stack Overflow, and official (moderated for security) resources to help solve problems while programming. However, we found that, compared to official documentation, Stack Overflow significantly contributes

to achieving functional code; however, using Stack Overflow also contributes to insecure applications. Compared to this, while often approached, official API documentation is less accessible, and less often successful in helping developers achieve functional code, which contributes to the prominence of insecure resource use. We do not think that it's likely for Android developers to give up using resources that help them quickly address their immediate problems in favor of secure outcomes. Therefore, we think it is critical to develop documentation and resources that combine the usefulness of forums like Stack Overflow with the security awareness of books or official API documents. One approach might involve rewriting API documents to be more usable, e.g. by adding secure and functional code examples, and making them search-able for common errors. Another might be to develop a separate programming-answers site in which experts address popular questions, perhaps initially drawn from other forums, in a security-sensitive manner. Alternatively, Stack Overflow could add a mechanism for explicitly rating the security of provided answers and weighting those rated secure more heavily in search results and thread ordering. Further research is needed to develop and evaluate solutions to help prevent inexperienced or overwhelmed mobile developers from making critical mistakes that put their users at risk.

Third, as a next step, we aimed to evaluate the efficacy of efforts to make cryptographic libraries more usable. We conducted a 256-participants online experiment, where we asked participants to use different cryptographic libraries to write Python code to solve security-relevant tasks, and evaluated their success in solving the tasks securely. Our results suggest that usability and security are deeply interconnected in sometimes surprising ways. Our high-level findings show that simplicity in libraries does promote security, as intended by library developers—to a point. In general, the simplified libraries we tested produced more secure results than the comprehensive libraries, validating the belief that simplicity is better. However, even simplified libraries did not entirely solve the security problem; for all but one library, the rate of security success was below 80%, frequently, missing features caused the problem. More problematically, for 20% of functional solutions, participants rated their code as secure when it was not; this indicates a dangerous gap in recognition of potential security problems. Several libraries in our study did not (or not well) support tasks auxiliary to encryption and decryption, such as secure key storage and password-based key generation. These missing features caused many of the insecure results in the otherwise-successful simplified libraries. We argue that to be usably secure, a cryptographic API must support such auxiliary tasks, rather than relying on the developer to recognize the potential for danger and identify a secure alternate solution. Further, we suggest that cryptographic APIs should be designed to support a reasonably broad range of use cases; requiring developers to learn and use new APIs for closely related tasks seems likely to drive them back to comprehensive libraries, which pose security risks. Future work might want to survey common main and auxiliary tasks to provide a benchmark for a comprehensive feature set. Documentation is also critical. Documentation containing insecure examples contributed to insecure code being used by participants. Incomplete or inaccessible documentation contributes to participants leaving the official documentation to search for help on Stack Overflow and blogs, which often resulted in insecure solutions. This suggests the importance of creating official documentation that is useful enough to keep developers from searching out unvetted, potentially insecure alternatives. Many participants copied these insecure examples in their solutions. In contrast, library documentations that contained useful code examples appear to have contributed to high rates of security success. Taken together, our results suggest several important

considerations for designers of future cryptographic libraries. First, the recent emphasis on simplifying APIs (and choosing secure defaults) has provided improvement; we endorse continuing in this direction. We suggest, however, that library designers go further, by treating documentation quality as a first-class requirement, with particular emphasis on secure code examples. We also recommend that library designers consider a broad range of potential tasks users might need to accomplish cryptographic goals, and build support for each of them into a more comprehensive whole.

Our results suggest that supporting holistic, application-level tasks with ready-to-use APIs is the best option. That said, we acknowledge that it may be difficult or impossible to predict all tasks API users may want or need. Therefore, where lower-level features are necessary, they should be intentionally designed to make combining them into more complex tasks securely as easy as possible.

Looking forward, further research is needed to design and evaluate libraries that meet these goals. Some changes can also be made within existing libraries—for example, improving documentation, changing insecure defaults to secure defaults, or even adding compiletime or runtime warnings for insecure parameters. These changes should be evaluated involving future users both before they are deployed and longitudinally to see how they affect outcomes within real-world code. We also hope to refine, expand, and validate the usability scale we developed to create an evaluation framework for security APIs generally, providing both feedback and guidance for improvement.

Fourth, we explored the novel strategy of recruiting developers directly from GitHub, and the properties of the recruited population, especially as they pertain to programming studies with security outcomes. In our two previous experiments, the question was raised whether recruiting students in lieu of professional developers impacts study validity. The GitHub sample contained both student and professional developers, and allowed us to explore that question. In our online quasi-experiment with 307 GitHub participants, we measured functionality and security outcomes across Python programming tasks. We came into the experiment hypothesizing that whether or not a participant wrote code professionally or as a student would impact at least the functional correctness of their code. However, we found that neither student nor professional status (self-reported) was a significant factor for functionality, security, or security perception. We were also surprised to learn that self-reported security background was equally unimportant. We did, however, find a significant effect for Python experience: additional experience correlated with better functionality and security outcomes, and we did find that professionals reported significantly more years of experience. While expertise with Python in our study differs significantly between students and professional developers, their security and privacy expertise is similar (in both cases quite low). At least within GitHub then, it seems that students and professionals can be equally useful for studying usable security and privacy problems, particularly if overall experience is controlled for. We speculate that the very similar results across students and professional developers can be accounted for in part because writing security-related code is not a regular task for average software developers, nor is security education a strong focus at many universities teaching computer science. We hypothesize, therefore, that overall these results — experience matters somewhat, but professional status on its own does not — would continue to hold for student and professional populations recruited more traditionally, at local universities and via professional networks. Based on results from Chapter 3, we suspect that typically local university students may have less experience than students recruited from GitHub. We expect that future

research will compare student samples recruited locally at various universities to samples recruited online. We found the recruitment strategy of emailing GitHub developers to be convenient in many ways: We were able to recruit many experienced professionals quickly and at a low cost; many participants expressed to us how much they enjoyed the challenge of our tasks and the opportunity to contribute to our research. However, it does have important drawbacks: we received complaints about unsolicited email from 13 invited GitHub committers and were generally subject to a small opt-in rate. We also found that our participants were slightly more active and therefore not quite representative of the GitHub population; representativeness for professionals (or students) in general is considerably less likely. Overall, the practice of sending unsolicited emails was not ideal, and is unlikely to be sustainable over many future studies. Instead, future work includes developing a GitHub application that would allow developers who are interested in contributing to research to opt in to study recruitment requests, which would benefit both these developers and the research community.

Fifth, as Chapters 3 and 4 had underlined the importance of usable documentation and complete, secure advice sources, we conducted a survey of general online security advice for developers; assuming that this is the advice encountered especially by those without formal security training and/or without corporate security support when they search for information about how to write secure code. Our survey finds that they will find accessible information, appropriately layered and searchable, with good coverage of some topics central to security, while other topics are not sufficiently covered. However there are significant areas of concern: some readily available advice is outdated; most of this general-purpose guidance does not provide concrete examples or exercises; and some critical topics are not well represented. To remedy this would require a rather different team of authors from traditional security writers: pedagogical experts to generate exercises and tutorials, and human-centred security experts and legal experts to deal with social engineering and regulations. As shown in Chapters 3 and 4, security advice from official sources (as compared to developer forums like Stack Overflow) may not provide the content and format that is useful for developers in practice. In this work, we take the preliminary step of identifying and classifying a multitude of information sources. Further work is needed to assess their quality, and to understand whether and how developers use these guides as well as how to increase their effectiveness. Therefore, the question of how best to organize online security help – guides, crowdsourcing sites, aggregators that combine various sources of security advice – as well as how to ensure that the quality of such advice remains high – remains open. Thus, our results suggest two paths for understanding and improving the security-guidance ecosystem. First, we must examine whether and which of the gaps we have identified here—both in content and in format—represent serious omissions, and which are filled by other resources outside the scope of this survey. Second, we must continue to empirically evaluate the existing guidance, to understand which approaches do and don't prove to be effective and why. By understanding what current guidance is missing, where it succeeds, and where it fails, we can hope to provide a framework for developing better guidance, both now and as secure programming continues to evolve.

Finally, we think that more generally, we need a systematic, organized effort to understand developers' attitudes, needs, and priorities toward security. Based on this understanding, security tools and APIs can be improved to increase adoption and adherence. We recommend further research aimed at moving security management and security-critical decisions from apps to the OS and framework levels

whenever possible. This includes but is not limited to automatic security library updates, or automatic permission requests on Android. Not only could this reduce developers' opportunities to make errors, but it is also compatible with the tendency to prioritize reducing development time and effort over security correctness. Research is needed to identify cases where this is possible as well as to suggest effective ways to remove developers from the security loop without overly restricting functionality. Advancing usable security for developers will be challenging, but it has the potential to bring already-known solutions into greater use and provide enormous benefits to the overall security ecosystem.

Appendix A

Appendix: Android Documentation Study

A.1 Exit Survey Questions

Using the agreement scale (hand them sheet), how much do you agree with the following statements?

- The login task was difficult.
- The login task was fun.
- I'm confident I got the right answer for the login task.
- The HTTPS task was difficult.
- The HTTPS task was fun.
- I'm confident I got the right answer for the HTTPS task.
- The phone task was difficult.
- The phone task was fun.
- I'm confident I got the right answer for the phone task.
- The manifest task was difficult.
- The manifest task was fun.
- I'm confident I got the right answer for the manifest task.
- The documentation and resources I used were easy to use.
- The documentation and resources I used were helpful.
- The documentation and resources I used were correct.

Free response questions. *[Questions 2-4 apply to restricted documentation only.]*

- What about the documentation you used was easy, helpful or correct? (Give them a chance to elaborate on the Likerts).
- Have you used [documentation] before?
- If you had been allowed to use different documentation, do you think you would have performed differently? Why/how/why not?

- What different documentation would you prefer (if any)? Why/why not?
- Would you have performed differently if you'd had more time? If yes, how?
- When you performed any of the tasks, were you thinking about security or privacy? Which tasks and what specifically? (Prompt for each task)
- Have you written similar code or come across similar problems in the past? (Prompt for each task). Tell us about it. When was it and what did you do?

Demographics and past experience

- For how long have you been developing Android apps?
- How many Android apps have you written in the past? When did you last work on an Android app?
- How long have you been programming in general (not just for Android)?
- How did you learn to code? (self-taught, online class, in university (BS/MS/PhD/no degree?), in professional certification program, on the job, coding bootcamp/hacker camp, etc.)
- How many Android classes have you taken (hours + semesters)? When did you last take an Android class?
- Is developing Android apps your primary job?
- Which IDE do you use to develop Android apps (Eclipse, Android Studio, other?)
- Have you ever taken a security class? How many / where / at what education level? How about a class that had security as one major component or module?
- Do you have experience working in computer security or privacy outside of school? Professionally or as a hobby? When/how much?
- What documentation or resources do you normally use when developing Android apps?
- When solving a security- or privacy-related problem (for android), do you use any different resources than those you already mentioned?
- What is your gender?
- What is your age?
- What is your occupation?
- What country did you (primarily) grow up in?
- What is your native language and what other languages are you fluent in (if any)?

