UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Human Factors in Open Source Security

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

dem Fachbereich Elektrotechnik, Informatik und Mathematik
der Universität Paderborn
vorgelegt von

Marcel Fourné
geboren in Eschweiler

Bochum, 2023-12-12

Email: email@marcelfourne.de
Website: https://marcelfourne.de/

First edition: 2023-12-12
Version: 2024-04-12

Vom Fachbereich Elektrotechnik, Informatik und Mathematik
der Universität Paderborn als Dissertation am
2023-12-12 angenommen.

Erstgutachterin: Prof. Dr. Yasemin Acar

Zweitgutachter: Prof. Dr. Sascha Fahl

Tag der mündlichen Prüfung: Freitag, 2024-01-26

Beisitzer der mündlichen Prüfung:

1. Prof. Dr. Peter Schwabe
2. Prof. Dr.-Ing. Juraj Somorovsky
3. Dr. Harald Selke

Erklärung

Ich versichere, dass ich meine Dissertation

"Human Factors in Open Source Security"

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

Software security research has begun to formalize attacks and defenses against commonly deployed software, even the most optimized cryptography [38]. Newer programming languages make memory access violations and other programming errors rarer, but still those problems are common in yearly vulnerability rankings[1]. It seems there is a gap between what is possible in the most optimized and security-hardened software, and what is used by most people. As mentioned previously [4], human factors, and developers in particular, seem to be a weaker link in software security.

Most software users have to consider an important question: How to choose which developers—and by extension their software—they trust to handle our most precious secrets, to secure the integrity of our computing environment now that almost no-one can live without it.

We will use the scientific "we" to mark our research clearly as group efforts. In order to help developers of critical software, we set out to identify some of their biggest challenges. We did qualitative, quantitative, and mixed method user studies in a field that is a lot smaller than the population of all developers. Cryptographers, open source maintainers and developers are driven by strong opinions and are not afraid to voice them, as long as they are not talking to complete strangers. With this dissertation, we want to provide hard, scientific data, benefitting from our contacts and backgrounds.

The outline for this dissertation will be to first describe an important problem that is the main topic of two of our publications: usability of cryptographic Constant Time analysis, the tools for it, and the effects it has on the security of the bigger software ecosystem. We will follow up with another publication which focuses more on the general security and quality aspect of software packages in the software ecosystem when viewed from the perspective of a software supply chain. The angle is a frame around this dissertation with a viewpoint article about an agenda for doing more research on the ecosystem that was also published, but we will separate its motivation and recommendations into our introduction and conclusion of this dissertation.

The main findings of this dissertation are that most developers would prefer to have software that is secure by default and can be built deterministically, does what they expect it to do, can be analyzed easily, and fulfills all security requirements. Alas, most of them don't get paid to do that and even if they are, they are lack the numbers to fix all problems. Academic research highlights possible solutions, but its outputs are more like proof-of-concept prototypes instead of long-time maintainable and effective software.

Our work finds that thinking about the work that hobbyist software developers and

[1] See: https://cwe.mitre.org/top25/

maintainers do, and usability for these specific types of requirements is necessary for them to keep making our most basic software dependencies work for another forty years.

With this thesis in hand, we have identified two core groups who maintain software necessary for the core of our modern infrastructure and some of their most pressing needs. These should be adressed to improve the security and reliability of all our infrastructure. From these, we can speculate on how to bring their benefits directly to less knowledgeable users and make new software start from better laid security and usability foundations.

Zusammenfassung

Die Software-Sicherheitsforschung formalisiert heute sowohl Angriffe als auch Verteidigungsmaßnahmen gegen die meistgenutzte Software, selbst die am stärksten optimierte Kryptographie [38]. Neuere Programmiersprachen schützen die Entwickler:innen sowie die Nutzer:innen besser vor Speicherzugriffsfehlern so wie anderen Programmierfehlern, aber trotzdem führen diese immer noch die jährlichen Ranglisten der am weitesten verbreiteten Schwachstellen an[2]. Die Sicherheit der am stärksten optimierten Software wurde manchmal auch am weitesten gehärtet. Es scheint, dass es eine Lücke gibt zwischen der am stärksten optimierten und geschützten Software und der Software die von den meisten Benutzer:innen genutzt wird. Wie schon vormals erwähnt [4], scheint der "Faktor Mensch", und Entwickler:innen im besonderen, ein schwächeres Glied der Softwaresicherheit zu sein.