Appendix B

Appendix: Cryptographic APIs Study

B.1 Errors

Table B.1 details the different types of security errors made by our participants, across the libraries we tested and the tasks we assigned. Our definitions of security are discussed in Section 4.3.8.

B.2 Survey

Task-specific questions: Asked about each task
Please rate your agreement to the following statements: (Strongly agree; agree; neutral; disagree; strongly disagree; I don't know.)

- I think I solved this task correctly.
- I think I solved this task securely.
- The documentation was helpful in solving this task.

General questions

- Are you aware of a specific library or other resource you would have preferred to solve the tasks? Which? (Yes with free response; no; I don't know.)
- Have you used or seen the assigned library before? For example, maybe you worked on a project that used the assigned library, but someone else wrote that portion of the code. (I have used the assigned library before; I have seen the assigned library used but have not used it myself; No, neither; I don't know.)
- Have you written or seen code for tasks similar to this one before? For example, maybe you worked on a project that included a similar task, but someone else wrote that portion of the code. (I have written similar code; I have seen similar code but have not written it myself; No, neither; I don't know.)

System Usability Scale (SUS)
We asked you to use the assigned library and the following questions refer to the assigned library and its documentation. Please rate your agreement or disagreement with the following statements: (Strongly agree; agree; neutral; disagree; strongly disagree)

Symmetric Keygen	Key Size	Key in Plain	Weak Cipher	Weak Mode	Static IV	No KDF	Custom KDF	KDF Salt	KDF Algo.	KDF Iter.
PyCrypto	6	4	11	14	3	15	11	1	1	2
M2Crypto	2	2	0	0	7	4	2	2	1	1
cryptography.io	1	7	0	0	0	1	3	10	0	0
Keyczar	0	3	0	0	0	1	0	0	0	0
PyNaCl	0	2	0	0	0	1	17	1	1	0

Symmetric Encryption	No Enc.	Weak Algo.	Weak Mode	Static IV
PyCrypto	0	17	23	29
M2Crypto	0	0	1	9
cryptography.io	0	0	0	0
Keyczar	0	0	0	0
PyNaCl	0	0	0	0

Asymmetric Keygen	Key Size	Key in Plain	Weak Cipher	Weak Mode	Static IV	No KDF	Custom KDF	KDF Salt	KDF Algo.	KDF Iter.
PyCrypto	6	0	0	0	0	0	0	0	0	0
M2Crypto	6	0	0	0	0	0	0	0	0	0
cryptography.io	0	0	0	0	0	0	0	0	0	0
Keyczar	0	1	0	0	0	0	0	0	0	0
PyNaCl	0	3	0	0	0	0	7	0	0	0

Asymmetric Encryption	Key Size	Padding
PyCrypto	9	0
M2Crypto	6	1
cryptography.io	0	0
Keyczar	0	0
PyNaCl	0	0

Certificate Validation	Sig. Check	CA Flag Check	Hostname Check	Date Check
PyCrypto	1	1	1	1
M2Crypto	2	13	11	14
cryptography.io	4	7	7	7
Keyczar	0	0	0	0
PyNaCl	1	1	1	1

TABLE B.1: Security errors made by our participants, as categorized by our codebook.

- I think that I would like to use this library frequently.
- I found the library unnecessarily complex.
- I thought the library was easy to use.
- I think that I would need the support of a technical person to be able to use this library.
- I found the various functions in this library were well integrated.
- I thought there was too much inconsistency in this library.
- I would imagine that most people would learn to use this library very quickly.
- I found the library very cumbersome to use.
- I felt very confident using the library.

- I needed to learn a lot of things before I could get going with this library.

Our usability scale
Please rate your agreement to the following questions on a scale from 'strongly agree' to 'strongly disagree.' (Strongly agree; agree; neutral; disagree; strongly disagree) Calculate the 0-100 score as follows: 2.5 * (5-Q_1 + $\sum_{i=2..10}(Q_i - 1)$); for the score, Q11 is omitted.

- I had to understand how most of the assigned library works in order to complete the tasks.
- It would be easy and require only small changes to change parameters or configuration later without breaking my code.
- After doing these tasks, I think I have a good understanding of the assigned library overall.
- I only had to read a little of the documentation for the assigned library to understand the concepts that I needed for these tasks.
- The names of classes and methods in the assigned library corresponded well to the functions they provided.
- It was straightforward and easy to implement the given tasks using the assigned library.
- When I accessed the assigned library documentation, it was easy to find useful help.
- In the documentation, I found helpful explanations.
- In the documentation, I found helpful code examples.

Please rate your agreement to the following questions on a scale from 'strongly agree' to 'strongly disagree.' (Strongly agree; agree; neutral; disagree; strongly disagree; does not apply)

- When I made a mistake, I got a meaningful error message/exception.
- Using the information from the error message/exception, it was easy to fix my mistake.

Demographics

- How long have you been programming in Python? (Less than 1 year; 1-2 years; 2-5 years; more than five years)
- How long have you been coding in general? (Less than 1 year; 1-2 years; 2-5 years; more than five years)
- How did you learn to code? [all that apply] (self-taught, online class, college, on-the-job training, coding camp)
- Is programming your primary job? If yes: Is writing Python code (part of) your primary job?

- Do you have an IT-security background? If yes, please specify.
- Please tell us your highest degree of education. (dropdown)
- Please tell us your gender. (female, male, other (please specify), decline to say)
- How old are you? (free text, check that the answer is a number)
- What country/countries do you live in / which country/countries are you a citizen of? (dropdown)
- What is your occupation? (free text)

Appendix C

Appendix: GitHub Study

C.1 Exit Survey Questions

Task-specific questions: Each task has these questions
On a five-point scale, how much do you agree with the following statements: [strongly agree, agree, neither agree nor disagree, disagree, strongly disagree]

- The task was difficult. (for each task)
- I am confident my solution is correct. (for each task)
- I am confident my solution is secure. (for each task)

What makes this solution either secure or insecure? (free text per task)

When you performed the task, were you thinking about security or privacy? (for each task)

- yes, a lot
- yes, a little
- no

What specifically? (For each task) [free text]

Have you written similar code or come across similar problems in the past? (For each task).

- yes
- sort of
- no

Tell us about it. When was it and what did you do; did you do something differently? [free text]

Demographics and past experience
Check all that apply: Have you ever taken a computer security class?

- at an undergraduate level
- at a graduate level
- via online learning

- via professional training
- another way [specify]
- no, but I took a class that had security as one major component or module
- no

How many computer security classes total have you taken? [input a number]

When did you last take a computer security class? [input a year]

Check all that apply: Do you have experience working in computer security or privacy outside of school?

- Professionally (you got paid to do it)
- As a hobby
- No
- Other [specify]

Check all that apply: Have you ever taken a Python programming class?

- at an undergraduate level
- at a graduate level
- via online learning
- via professional training
- another way [specify]
- no, but I took a class that had Python as one major component or module
- no

How many total Python classes have you taken? [input a number]

When did you last take a Python class [input a year]

Do you have experience programming in Python outside of school?

- Professionally (you got paid to do it)
- As a hobby
- No
- Other [specify]

For how many years have you been programming in Python? [number]

How many Python projects have you worked on in the past? [number]

When did you last work on a Python project? [year]

For how many years have you been programming in general (not just in Python)? [number]

How did you primarily learn to program? (Choose one)

- Self-taught
- In a university / as part of a degree
- In an online learning program
- In a professional certification program
- On the job
- Other [specify]

What is your gender?

- Male
- Female
- Other
- Prefer not to answer

What is your age? [number]

Are you currently a student?

- Undergraduate
- Graduate
- Professional certification program
- Other [specify]
- Not a student

Are you currently employed at a job where programming is a critical part of your job responsibility? [yes/no]

What country did you (primarily) grow up in? [list of countries]

What is your native language (mother tongue)? [list of languages]

C.2 GitHub Demographics

Table C.1 compares demographics for invited users vs. participants.

C.3 Installed Python libraries

Table C.2 lists the Python libraries we pre-installed in the study infrastructure.

	Invited	**Pros**	**Students**	**Both**	**Neither**
Hireable	20.5%	19.4%	40.0%	30.6%	23.5%
Company listed	39.4%	43.4%	30.0%	38.9%	17.6%
URL to blog	48.0%	47.3%	40.0%	63.9%	58.8%
Biography added	14.1%	21.7%	20.0%	16.7%	29.4%
Location provided	62.0%	69.8%	50.0%	69.4%	29.4%
GitHub profile creation[2]	2158	2148	1712	2101	2191
GitHub profile last update[2]	22	20	23	18	14
Minimal/Maximal age	—	18/54	19/37	19/43	24/81
Average age (Std)	—	32.9 (6.7)	25.3 (5.2)	27.5 (4.7)	35.2 (12.7)
> 2 years coding experience	—	99.5%	100.0%	100.0%	100.0%
> 2 years Python experience	—	92.5%	85.7%	81.2%	88.7%
Security background	—	6.5%	4.8%	5.7%	6.2%
Male/Female[1]	—	96.5%/1.5%	100.0%/0.0%	94.3%/5.7%	96.9%/0.0%

[1] the remainder either answered "other" or prefer not to disclose their gender.
[2] days ago, median

TABLE C.1: GitHub demographics for invited users vs. our valid participants.

Library	Version
apsw	3.8.11.1.post1
backports-abc	0.5
backports.shutil-get-terminal-size	1.0.0
bcrypt	2.0.0
blinker	1.3
certifi	2016.9.26
cffi	1.9.1
chardet	2.3.0
configparser	3.5.0
cryptography	1.2.3
cryptography-vectors	1.2.3
decorator	4.0.10
ecdsa	0.13
entrypoints	0.2.2
enum34	1.1.6
file-encryptor	0.2.9
Flask	0.10.1
flufl.password	1.3
functools32	3.2.3.post2
idna	2.0
ipaddress	1.0.16
ipykernel	4.5.2
ipython	5.1.0
ipython-genutils	0.1.0
ipywidgets	5.2.2
itsdangerous	0.24
Jinja2	2.8
jsonschema	2.5.1
jupyter	1.0.0
jupyter-client	4.4.0
jupyter-console	5.0.0
jupyter-core	4.2.0
M2Crypto	0.22.6rc4
m2ext	0.1
macaron	0.3.1
MarkupSafe	0.23
mistune	0.7.3
nbconvert	4.2.0
nbformat	4.1.0

Library	Version
ndg-httpsclient	0.4.0
notebook	4.2.3
passlib	1.6.5
pathlib2	2.1.0
pexpect	4.2.1
pickleshare	0.7.4
prompt-toolkit	1.0.9
ptyprocess	0.5.1
pyasn1	0.1.9
pycparser	2.17
pycrypto	2.6.1
pycryptopp	0.6.0.12...
Pygments	2.1.3
pyinotify	0.9.6
PyNaCl	1.0.1
pyOpenSSL	0.15.1
pysodium	0.6.9.1
pysqlite	2.7.0
python-geohash	0.8.3
python-keyczar	0.715
python-mhash	1.4
pyzmq	16.0.2
qtconsole	4.2.1
requests	2.9.1
simplegeneric	0.8.1
singledispatch	3.4.0.3
six	1.10.0
smbpasswd	1.0.1
ssdeep	3.1
terminado	0.6
tlsh	0.2.0
tornado	4.4.2
traitlets	4.3.1
typing	3.5.3.0
urllib3	1.13.1
wcwidth	0.1.7
Werkzeug	0.10.4
widgetsnbextension	1.2.6
withsqlite	0.1

TABLE C.2: Pre-installed libraries.

Bibliography

[1] 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: *Proc. 22nd ACM Conference on Computer and Communication Security (CCS'15)*. ACM, 2015.

[2] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick McDaniel, and Matthew Smith. "Sok: Lessons learned from android security research for appified software platforms". In: *Proc. 37th IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 2016.

[3] Yasemin Acar, Michael Backes, Sascha Fahl, Simson Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. "Comparing the Usability of Cryptographic APIs". In: *Proc. 38th IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 2017.

[4] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L Mazurek, and Christian Stransky. "You Get Where You're Looking For: The Impact of Information Sources on Code Security". In: *Proc. 37th IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 2016.

[5] Yasemin Acar, Sascha Fahl, and Michelle L. Mazurek. "You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users". In: *Proc. 2016 IEEE Secure Development Conference (SecDev'16)*. IEEE, 2016.

[6] Yasemin Acar, Christian Stransky, Dominik Wermke, Michelle L. Mazurek, and Sascha Fahl. "Security Developer Studies with GitHub Users: Exploring a Convenience Sample". In: *Proc. 13th Symposium on Usable Privacy and Security (SOUPS'17)*. USENIX Association, 2017.

[7] Alessandro Acquisti, Laura Brandimarte, and George Loewenstein. "Privacy and human behavior in the age of information". In: *Science* 347.6221 (2015), pp. 509–514.

[8] Alessandro Acquisti, Leslie K John, and George Loewenstein. "What is privacy worth?" In: *The Journal of Legal Studies* 42.2 (2013), pp. 249–274.

[9] Anne Adams and Martina Angela Sasse. "Users are not the enemy: Why users compromise security mechanisms and how to take remedial measures". In: *Communications of the ACM* 42.12 (1999), pp. 40–46.

[10] Devdatta Akhawe and Adrienne Porter Felt. "Alice in Warningland: A Large-Scale Field Study of Browser Security Warning Effectiveness." In: *Proc. 22nd Usenix Security Symposium (SEC'13)*. USENIX Association, 2013.

[11] Abrar Al-Heeti. *WhatsApp: 65B messages sent each day, and more than 2B minutes of calls*. `https://www.cnet.com/news/whatsapp-65-billion-messages-sent-each-day-and-more-than-2-billion-minutes-of-calls/`. May 2018.

[12] Amnesty International USA. *Encryption - A Matter of Human Rights*. `https://www.amnestyusa.org/sites/default/files/encryption_-_a_matter_of_human_rights_-_pol_40-3682-2016.pdf`. 2016.

[13] Ross J. Anderson. "Why Cryptosystems Fail". In: *Communications of the ACM* 37.11 (1994), 32–40.

[14] Noah Apthorpe, Sarah Varghese, and Nick Feamster. "Evaluating the Contextual Integrity of Privacy Regulation: Parents' IoT Toy Privacy Norms Versus COPPA". In: *Proc. 28th Usenix Security Symposium (SEC'19)*. USENIX Association, 2019.

[15] Steven Arzt, Sarah Nadi, Karim Ali, Eric Bodden, Sebastian Erdweg, and Mira Mezini. "Towards Secure Integration of Cryptographic Software". In: *Proc. 2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2015)*. ACM, 2015.

[16] 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: *Proc. ACM SIGPLAN 2014 Conference on Programming Language Design and Implementation (PLDI'14)*. ACM, 2014.

[17] Hala Assal and Sonia Chiasson. "Security in the Software Development Lifecycle". In: *Proc. 14th Symposium on Usable Privacy and Security (SOUPS'18)*. USENIX Association, 2018.

[18] Hala Assal and Sonia Chiasson. "'Think Secure from the Beginning': A Survey with Software Developers". In: *Proc. CHI Conference on Human Factors in Computing Systems (CHI'19)*. ACM, 2019.

[19] Kathy Wain Yee Au, Yi Fan Zhou, Zhen Huang, and David Lie. "PScout: Analyzing the Android Permission Specification". In: *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[20] Michael Backes, Sven Bugiel, Christian Hammer, Oliver Schranz, and Philipp von Styp-Rekowsky. "Boxify: Full-fledged App Sandboxing for Stock Android". In: *Proc. 24th Usenix Security Symposium (SEC'15)*. USENIX Association, 2015.

[21] Michael Backes, Sebastian Gerling, Christian Hammer, Matteo Maffei, and Philipp von Styp-Rekowsky. "AppGuard – Enforcing User Requirements on Android Apps". In: *Proc. 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '13)*. Springer, 2013.

[22] R Balebako, A Marsh, J Lin, and J Hong. "The Privacy and Security Behaviors of Smartphone App Developers". In: *Proc. Workshop on Usable Security (USEC'14)*. The Internet Society, 2014.

[23] Rebecca Balebako and Lorrie Cranor. "Improving App Privacy: Nudging App Developers to Protect User Privacy". In: *IEEE Security & Privacy* 12.4 (2014), pp. 55–58.

[24] Antoaneta Baltadzhieva and Grzegorz Chrupala. "Question Quality in Community Question Answering Forums: A Survey". In: *SIGKDD Explorations* 17.1 (2015), pp. 8–13.

[25] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. "Do Developers Read Compiler Error Messages?" In: *Proc. 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)*. IEEE, 2017.

[26] David Barrera, Jeremy Clark, Daniel McCarney, and Paul C. van Oorschot. "Understanding and Improving App Installation Security Mechanisms Through Empirical Analysis of Android". In: *Proc. 2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'12)*. ACM, 2012.

[27] David Barrera, William Enck, and Paul C. Van Oorschot. "Meteor: Seeding a Security-Enhancing Infrastructure for Multi-market Application Ecosystems". In: *Proc. 2012 Mobile Security Technologies Workshop (MoST'12)*. IEEE, 2012.

[28] David Barrera, H Güneş Kayacik, Paul C Van Oorschot, and Anil Somayaji. "A Methodology for Empirical Analysis of Permission-Based Security Models and its Application to Android". In: *Proc. 17th ACM Conference on Computer and Communication Security (CCS'10)*. ACM, 2010.

[29] Anton Barua, Stephen W. Thomas, and Ahmed E. Hassan. "What Are Developers Talking About? An Analysis of Topics and Trends in Stack Overflow". In: *Empirical Software Engineering* 19.3 (2012), pp. 619–654.

[30] Blerina Bazelli, Abram Hindle, and Eleni Stroulia. "On the Personality Traits of StackOverflow Users". In: *Proc. 29th IEEE International Conference on Software Maintenance (ICSM'13*. IEEE, 2013.

[31] *bcrypt*. `https://github.com/pyca/bcrypt`.

[32] Daniel J Bernstein, Tanja Lange, and Peter Schwabe. "The Security Impact of a New Cryptographic Library". In: *Proc. 2nd International Conference on Cryptology and Information Security in Latin America (LATINCRYPT'12)*. Springer, 2012.

[33] Robert Biddle, Sonia Chiasson, and P C van Oorschot. "Graphical passwords: Learning from the first twelve years". In: *ACM Computing Surveys* 44.4 (2012), 19:1–19:41.

[34] Joshua Bloch. "How to design a good API and why it matters". In: *Proc. 21st Conference on Object-oriented Programming Systems Languages and Applications (OOPSLA'06)*. ACM, 2006.

[35] David Botta, Rodrigo Werlinger, André Gagné, Konstantin Beznosov, Lee Iverson, Sidney Fels, and Brian Fisher. "Towards Understanding IT Security Professionals and Their Tools". In: *Proc. 3rd Symposium on Usable Privacy and Security (SOUPS'07)*. ACM, 2007.

[36] Russell Brandom. *Android Marshmallow's best security measure is a simple date*. `http://www.theverge.com/2015/9/29/9415313/android-marshmallow-security-update-vulnerability`. Last visited: 11/12/2015. 2015.

[37] Cristian Bravo-Lillo, Saranga Komanduri, Lorrie Faith Cranor, Robert W. Reeder, Manya Sleeper, Julie Downs, and Stuart Schechter. "Your Attention Please: Designing Security-decision UIs to Make Genuine Risks Harder to Ignore". In: *Proc. 9th Symposium on Usable Privacy and Security (SOUPS'13)*. ACM, 2013.

[38] Carolyn A Brodie, Clare-Marie Karat, and John Karat. "An empirical study of natural language parsing of privacy policy rules using the SPARCLE policy workbench". In: *Proc. 2nd Symposium on Usable Privacy and Security (SOUPS'06)*. ACM, 2006.

[39] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Towards Taming Privilege-Escalation Attacks on Android". In: *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. The Internet Society, 2012.

[40] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Stephan Heuser, Ahmad-Reza Sadeghi, and Bhargava Shastry. "Practical and Lightweight Domain Isolation on Android". In: *Proc. 1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'11)*. ACM, 2011.

[41] Sven Bugiel, Stephan Heuser, and Ahmad-Reza Sadeghi. "Flexible and Fine-Grained Mandatory Access Control on Android for Diverse Security and Privacy Policies". In: *Proc. 22nd Usenix Security Symposium (SEC'13)*. USENIX Association, 2013.

[42] Bureau of Labor Statistics. *Occupational Employment Statistics*. `https://www.bls.gov/oes/tables.htm`. 2016.

[43] K. P. Burnham. "Multimodel Inference: Understanding AIC and BIC in Model Selection". In: *Sociological Methods & Research* 33.2 (2004), pp. 261–304. URL: `http://smr.sagepub.com/cgi/doi/10.1177/0049124104268644`.