Die meisten Software-Nutzer:innen müssen heute über eine Frage entscheiden: Wie wählen wir aus, welchen Entwickler:innen—und damit auch ihrer Software—wir vertrauen wollen, wenn es um unsere wichtigsten Geheimnisse geht, um die Absicherung der Integrität unserer Computer, ohne die heute kaum jemand leben kann?

Im folgenden der Dissertation wird das wissenschaftliche "wir" genutzt, um die Forschung klar als gemeinschaftlichen Aufwand zu kennzeichnen. Unser Ziel war es herauszufinden, welche großen Probleme Entwickler:innen für die kritischste Software haben und wie wir ihnen helfen können. Wir haben dabei Studien durchgeführt mit qualitativen, quantitativen, sowie Mischverfahren in einem Bereich, der sehr viel kleiner ist als der allgemeiner Softwareentwickler:innen. Kryptograf:innen, Open-Source-Maintainer und Entwickler:innen sind oft angetrieben von starken Meinungen und haben keine Angst, diese auch zu äußern, sofern sie nicht von in der Community unbekannten Menschen gefragt werden.

Im Rahmen dieser Dissertation werden wir zuerst ein Problem beschreiben, das der Fokus zweier unserer Publikationen ist—Benutzbarkeit von kryptografischer Constant-Time-Analyse und entsprechenden Werkzeugen sowie den Effekten, die dies auf die Sicherheit des größeren Software-Ökosystems als Ganzes hat. Darauf folgt eine weitere Publikation, die mehr darauf fokussiert ist, die allgemeine Sicherheit und Qualität von Softwarepaketen zu steigern im ganzen Software-Ökosystem, wenn wir es unter dem Blickwinkel einer Softwarelieferkette betrachten. Diese Sichtweise bildet das Umfeld dieser Dissertation durch einen Artikel, der eine Agenda für mehr Erforschung des ganzen Ökosystems fordert und ebenfalls wissenschaftlich veröffentlicht wurde. Wir teilen den Artikel auf nach Motivation und Empfehlungen in Einleitung und abschließende Zusammenfassung dieser Dissertation.

[2]Siehe: https://cwe.mitre.org/top25/

Die hauptsächlichen Erkenntnisse sind, dass die meisten Entwickler:innen standardmäßig sichere Software bevorzugen, die deterministisch gebaut wird, macht, was von ihr erwartet wird, einfach analysiert werden kann und alle Sicherheitsanforderungen erfüllt. Die Meisten von ihnen werden jedoch nicht dafür bezahlt dies alles herzustellen und selbst wenn, dann ist nicht genug Personal vorhanden um alle Probleme zu beheben. Die akademische Forschung zeigt mögliche Lösungen auf, aber ihre Ergebnisse sind eher Prototypen anstatt langfristig wartbare Software, die bei den täglichen Arbeiten hilft.

Unsere Forschungsergebnisse zeigen, dass es für gute Benutzbarkeit bei diesen Anforderungen notwendig ist über die Arbeit von Hobby-Softwareentwickler:innen nachzudenken und was sie benötigen, um weiterhin die grundlegende Software und alles von ihr Abhängende für weitere vierzig Jahre funktionsfähig zu erhalten.