[44] Chris Burns, Jennifer Ferreira, Theodore D Hellmann, and Frank Maurer. "Usable results from the field of API usability: A systematic mapping and further analysis". In: *Proc. 2012 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'12)*. IEEE, 2012.

[45] J. Carver, L. Jaccheri, S. Morasca, and F. Shull. "Issues in using students in empirical studies in software engineering education". In: *Proc. 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry (Healthcom'03)*. IEEE, 2003.

[46] Cepl, Matěj. *M2Crypto*. `https://gitlab.com/m2crypto/m2crypto`.

[47] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. "MAST: Triage for Market-scale Mobile Malware Analysis". In: *Proc. 6th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'13)*. ACM, 2013.

[48] Patrick P.F. Chan, Lucas C.K. Hui, and Siu-Ming Yiu. "Droidchecker: analyzing android applications for capability leak". In: *Proc. 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'12)*. ACM, 2012.

[49] Avik Chaudhuri, Adam Fuchs, and Jeffrey Foster. *SCanDroid: Automated Security Certification of Android Applications*. Tech. rep. University of Maryland, 2009.

[50] Eric Y. Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. "OAuth Demystified for Mobile Application Developers". In: *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[51] Kai Chen, Peng Wang, Yeonjoon Lee, Xiao Feng Wang, Nan Zhang, Heqing Huang, Wei Zou, and Peng Liu. "Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale". In: *Proc. 24th Usenix Security Symposium (SEC'15)*. USENIX Association, 2015.

[52] Sonia Chiasson and P. Oorschot. "Quantifying the security advantage of password expiration policies". In: *Designs, Codes and Cryptography* 77.2 (2015), pp. 401–408.

[53] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. "Analyzing inter-application communication in Android". In: *Proc. 9th International Conference on Mobile Systems, Applications, and Services (MobiSys'11)*. ACM, 2011.

[54] Erika Chin, Adrienne Porter Felt, Vyas Sekar, and David Wagner. "Measuring User Confidence in Smartphone Security and Privacy". In: *Proc. 8th Symposium on Usable Privacy and Security (SOUPS'12)*. ACM, 2012.

[55] Erika Chin and David Wagner. "Bifocals: Analyzing WebView Vulnerabilities in Android Applications". In: *Proc. Information Security Applications - 14th International Workshop (WISA'13)*. Springer, 2013.

[56] Jeremy Clark, P C van Oorschot, and Carlisle Adams. "Usability of anonymous web browsing: an examination of Tor interfaces and deployability". In: *Proc. 3rd Symposium on Usable Privacy and Security (SOUPS'07)*. ACM, 2007.

[57] Steven Clarke. *Using the cognitive dimensions framework to design usable APIs*. `https://blogs.msdn.microsoft.com/stevencl/2003/11/14/using-the-cognitive-dimensions-framework-to-design-usable-apis/`. Last visited: 10/18/2020. 2003.

[58] Jessica Colnago, Yuanyuan Feng, Tharangini Palanivel, Sarah Pearman, Megan Ung, Alessandro Acquisti, Lorrie Faith Cranor, and Norman Sadeh. "Informing the Design of a Personalized Privacy Assistant for the Internet of Things". In: *Proc. CHI Conference on Human Factors in Computing Systems (CHI'20)*. ACM, 2020.

[59] Mauro Conti, Vu Thien Nga Nguyen, and Bruno Crispo. "CRePE: Context-Related Policy Enforcement for Android". In: *Proc. 13th Information Security Conference (ISC'10)*. Springer, 2010.

[60] Lorrie Cranor. *Time to rethink mandatory password changes*. `https://www.ftc.gov/news-events/blogs/techftc/2016/03/time-rethink-mandatory-password-changes/`. Mar. 2016.

[61] Lorrie Faith Cranor. "A Framework for Reasoning about the Human in the Loop". In: *Proc. Usenix Workshop on Usability, Psychology, and Security (UPSEC'08)*. USENIX Association, 2008.

[62] Jonathan Crussell, Clint Gibler, and Hao Chen. "Attack of the Clones: Detecting Cloned Applications on Android Markets". In: *Proc. 17th European Symposium on Research in Computer Security (ESORICS'12)*. Springer, 2012.

[63] Jonathan Crussell, Ryan Stevens, and Hao Chen. "MAdFraud: Investigating Ad Fraud in Android Applications". In: *Proc. 12th International Conference on Mobile Systems, Applications, and Services (MobiSys'14)*. ACM, 2014.

[64] *Cryptography.io*. `https://cryptography.io`.

[65] L. Davi, A. Dmitrienko, A.R. Sadeghi, and M. Winandy. "Privilege escalation attacks on Android". In: *Proc. 13th Information Security Conference (ISC'10)*. Springer, 2010.

[66] Benjamin Davis, Ben Sanders, Armen Khodaverdian, and Hao Chen. "I-ARM-Droid: A Rewriting Framework for In-App Reference Monitors for Android Applications". In: *Proc. 2012 Mobile Security Technologies Workshop (MoST'12)*. IEEE, 2012.

[67] Michael Dietz, Shashi Shekhar, Yuliy Pisetsky, Anhei Shu, and Dan S. Wallach. "QUIRE: Lightweight Provenance for Smart Phone Operating Systems". In: *Proc. 20th Usenix Security Symposium (SEC'11)*. USENIX Association, 2011.

[68] Whitfield Diffie and Martin Hellman. "New directions in cryptography". In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654.

[69] *ed25519*. `https://pypi.python.org/pypi/ed25519`.

[70] Manuel Egele, David Brumley, Yanick Fratantonio, and Christopher Kruegel. "An empirical study of cryptographic misuse in android applications". In: *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[71] Nikolay Elenkov. *Android Security Internals*. No Starch Press, 2015.

[72] Brian Ellis, Jeffrey Stylos, and Brad Myers. "The Factory Pattern in API Design: A Usability Evaluation". In: *Proc. 29th IEEE/ACM International Conference on Software Engineering (ICSE'07)*. IEEE, 2007.

[73] 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: (2010).

[74] William Enck, Damien Octeau, Patrick D McDaniel, and Swarat Chaudhuri. "A Study of Android Application Security". In: *Proc. 20th Usenix Security Symposium (SEC'11)*. USENIX Association, 2011.

[75] William Enck, Machigar Ongtang, and Patrick McDaniel. "On Lightweight Mobile Phone Application Certification". In: *Proc. 16th ACM Conference on Computer and Communication Security (CCS'09)*. ACM, 2009.

[76] William Enck, Machigar Ongtang, and Patrick McDaniel. "Understanding Android Security". In: *IEEE Security & Privacy* 7.1 (2009), pp. 50–57.

[77] Úlfar Erlingsson. "The Inlined Reference Monitor Approach to Security Policy Enforcement". PhD thesis. Cornell University, Jan. 2004.

[78] Michael D. Ernst, René Just, Suzanne Millstein, Werner Dietl, Stuart Pernsteiner, Franziska Roesner, Karl Koscher, Paulo Barros Barros, Ravi Bhoraskar, Seungyeop Han, Paul Vines, and Edward X. Wu. "Collaborative Verification of Information Flow for a High-Assurance App Store". In: *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[79] S Eskandari, D Barrera, and E Stobert. "A first look at the usability of bitcoin key management". In: *Proc. Workshop on Usable Security (USEC'15)*. The Internet Society, 2015.

[80] Sascha Fahl, Sergej Dechand, Henning Perl, Felix Fischer, Jaromir Smrcek, and Matthew Smith. "Hey, NSA: Stay away from my market! Future proofing app markets against powerful attackers". In: *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[81] Sascha Fahl, Marian Harbach, Yasemin Acar, and Matthew Smith. "On The Ecological Validity of a Password Study". In: *Proc. 9th Symposium on Usable Privacy and Security (SOUPS'13)*. ACM, 2013.

[82] 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: *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[83] Sascha Fahl, Marian Harbach, Thomas Muders, Matthew Smith, and Uwe Sander. "Helping Johnny 2.0 to encrypt his Facebook conversations". In: *Proc. 8th Symposium on Usable Privacy and Security (SOUPS'12)*. ACM, 2012.

[84] Sascha Fahl, Marian Harbach, Marten Oltrogge, Thomas Muders, and Matthew Smith. "Hey, you, get off of my clipboard - On How Usability Trumps Security in Android Password Managers". In: *Proc. 17th International Conference on Financial Cryptography and Data Security (FC'13)*. Springer, 2013.

[85] Sascha Fahl, Marian Harbach, Henning Perl, Markus Koetter, and Matthew Smith. "Rethinking SSL Development in an Appified World". In: *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[86] Adrienne Porter Felt, Alex Ainslie, Robert W. Reeder, Sunny Consolvo, Somas Thyagaraja, Alan Bettes, Helen Harris, and Jeff Grimes. "Improving SSL Warnings: Comprehension and Adherence". In: *Proc. 33rd Annual ACM Conference on Human Factors in Computing Systems (CHI'15)*. ACM, 2015.

[87] Adrienne Porter Felt, Erika Chin, Steve Hanna, Dawn Song, and David Wagner. "Android permissions demystified". In: *Proc. 18th ACM Conference on Computer and Communication Security (CCS'11)*. ACM, 2011.

[88] Adrienne Porter Felt, Serge Egelman, Matthew Finifter, Devdatta Akhawe, and David Wagner. "How to ask for permission". In: *Proc. 7th USENIX Workshop on Hot Topics in Security (HotSec'12)*. USENIX Association, 2012.

[89] Adrienne Porter Felt, Serge Egelman, and David Wagner. "I've got 99 problems, but vibration ain't one: A survey of smartphone users' concerns". In: *Proc. 2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'12)*. ACM, 2012.

[90] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. "A Survey of Mobile Malware in the Wild". In: *Proc. 1st ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'11)*. ACM, 2011.

[91] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. "Android Permissions: User Attention, Comprehension, and Behavior". In: *Proc. 8th Symposium on Usable Privacy and Security (SOUPS'12)*. ACM, 2012.

[92] Adrienne Porter Felt, Robert W. Reeder, Alex Ainslie, Helen Harris, Max Walker, Christopher Thompson, Mustafa Embre Acer, Elisabeth Morant, and Sunny Consolvo. "Rethinking Connection Security Indicators". In: *Proc. 12th Symposium on Usable Privacy and Security (SOUPS'16)*. USENIX Association, 2016.

[93] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. "Permission Re-Delegation: Attacks and Defenses". In: *Proc. 20th Usenix Security Symposium (SEC'11)*. USENIX Association, 2011.

[94] Matthew Finifter and David Wagner. "Exploring the Relationship Between Web Application Development Tools and Security". In: *Proc. 2nd USENIX conference on Web application development (WebApps'11)*. USENIX Association, 2011.

[95] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. "Stack Overflow Considered Harmful? The Impact of Copy&Paste on Android Application Security". In: *Proc. 38th IEEE Symposium on Security and Privacy (SP'17)*. IEEE, 2017.

[96] Elli Fragkaki, Lujo Bauer, Limin Jia, and David Swasey. "Modeling and Enhancing Android's Permission System". In: *Proc. 17th European Symposium on Research in Computer Security (ESORICS'12)*. Springer, 2012.

[97] Simson Garfinkel and Heather Richter Lipford. "Usable Security: History, Themes, and Challenges". In: *Synthesis Lectures on Information Security, Privacy, and Trust* 5.2 (2014), pp. 1–124.

[98] Simson L. Garfinkel and Robert C. Miller. "Johnny 2: a user test of key continuity management with S/MIME and Outlook Express". In: *Proc. 1st Symposium on Usable Privacy and Security (SOUPS'05)*. ACM, 2005.

[99] Martin Georgiev, Subodh Iyengar, Suman Jana, Rishita Anubhai, Dan Boneh, and Vitaly Shmatikov. "The most dangerous code in the world: validating SSL certificates in non-browser software". In: *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[100] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. "AndroidLeaks: Automatically Detecting Potential Privacy Leaks in Android Applications on a Large Scale". In: *Proc. 5th International Conference on Trust and Trustworthy Computing (TRUST'12)*. Springer, 2012.

[101] Clint Gibler, Ryan Stevens, Jonathan Crussell, Hao Chen, Hui Zang, and Heesook Choi. "AdRob: Examining the Landscape and Impact of Android Application Plagiarism". In: *Proc. 11th International Conference on Mobile Systems, Applications, and Services (MobiSys'13)*. ACM, 2013.

[102] *GitHub Archive*. `https://www.githubarchive.org`. visited. Nov. 2016.

[103] *GitHut: A Small Place to discover languages in github*. `http://githut.info`. visited. Nov. 2016.

[104] *gnupg*. `https://github.com/isislovecruft/python-gnupg`.

[105] Google. *Google Report: Android Security 2014 Year in Review*. `https://source.android.com/security/reports/Google_Android_Security_2014_Report_Final.pdf`. Apr. 2015.

[106] Google. *Nexus Security Bulletins*. `https://source.android.com/security/bulletin/index.html`. Last visited: 11/13/2015.

[107] Google. *Review app permissions thru Android 5.9*. `https://support.google.com/googleplay/answer/6014972?hl=en`. Last visited: 11/13/2015. 2015.

[108] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. "Checking App Behavior Against App Descriptions". In: *Proc. 36th IEEE/ACM International Conference on Software Engineering (ICSE'14)*. ACM, 2014.

[109] 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: *Proc. CHI Conference on Human Factors in Computing Systems (CHI'20)*. ACM, 2020.

[110] 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: *Proc. 14th Symposium on Usable Privacy and Security (SOUPS'18)*. USENIX Association, 2018.

[111] PL Gorski and L Lo Iacono. "Towards the Usability Evaluation of Security APIs". In: *Proc. Tenth International Symposium on Human Aspects of Information Security & Assurance (HAISA 2016)*. School of Computing & Mathematics Plymouth University, 2016.

[112] Michael Grace, Yajin Zhou, Zhi Wang, and Xuxian Jiang. "Systematic detection of capability leaks in stock Android smartphones". In: *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. The Internet Society, 2012.

[113] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. "RiskRanker: Scalable and Accurate Zero-day Android Malware Detection". In: *Proc. 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12)*. ACM, 2012.

[114] Michael C Grace, Wu Zhou, Xuxian Jiang, and Ahmad-Reza Sadeghi. "Unsafe Exposure Analysis of Mobile In-App Advertisements". In: *Proc. 5th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'12)*. ACM, 2012.

[115] Matthew Green and Matthew Smith. "Developers are Not the Enemy!: The Need for Usable Security APIs". In: *IEEE Security & Privacy* 14.5 (2016), pp. 40–46.

[116] Julie M Haney, Mary Theofanos, Yasemin Acar, and Sandra Spickard Prettyman. ""We make it a big deal in the company": Security Mindsets in Organizations that Develop Cryptographic Products". In: *Proc. 14th Symposium on Usable Privacy and Security (SOUPS'18)*. USENIX Association, 2018.

[117] Marian Harbach, Alexander De Luca, and Serge Egelman. "The Anatomy of Smartphone Unlocking". In: *Proc. CHI Conference on Human Factors in Computing Systems (CHI'16)*. ACM, 2016.

[118] Marian Harbach, Markus Hettig, Susanne Weber, and Matthew Smith. "Using Personal Examples to Improve Risk Communication for Security and Privacy Decisions". In: *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'14)*. ACM, 2014.

[119] Norman Hardy. "The Confused Deputy (or why capabilities might have been invented)". In: *ACM SIGOPS: Operating Systems Review* 22.4 (1988), pp. 36–38.

[120] Michi Henning. "API: Design Matters: Why Changing APIs Might Become a Criminal Offense." In: 5.4 (2007), pp. 24–36.

[121] C Herley. "More Is Not the Answer". In: *Proc. 35th IEEE Symposium on Security and Privacy (SP'14)*. IEEE, 2014.

[122] Stephan Heuser, Adwait Nadkarni, William Enck, and Ahmad-Reza Sadeghi. "ASM: A Programmable Interface for Extending Android Security". In: *Proc. 23rd Usenix Security Symposium (SEC'14)*. USENIX Association, 2014.

[123] Rolf Holtz and Norman Miller. "Assumed similarity and opinion certainty". In: *Journal of Personality and Social Psychology* 48.4 (1985), pp. 890–898.

[124] Martin Höst, Björn Regnell, and Claes Wohlin. "Using Students as Subjects—A Comparative Study of Students and Professionals in Lead-Time Impact Assessment". In: *Empirical Software Engineering* 5.3 (2000), pp. 201–214. URL: `http://dx.doi.org/10.1023/A:1026586415054`.

[125] Soumya Indela, Mukul Kulkarni, Kartik Nayak, and Tudor Dumitraş. "Helping Johnny Encrypt: Toward Semantic Interfaces for Cryptographic Frameworks". In: *Proc. 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward! 2016)*. ACM, 2016.

[126] S Jain and J Lindqvist. "Should I Protect You? Understanding Developers' Behavior to Privacy-Preserving APIs". In: *Proc. Workshop on Usable Security (USEC'14)*. The Internet Society, 2014.

[127] 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: *Proc. 2nd ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'12)*. ACM, 2012.

[128] Xing Jin, Xuchao Hu, Kailiang Ying, Wenliang Du, Heng Yin, and Gautam Nagesh Peri. "Code Injection Attacks on HTML5-based Mobile Apps: Characterization, Detection and Mitigation". In: *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[129] Brittany Johnson, Rahul Pandita, Justin Smith, Denae Ford, Sarah Elder, Emerson Murphy-Hill, Sarah Heckman, and Caitlin Sadowski. "A Cross-Tool Communication Study on Program Analysis Tool Notifications". In: *Proc. 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16)*. ACM, 2016.

[130] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. "Why don't software developers use static analysis tools to find bugs?" In: *Proc. 35th IEEE/ACM International Conference on Software Engineering (ICSE'13)*. IEEE, 2013.

[131] P. W. Jordan, B. Thomas, B. A. Weerdmeester, and A. L. McClelland. "SUS: A "quick and dirty" usability scale". In: *Usability Evaluation in Industry*. Taylor and Francis, 1996, pp. 189–194.

[132] Simon Josefsson. *PKCS #5: Password-Based Key Derivation Function 2 (PBKDF2) Test Vectors*. `http://www.ietf.org/rfc/rfc6070.txt`. Jan. 2011.

[133] *Jupyter Notebook*. `http://jupyter.org/`. visited. Nov. 2016.

[134] Kate Kelley, Belinda Clark, Vivienne Brown, and John Sitzia. "Good practice in the conduct and reporting of survey research". In: *International Journal for Quality in Health Care* 15.3 (2003), pp. 261–266.

[135] Patrick Gage Kelley, Lorrie Faith Cranor, and Norman Sadeh. "Privacy as Part of the App Decision-Making Process". In: *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'13)*. ACM, 2013.

[136] *Keyczar*. `https://github.com/google/keyczar`.

[137] Saranga Komanduri, Richard Shay, Patrick Gage Kelley, Michelle L Mazurek, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, and Serge Egelman. "Of Passwords and People: Measuring the Effect of Password-Composition Policies". In: *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'11)*. ACM, 2011.

[138] Satya Komatineni and Dave MacLean. *Pro Android 4*. Apress, 2012.

[139] H. Krawczyk, M. Bellare, and R. Canetti. *HMAC: Keyed-Hashing for Message Authentication*. `http://www.ietf.org/rfc/rfc2104.txt`. Feb. 1997.

[140] Klaus Krippendorff. *Content Analysis: An Introduction to Its Methodology (2nd ed.)* SAGE Publications, 2004.

[141] Enrique Larios Vargas, Maurício Aniche, Christoph Treude, Magiel Bruntink, and Georgios Gousios. "Selecting Third-Party Libraries: The Practitioners' Perspective". In: *Proc. 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2020.

[142] Lucas Layman, Laurie Williams, and Robert St Amant. "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools". In: *Proc. First International Symposium on Empirical Software Engineering and Measurement (ESEM'07)*. IEEE, 2007.

[143] David Lazar, Haogang Chen, Xi Wang, and Nickolai Zeldovich. "Why does cryptographic software fail?: a case study and open problems". In: *Proc. 5th Asia-Pacific Workshop on Systems (APSys'14)*. ACM, 2014.

[144] J Lee, L. Bauer, and M L Mazurek. "The Effectiveness of Security Images in Internet Banking". In: *IEEE Internet Computing* 19.01 (2015), pp. 54–62.

[145] Pedro Giovanni Leon, Lorrie Faith Cranor, Aleecia M McDonald, and Robert McGuire. "Token Attempt: The Misrepresentation of Website Privacy Policies Through the Misuse of P3P Compact Policy Tokens". In: *Proc. 9th Annual ACM Workshop on Privacy in the Electronic Society (WPES '10)*. ACM, 2010.

[146] Mario Linares-Vásquez, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Denys Poshyvanyk. "How Do API Changes Trigger Stack Overflow Discussions? A Study on the Android SDK". In: *Proc. 22th IEEE International Conference on Program Comprehension (ICPC'14)*. ACM, 2014.

[147] Dirk van der Linden, Emma Williams, Joseph Hallett, and Awais Rashid. "The impact of surface features on choice of (in) secure answers by Stackoverflow readers". In: *IEEE Transactions on Software Engineering* (2020), pp. 1–1.

[148] Darsey Litzenberger. *PyCrypto*. `https://www.dlitz.net/software/pycrypto`.

[149] Bin Liu, Jialiu Lin, and Norman Sadeh. "Reconciling Mobile App Privacy and Usability on Smartphones: Could User Privacy Profiles Help?" In: *Proc. 23rd International Conference on World Wide Web (WWW'14)*. ACM, 2014.

[150] lohan. *AntiLVL: android cracking*. `http://androidcracking.blogspot.in/p/antilvl_01.html`. Last visited: 11/06/15.

[151] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. "Chex: Statically Vetting Android Apps for Component Hijacking Vulnerabilities". In: *Proc. 19th ACM Conference on Computer and Communication Security (CCS'12)*. ACM, 2012.

[152] Tongbo Luo, Hao Hao, Wenliang Du, Yifei Wang, and Heng Yin. "Attacks on WebView in the Android system". In: *Proc. 27th Annual Computer Security Applications Conference (ACSAC'11)*. ACM, 2011.

[153] Raj Mallempati. *Google I/O Recap, Part 1: Google is Serious About Enterprise Mobility*. `https://www.mobileiron.com/en/smartwork-blog/google-io-recap-part-1-google-serious-about-enterprise-mobility`. Last visited: 11/13/2015. June 2014.

[154] Claudio Marforio, Hubert Ritzdorf, Aurélien Francillon, and Srdjan Čapkun. "Analysis of the Communication between Colluding Applications on Modern Smartphones". In: *Proc. 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, 2012.

[155] Vitaly Shmatikov Martin Georgiev Suman Jana. "Breaking and Fixing Origin-Based Access Control in Hybrid Web/Mobile Application Frameworks". In: *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[156] Arunesh Mathur, Josefine Engel, Sonam Sobti, Victoria Chang, and Marshini Chetty. ""They Keep Coming Back Like Zombies": Improving Software Updating Interfaces". In: *Proc. 12th Symposium on Usable Privacy and Security (SOUPS'16)*. USENIX Association, 2016.

[157] Michelle L Mazurek, Saranga Komanduri, Timothy Vidas, Lujo Bauer, Nicolas Christin, Lorrie Faith Cranor, Patrick Gage Kelley, Richard Shay, and Blase Ur. "Measuring Password Guessability for an Entire University". In: *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'13)*. ACM, 2013.

[158] P. McDaniel and W. Enck. "Not So Great Expectations: Why Application Markets Haven't Failed Security". In: *IEEE Security & Privacy* 8.5 (2010), pp. 76–78.

[159] David A McMeekin, Brian R von Konsky, Michael Robey, and David JA Cooper. "The significance of participant experience when evaluating software inspection techniques". In: *Proc. 20th Australian Conference on Software Engineering (ASWEC'09)*. IEEE, 2009.

[160] André Moulo. *Android OEM's applications (in)security and backdoors without permission*. `http://www.quarkslab.com/dl/Android-OEM-applications-insecurity-and-backdoors-without-permission.pdf`. 2014.

[161] D Movshovitz-Attias, Y Movshovitz-Attias, P Steenkiste, and C Faloutsos. "Analysis of the Reputation System and User Contributions on a Question Answering Website: StackOverflow". In: *Proc. 2013 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM'13)*. IEEE, 2013.

[162] Emerson Murphy-Hill, Da Y. Lee, Gail C. Murphy, and Joanna McGrenere. "How Do Users Discover New Tools in Software Development and Beyond?" In: *Computer Supported Cooperative Work (CSCW)* 24.5 (2015), pp. 389–422. URL: `http://people.engr.ncsu.edu/ermurph3/papers/jcscw15.pdf`.

[163] Patrick Mutchler, Adam Doupé, John Mitchell, Christopher Kruegel, and Giovanni Vigna. "A Large-Scale Study of Mobile Web App Security". In: *Proc. 2015 Mobile Security Technologies Workshop (MoST'15)*. IEEE, 2015.

[164] Andrew C. Myers and Barbara Liskov. "A Decentralized Model for Information Flow Control". In: *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP'97)*. ACM, 1997.

[165] Brad A Myers and Jeffrey Stylos. "Improving API Usability". In: 59.6 (2016), pp. 62–69.

[166] Sarah Nadi, Stefan Krüger, Mira Mezini, and Eric Bodden. ""Jumping Through Hoops": Why do Java Developers Struggle With Cryptography APIs?" In: *Proc. 37th IEEE/ACM International Conference on Software Engineering (ICSE'15)*. IEEE, 2016. URL: https://googledrive.com/host/0BypYpzQy3hi8YWw1Umw1cl9ZX0k/NADI_ICSE16.pdf.

[167] Nico JD Nagelkerke. "A note on a general definition of the coefficient of determination". In: *Biometrika* 78.3 (1991), pp. 691–692.

[168] National Institute of Standards and Technology (NIST). *NIST Special Publication 800-57 Part 1 Revision 4: Recommendation for Key Management*. http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf. 2016.

[169] National Institute of Standards and Technology (NIST). *NIST Special Publication 800-63B Digital Authentication Guideline*. https://pages.nist.gov/800-63-3/sp800-63b.html. 2016.

[170] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. "Apex: Extending Android Permission Model and Enforcement with User-defined Runtime Constraints". In: *Proc. 5th ACM Symposium on Information, Computer and Communication Security (ASIACCS'10)*. ACM, 2010.

[171] Lily Hay Newman. *The Worst Hacks and Breaches of 2020 So Far | WIRED*. https://www.wired.com/story/worst-hacks-breaches-2020-so-far/. Mar. 2020.

[172] 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: *Proc. 24th ACM Conference on Computer and Communication Security (CCS'17)*. ACM, 2017.

[173] Jakob Nielsen. *Usability engineering*. Morgan Kaufmann Publishers, 1993.

[174] Daniela Oliveira, Marissa Rosenthal, Nicole Morin, Kuo-Chuan Yeh, Justin Cappos, and Yanyan Zhuang. "It's the Psychology Stupid: How Heuristics Explain Software Vulnerabilities and How Priming Can Illuminate Developer's Blind Spots". In: *Proc. 30th Annual Computer Security Applications Conference (ACSAC'14)*. ACM, 2014.

[175] Marten Oltrogge, Yasemin Acar, Sergej Dechand, Matthew Smith, and Sascha Fahl. "To Pin or Not to Pin - Helping App Developers Bullet Proof Their TLS Connections". In: *Proc. 24th Usenix Security Symposium (SEC'15)*. USENIX Association, 2015.

[176] 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: *Proc. 30th Usenix Security Symposium (SEC'21)*. USENIX Association, 2021.

[177] Machigar Ongtang, Stephen E. McLaughlin, William Enck, and Patrick McDaniel. "Semantically Rich Application-Centric Security in Android". In: *Proc. 25th Annual Computer Security Applications Conference (ACSAC'09)*. ACM, 2009.

[178] Lucky Onwuzurike and Emiliano De Cristofaro. "Danger is my middle name: experimenting with SSL vulnerabilities in Android apps". In: *Proc. 8th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec'15)*. ACM, 2015.

[179] Open Handset Alliance. *Android*. `http://www.openhandsetalliance.com/android_overview.html`. Last visited: 11/13/2015. 2015.

[180] Open Signal. *Android Fragmentation Visualized (August 2015)*. `http://opensignal.com/reports/2015/08/android-fragmentation/`. Last visited: 11/06/2015. Aug. 2015.

[181] R Pandita, X Xiao, W Yang, W Enck, and T Xie. "WHYPER: towards automating risk assessment of mobile applications". In: *Proc. 22nd Usenix Security Symposium (SEC'13)*. USENIX Association, 2013.

[182] Nikhil Patnaik, Joseph Hallett, and Awais Rashid. "Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries". In: *Proc. 15th Symposium on Usable Privacy and Security (SOUPS'19)*. USENIX Association, 2019.

[183] Paul Pearce, Adrienne Porter Felt, Gabriel Nunez, and David Wagner. "AdDroid: Privilege Separation for Applications and Advertisers in Android". In: *Proc. 7th ACM Symposium on Information, Computer and Communication Security (ASIACCS'12)*. ACM, 2012.

[184] Henning Perl, Sascha Fahl, and Matthew Smith. "You Won't Be Needing These Any More: On Removing Unused Certificates from Trust Stores". In: *Proc. 18th International Conference on Financial Cryptography and Data Security (FC'14)*. Springer, 2014.

[185] Trevor Perrin. *tlslite*. `http://trevp.net/tlslite/`.

[186] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. "An Empirical Study of API Usability". In: *Proc. 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*. IEEE, 2013.

[187] Olgierd Pieczul, S. Foley, and Mary Ellen Zurko. "Developer-centered security and the symmetry of ignorance". In: *Proc. 2017 New Security Paradigms Workshop (NSPW'17)*. ACM, 2017.

[188] Sebastian Poeplau, Yanick Fratantonio, Antonio Bianchi, Christopher Kruegel, and Giovanni Vigna. "Execute This! Analyzing Unsafe and Malicious Dynamic Code Loading in Android Applications." In: *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[189] Andreas Poller, Laura Kocksch, Sven Türpe, Felix Anand Epp, and Katharina Kinder-Kurlanda. "Can Security Become a Routine?: A Study of Organizational Change in an Agile Software Development Group". In: *Proc. 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing (CSCW '17)*. ACM, 2017.

[190] L Ponzanelli, A Mocci, A Bacchelli, and M Lanza. "Understanding and Classifying the Quality of Technical Forum Questions". In: *Proc. 14th International Conference on Quality Software (QSIC'14)*. IEEE, 2014.

[191] Lutz Prechelt. "Plat_Forms: A web development platform comparison by an exploratory experiment searching for emergent platform properties". In: *IEEE Transactions on Software Engineering* 37.1 (2011), pp. 95–108.

[192] Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. "Evaluating the Evaluations of Code Recommender Systems: A Reality Check". In: *Proc. 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. ACM, 2016.

[193] *pyaes*. `https://github.com/ricmoo/pyaes`.

[194] *PyCA-TLS*. `https://github.com/pyca/tls`.

[195] *PyCryptodome*. `http://pycryptodome.readthedocs.io`.

[196] *pycryptopp*. `https://tahoe-lafs.org/trac/pycryptopp`.

[197] *pyDes*. `https://github.com/toddw-as/pyDes`.

[198] *PyMe*. `http://pyme.sourceforge.net`.

[199] *PyNaCl*. `https://pynacl.readthedocs.io/en/latest`.

[200] *pyOpenSSL*. `http://www.pyopenssl.org/en/stable`.

[201] *pysodium*. `https://github.com/stef/pysodium`.

[202] Zhengyang Qu, Vaibhav Rastogi, Xinyi Zhang, Yan Chen, Tiantian Zhu, and Zhong Chen. "AutoCog: Measuring the Description-to-permission Fidelity in Android Applications". In: *Proc. 21st ACM Conference on Computer and Communication Security (CCS'14)*. ACM, 2014.

[203] Sai Ramanan. *The Top 10 Security Breaches Of 2015*. `http://www.forbes.com/sites/quora/2015/12/31/the-top-10-security-breaches-of-2015/#7a67d9d5694f`. Last visited: 10/18/2020. Dec. 2015.

[204] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. "DroidChameleon: evaluating Android anti-malware against transformation attacks". In: *Proc. 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS'13)*. ACM, 2013.

[205] B Reaves, N Scaife, A Bates, and P Traynor. "Mo(bile) Money, Mo(bile) Problems: Analysis of Branchless Banking Applications in the Developing World". In: *Proc. 24th Usenix Security Symposium (SEC'15)*. USENIX Association, 2015.

[206] Elissa M. Redmiles, Sean Kross, and Michelle L. Mazurek. "How I Learned to Be Secure: A Census-Representative Survey of Security Advice Sources and Behavior". In: *Proc. 23nd ACM Conference on Computer and Communication Security (CCS'16)*. ACM, 2016.

[207] Rob Reeder, E Cram Kowalczyk, and Adam Shostack. "Helping engineers design NEAT security warnings". In: *Proc. 7th Symposium on Usable Privacy and Security (SOUPS'11)*. ACM, 2011.

[208] Karen Renaud, Melanie Volkamer, and Arne Renkema-Padmos. "Why Doesn't Jane Protect Her Privacy?" In: *Proc. on Privacy Enhancing Technologies 2014 (PETS'14)*. Springer, 2014.

[209] Ronald L Rivest, Adi Shamir, and Leonard Adleman. "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems". In: *Communications of the ACM* 21.2 (1978), 120—126.

[210] Franziska Roesner and Tadayoshi Kohno. "Securing Embedded User Interfaces: Android and Beyond". In: *Proc. 22nd Usenix Security Symposium (SEC'13)*. USENIX Association, 2013.

[211] Franziska Roesner, Tadayoshi Kohno, Alexander Moshchuk, Bryan Parno, Helen J. Wang, and Crispin Cowan. "User-Driven Access Control: Rethinking Permission Granting in Modern Operating Systems". In: *Proc. 33rd IEEE Symposium on Security and Privacy (SP'12)*. IEEE, 2012.

[212] Scott Ruoti, Jeff Andersen, Scott Heidbrink, Mark O'Neill, Elham Vaziripour, Justin Wu, Daniel Zappala, and Kent Seamons. "'We're on the Same Page": A Usability Study of Secure Email Using Pairs of Novice Users". In: *Proc. CHI Conference on Human Factors in Computing Systems (CHI'16)*. ACM, 2016.

[213] Scott Ruoti, Nathan Kim, Ben Burgon, Timothy Van Der Horst, and Kent Seamons. "Confused Johnny: when automatic encryption leads to confusion and mistakes". In: *Proc. 9th Symposium on Usable Privacy and Security (SOUPS'13)*. ACM, 2013.

[214] C. Sadowski, J. v. Gogh, C. Jaspan, E. Söderberg, and C. Winter. "Tricorder: Building a Program Analysis Ecosystem". In: *Proc. 37th IEEE/ACM International Conference on Software Engineering (ICSE'15)*. IEEE, 2015.

[215] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo. "Are students representatives of professionals in software engineering experiments?" In: *Proc. 37th IEEE/ACM International Conference on Software Engineering (ICSE'15)*. IEEE Press, 2015.

[216] Jerome H Saltzer and Michael D Schroeder. "The Protection of Information in Computer Systems". In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.

[217] Samsung. *Knox*. `https://www.samsungknox.com`. Last visited: 11/13/2015.

[218] Riccardo Scandariato, James Walden, and Wouter Joosen. "Static analysis versus penetration testing: A controlled experiment". In: *Proc. 24th International Symposium on Software Reliability Engineering (ISSRE'13)*. IEEE, 2013.

[219] Stuart E Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer. "The Emperor's New Security Indicators". In: *Proc. 28th IEEE Symposium on Security and Privacy (SP'07)*. IEEE, 2007.

[220] Thomas Scheller and Eva Kühn. "Usability Evaluation of Configuration-Based API Design Concepts". In: *Proc. 1st International Conference on Human Factors in Computing and Informatics (SouthCHI'13)*. Springer, 2013.

[221] *scrypt*. `http://bitbucket.org/mhallin/py-scrypt`.

[222] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. "EROS: a fast capability system". In: *Proc. 17th ACM Symposium on Operating Systems Principles (SOSP'99)*. ACM, 1999.

[223] Shashi Shekhar, Michael Dietz, and Dan S. Wallach. "AdSplit: Separating Smartphone Advertising from Applications". In: *Proc. 21st Usenix Security Symposium (SEC'12)*. USENIX Association, 2012.

[224] Steve Sheng, Levi Broderick, Colleen Alison Koranda, and Jeremy J Hyland. "Why Johnny Still Can't Encrypt: Evaluating the Usability of Email Encryption Software". In: *Proc. 2nd Symposium on Usable Privacy and Security (SOUPS'06)*. ACM, 2006.

[225] *simple-crypt*. `https://github.com/andrewcooke/simple-crypt`.

[226] D. I. K. Sjoeberg, J. E. Hannay, O. Hansen, V. B. Kampenes, A. Karahasanovic, N. K. Liborg, and A. C. Rekdal. "A survey of controlled experiments in software engineering". In: *IEEE Transactions on Software Engineering* 31.9 (2005), pp. 733–753.

[227] Stephen Smalley and Robert Craig. "Security Enhanced (SE) Android: Bringing Flexible MAC to Android". In: *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.

[228] D. K. Smetters and Nathan Good. "How Users Use Access Control". In: *Proc. 5th Symposium on Usable Privacy and Security (SOUPS'09)*. ACM, 2009.

[229] Edward Smith, Robert Loftin, Emerson Murphy-Hill, Christian Bird, and Thomas Zimmermann. "Improving developer participation rates in surveys". In: *Proc. 6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE'13)*. IEEE, 2013. URL: http://people.engr.ncsu.edu/ermurph3/papers/chase13.pdf.

[230] 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: *Proc. 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015.

[231] Sooel Son, Daehyeok Kim, and Vitaly Shmatikov. "What Mobile Ads Know About Mobile Users". In: *Proc. 23rd Annual Network and Distributed System Security Symposium (NDSS'16)*. The Internet Society, 2016.

[232] SophosLabs. *Sophos 2020 Threat Report*. https://www.sophos.com/en-us/medialibrary/pdfs/technical-papers/sophoslabs-uncut-2020-threat-report.pdf. Dec. 2019.

[233] 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: *Proc. 21st Annual Network and Distributed System Security Symposium (NDSS'14)*. The Internet Society, 2014.

[234] M. Souppaya and K. Scarfone. *NIST Special Publication 800-124 Revision 1: Guidelines for Managing the Security of Mobile Devices in the Enterprise*. https://doi.org/10.6028/NIST.SP.800-124r1. June 2013.

[235] *Stack Overflow - Developer Survey Results*. https://insights.stackoverflow.com/survey/2017. visited. June 2017.

[236] Frank Stajano. "Pico: No More Passwords!" In: *Security Protocols XIX - 19th International Workshop*. Springer, 2011.

[237] Ryan Stevens, Clint Gibler, Jon Crussell, Jeremy Erickson, and Hao Chen. "Investigating User Privacy in Android Ad Libraries". In: *Proc. 2012 Mobile Security Technologies Workshop (MoST'12)*. IEEE, 2012.

[238] Christian Stransky, Yasemin Acar, Duc Cuong Nguyen, Dominik Wermke, Elissa M. Redmiles, Doowon Kim, Michael Backes, Simson Garfinkel, Michelle L. Mazurek, and Sascha Fahl. "Lessons Learned from Using an Online Platform to Conduct Large-Scale, Online Controlled Security Experiments with Software Developers". In: *Proc. 10th USENIX Workshop on Cyber Security Experimentation and Test (CSET'17)*. USENIX Association, 2017.

[239] Jeffrey Stylos and Brad A Myers. "The implications of method placement on API learnability". In: *Proc. 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT'08/FSE-16)*. ACM, 2008.

[240] Sufatrio, Darell J. J. Tan, Tong-Wei Chua, and Vrizlynn L. L. Thing. "Securing Android: A Survey, Taxonomy, and Challenges". In: *ACM Computing Surveys* 47.4 (2015), 58:1–58:45.

[241] Joshua Sunshine, Serge Egelman, Hazim Almuhimedi, Neha Atri, and Lorrie Faith Cranor. "Crying Wolf: An Empirical Study of SSL Warning Effectiveness." In: *Proc. 18th Usenix Security Symposium (SEC'09)*. USENIX Association, 2009.

[242] Mohammad Tahaei. "'I Don't Know Too Much About It': On the Security Mindsets of Future Software Creators". In: *Proc. 2019 ACM Conference on Innovation and Technology in Computer Science Education*. ACM, 2019.

[243] Mohammad Tahaei and Kami Vaniea. "A Survey on Developer-Centred Security". In: *Proc. 2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2019.

[244] The Internet Society. *Internet Society Global Internet Report 2015*. `http://www.internetsociety.org/globalinternetreport/assets/download/IS_web.pdf`. 2015.

[245] The OpenSSL Project. *OpenSSL: The Open Source toolkit for SSL/TLS*. `www.openssl.org`. 2003.

[246] *The Sodium crypto library (libsodium)*. `https://libsodium.org`.

[247] Daniel R. Thomas, Alastair R. Beresford, and Andrew Rice. "Security Metrics for the Android Ecosystem". In: *Proc. 5th ACM CCS Workshop on Security and Privacy in Mobile Devices (SPSM'15)*. ACM, 2015.

[248] T. Thomas, B. Chu, H. Lipford, J. Smith, and E. Murphy-Hill. "A study of interactive code annotation for access control vulnerabilities". In: *Proc. 2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'15)*. IEEE, 2015.

[249] Tyler W Thomas, Heather Lipford, Bill Chu, Justin Smith, and Emerson Murphy-Hill. "What Questions Remain? An Examination of How Developers Understand an Interactive Static Analysis Tool". In: *Proc. 2nd Workshop on Security Information Workers (WSIW'16)*. USENIX Association, 2016.

[250] Tyler W. Thomas, Madiha Tabassum, Bill Chu, and Heather Lipford. "Security During Application Development: An Application Security Expert Perspective". In: *Proc. CHI Conference on Human Factors in Computing Systems (CHI'18)*. ACM, 2018.

[251] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. "How Do Programmers Ask and Answer Questions on the Web?: Nier track". In: *Proc. 33rd IEEE/ACM International Conference on Software Engineering (ICSE'11)*. IEEE, 2011.

[252] Amos Tversky and Daniel Kahneman. "Judgment under Uncertainty: Heuristics and Biases". In: *Utility, Probability, and Human Decision Making: Selected Proceedings of an Interdisciplinary Research Conference, Rome, 3–6 September, 1973*. Springer Netherlands, 1975, pp. 141–162.

[253] Blase Ur, Patrick Gage Kelley, Saranga Komanduri, Joel Lee, Michael Maass, Michelle L Mazurek, Timothy Passaro, Richard Shay, Timothy Vidas, Lujo Bauer, Nicolas Christin, and Lorrie Faith Cranor. "How does your password measure up? The effect of strength meters on password creation." In: *Proc. 21st Usenix Security Symposium (SEC'12)*. USENIX Association, 2012.

[254] Kami E Vaniea, Emilee Rader, and Rick Wash. "Betrayed by updates". In: *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI'14)*. ACM, 2014.

[255] B Vasilescu, A Capiluppi, and A Serebrenik. "Gender, Representation and Online Participation: A Quantitative Study of StackOverflow". In: *Proc. 2012 International Conference on Social Informatics (SocialInformatics'12)*. IEEE, 2012.

[256] Timothy Vidas, Nicolas Christin, and Lorrie Faith Cranor. "Curbing Android Permission Creep". In: *Proc. Workshop on Web 2.0 Security and Privacy 2011 (W2SP'11)*. IEEE, 2011.

[257] John Viega, Matt Messier, and Pravir" Chandra. *Network Security with OpenSSL*. O'Reilly Media, 2002.

[258] Nicolas Viennot, Edward Garcia, and Jason Nieh. "A Measurement Study of Google Play". In: *Proc. 2014 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'14)*. ACM, 2014.

[259] Daniel Votipka, Desiree Abrokwa, and Michelle L Mazurek. "Building and Validating a Scale for Secure Software Development Self-Efficacy". In: *Proc. CHI Conference on Human Factors in Computing Systems (CHI'20)*. ACM, 2020.

[260] Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. "Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It". In: *Proc. 29th Usenix Security Symposium (SEC'20)*. USENIX Association, 2020.

[261] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. "Hackers vs. Testers: A Comparison of Software Vulnerability Discovery Processes". In: *Proc. 39th IEEE Symposium on Security and Privacy (SP'18)*. IEEE, 2018.

[262] Rui Wang, Luyi Xing, XiaoFeng Wang, and Shuo Chen. "Unauthorized Origin Crossing on Mobile Platforms: Threats and Mitigation". In: *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[263] Shaowei Wang, David Lo, and Lingxiao Jiang. "An Empirical Study on Developer Interactions in StackOverflow". In: *Proc. 28th Annual ACM Symposium on Applied Computing (SAC'13)*. ACM, 2013.

[264] Wei Wang and Michael W. Godfrey. "Detecting API Usage Obstacles: A Study of iOS and Android Developer Questions". In: *Proc. 10th Working Conference on Mining Software Repositories (MSR'13)*. IEEE, 2013.

[265] Wei Wang, Haroon Malik, and Michael W. Godfrey. "Recommending Posts Concerning API Issues in Developer Q&A Sites". In: *Proc. 12th Working Conference on Mining Software Repositories (MSR'15)*. IEEE, 2015.

[266] Yifei Wang, Srinivas Hariharan, Chenxi Zhao, Jiaming Liu, and Wenliang Du. "Compac: Enforce Component-level Access Control in Android". In: *Proc. 4th ACM Conference on Data and Application Security and Privacy (CODASPY'14)*. ACM, 2014.

[267] R Wash and E Rader. "Too Much Knowledge? Security Beliefs and Protective Behaviors Among United States Internet Users". In: *Proc. 11th Symposium on Usable Privacy and Security (SOUPS'15)*. USENIX Association, 2015.

[268] Xuetao Wei, Lorenzo Gomez, Iulian Neamtiu, and Michalis Faloutsos. "Permission evolution in the Android ecosystem". In: *Proc. 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, 2012.

[269] Joel Weinberger and Adrienne Porter Felt. "A Week to Remember: The Impact of Browser Warning Storage Policies". In: *Proc. 12th Symposium on Usable Privacy and Security (SOUPS'16)*. USENIX Association, 2016.

[270] Rodrigo Werlinger, Kirstie Hawkey, and Konstantin Beznosov. "An integrated view of human, organizational, and technological challenges of IT security management". In: *Information Management & Computer Security* 17.1 (2009), pp. 4–19.

[271] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Brad Reaves, Patrick Traynor, and Sascha Fahl. "A Large Scale Investigation of Obfuscation Use in Google Play". In: *Proc. 34th Annual Computer Security Applications Conference (ACSAC'18)*. ACM, 2018.

[272] Alma Whitten and J. D. Tygar. "Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0". In: *Proc. 8th Usenix Security Symposium (SEC'99)*. USENIX Association, 1999.

[273] Chamila Wijayarathna, Nalin Asanka Gamagedara Arachchilage, and Jill Slay. "Generic Cognitive Dimensions Questionnaire to Evaluate the Usability of Security APIs". In: *Proceedings of the 19th International Conference on Human-Computer Interaction*. Springer, 2017.

[274] Primal Wijesekera, Arjun Baokar, Ashkan Hosseini, Serge Egelman, David Wagner, and Konstantin Beznosov. "Android Permissions Remystified: A Field Study on Contextual Integrity". In: *Proc. 24th Usenix Security Symposium (SEC'15)*. USENIX Association, 2015.

[275] Shawn Willden. *Keyczar Design Philosophy*. `https://github.com/google/keyczar/wiki/KeyczarPhilosophy`. 2015.

[276] Jim Witschey, Olga Zielinska, Allaire Welk, Emerson Murphy-Hill, Chris Mayhorn, and Thomas Zimmermann. "Quantifying developers' adoption of security tools". In: *Proc. 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015.

[277] Lei Wu, Michael Grace, Yajin Zhou, Chiachih Wu, and Xuxian Jiang. "The Impact of Vendor Customizations on Android Security". In: *Proc. 20th ACM Conference on Computer and Communication Security (CCS'13)*. ACM, 2013.

[278] Glenn Wurster and P C van Oorschot. "The developer is the enemy". In: *Proc. 2008 New Security Paradigms Workshop (NSPW'08)*. ACM, 2008.

[279] Tim Wyatt. *Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild*. `https://blog.lookout.com/blog/2010/12/29/geinimi_trojan/`. Last visited: 11/06/15. Dec. 2010.

[280] Jing Xie, Bill Chu, Heather Richter Lipford, and John T Melton. "ASIDE: IDE support for web application security". In: *Proc. 27th Annual Computer Security Applications Conference (ACSAC'11)*. ACM, 2011.

[281] Jing Xie, Heather Richter Lipford, and Bill Chu. "Why do programmers make security errors?" In: *Proc. 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)*. IEEE, 2011.

[282] Yu Xie and Daniel Powers. *Statistical Methods for Categorical Data Analysis*. Emerald Publishing, 2008.

[283] Luyi Xing, Xiaorui Pan, Rui Wang, Kan Yuan, and XiaoFeng Wang. "Upgrading Your Android, Elevating My Malware: Privilege Escalation Through Mobile OS Updating". In: *Proc. 35th IEEE Symposium on Security and Privacy (SP'14)*. IEEE, 2014.

[284] Rubin Xu, Hassen Saïdi, and Ross Anderson. "Aurasium: Practical policy enforcement for android applications". In: *Proc. 21st Usenix Security Symposium (SEC'12)*. USENIX Association, 2012.

[285] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. "Helping Johnny to Analyze Malware: A Usability-Optimized Decompiler and Malware Analysis User Study". In: *Proc. 37th IEEE Symposium on Security and Privacy (SP'16)*. IEEE, 2016.

[286] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazières. "Securing Distributed Systems with Information Flow Control". In: *Proc. 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI'08)*. USENIX Association, 2008.

[287] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. "Towards Automatic Generation of Security-Centric Descriptions for Android Apps". In: *Proc. 22nd ACM Conference on Computer and Communication Security (CCS'15)*. ACM, 2015.

[288] Yinqian Zhang, Fabian Monrose, and Michael K Reiter. "The Security of Modern Password Expiration: An Algorithmic Framework and Empirical Analysis". In: *Proc. 17th ACM Conference on Computer and Communication Security (CCS'10)*. ACM, 2010.

[289] Wu Zhou, Xinwen Zhang, and Xuxian Jiang. "AppInk: Watermarking Android Apps for Repackaging Deterrence". In: *Proc. 8th ACM Symposium on Information, Computer and Communication Security (ASIACCS'13)*. ACM, 2013.

[290] Wu Zhou, Yajin Zhou, Michael Grace, Xuxian Jiang, and Shihong Zou. "Fast, Scalable Detection of "Piggybacked" Mobile Applications". In: *Proc. 3rd ACM Conference on Data and Application Security and Privacy (CODASPY'13)*. ACM, 2013.

[291] Wu Zhou, Yajin Zhou, Xuxian Jiang, and Peng Ning. "Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces". In: *Proc. 2nd ACM Conference on Data and Application Security and Privacy (CODASPY'12)*. ACM, 2012.

[292] Xiaoyong Zhou, Yeonjoon Lee, Nan Zhang, Muhammad Naveed, and XiaoFeng Wang. "The Peril of Fragmentation: Security Hazards in Android Device Driver Customizations". In: *Proc. 35th IEEE Symposium on Security and Privacy (SP'14)*. IEEE, 2014.

[293] Yajin Zhou and Xuxian Jiang. "Detecting Passive Content Leaks and Pollution in Android Applications". In: *Proc. 20th Annual Network and Distributed System Security Symposium (NDSS'13)*. The Internet Society, 2013.

[294] Yajin Zhou and Xuxian Jiang. "Dissecting Android Malware: Characterization and Evolution". In: *Proc. 33rd IEEE Symposium on Security and Privacy (SP'12)*. IEEE, 2012.

[295] 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: *Proc. 19th Annual Network and Distributed System Security Symposium (NDSS'12)*. The Internet Society, 2012.

[296] Yajin Zhou, Xinwen Zhang, Xuxian Jiang, and Vincent W. Freeh. "Taming Information-stealing Smartphone Applications (on Android)". In: *Proc. 4th International Conference on Trust and Trustworthy Computing (TRUST'11)*. Springer, 2011.

[297] Philip Zimmermann. *PGP Version 2.6.2 User's Guide*. `ftp://ftp.pgpi.org/pub/pgp/2.x/doc/pgpdoc1.txt`. Oct. 1994.

Curriculum Vitae