Mit dieser Dissertation haben wir zwei Gruppen von Menschen identifiziert, die die notwendige Software instandhalten für den Kern unserer modernen Infrastruktur, sowie ein paar ihrer dringendsten Bedürfnisse und Wünsche zur Verbesserung bei den jeweiligen Tätigkeiten. Diese sollten angegangen werden um unser aller Infrastruktur sicherer und zuverlässiger zu machen. Basierend darauf können wir spekulieren, wie wir all die Vorteile auch an weniger technisch versierte Benutzer:innen bringen können und neue Software von den besseren Grundlagen aus aufbauen.

Foreword

Without the support of my advisors Peter Schwabe, Gilles Barthe and Yasemin Acar, the research for this dissertation would not have been possible.

I thank my family for their support in getting me to work hard and learn as much as possible, even against opposition.

I would like to thank Sascha Fahl for feedback, hints and insight into my research and discussions on the work for our papers and otherwise; my other coauthors Ján Jančár, Daniel De Almeida Braga, Mohamed Sabt, Pierre-Alain Fouque, Dominik Wermke, Noah Wöhler, Jan H. Klemmer and William Enck for all the hours of myriad work they put into our papers; Basavesh Ammanaghatta Shivakumar, Kai-Chun Ning, Miguel Quaresma, Sunjay Cauligi, Tiago Filipe Azevedo Oliveira, Roberto Blanco for discussions about all kinds of interesting topics and giving support not just through good coffee; Vincent Bert Hwang for keeping the office in a state of open nerdery; Amber Sprenkels and Karolin Varner for building an awesome support network; Thom Wiggers for motivating me to build my own thesis infrastructure; Łukasz Chmielewski for a shared drink or two and some good food (for thought); Holger Pieta, Olaf Rühenbeck, Jochim Rolf Selzer, Eugen Keller, Jan Benedikt Lieven, Nina Kiel, and Leonie Paltz for their support, being good friends, and being around for all sorts of discussions over days into long evenings; the RWC community and all the discussions and encouragement validating my research and other work; Nicky Mouha for showing us around Tokyo and getting us to the best tiny bar there, while still maintaining a good in-depth discussion on program verification and testing.

Without all the other people, it would not have been feasible for me to finish this dissertation. I want to thank all of you!

Contents

1. Introduction

Disclaimer

This thesis is based on three previous publications. Two of them were written with me as the main author. The content of this introduction and the conclusion in Chapter 6 was expanded and adapted from a previously peer-reviewed and published article titled "A Viewpoint on Human Factors in Software Supply Chain Security: A Research Agenda.", for the Special Issue on Secure Software Supply Chain of IEEE Security & Privacy magazine. This writing was conducted as a team with my co-authors Dominik Wermke, Sascha Fahl, and Yasemin Acar; this chapter therefore uses the academic "we". We developed the research agenda and reviewed the literature jointly as a team, incorporating feedback from Henrik Plate and Laurie Williams whom we want to thank for lively discussions and valuable feedback on this paper.

I think science's main purpose should not be to hide the truth, but to show it in a way that is as easy to understand as the scientists can make it. As per Schneier's Law, "[a]nyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break." [328] An alternative phrasing can be misattributed to Albert Einstein, better attributed to Richard Feynman who said "[i]f you can't explain it simply, you don't understand it well enough." I claim we do not understand our computers and their computations well enough.

1.1. Software Security Landscape

Attacks against software become more frequent [1]. More widely deployed software means a single vulnerability needs to be patched in a lot more places.

Software security today is a long way from the early nineties, where the buffer overflow [280] was only secret knowledge. Memory safe programming languages have grown to be more prominent [77]. The most security sensitive software is now used f.e. in online-banking, where high values are protected against outside attackers just listening in on any internet connection. Still, hacking occurs with destructive repercussions [122]. Even modern software suffers from bug classes that were discovered more than twenty years—the buffer overflows were published in 1996 [280]. But aside from these problems, higher security requirements for software and different attacks require different defenses.

When new attacks are found, all new and existing software may be vulnerable. For cryptography, one very public example were the Spectre [210] and Meltdown [239] vulnerabilities in 2018, which cost processor manufacturers an estimated 18 billion

US dollars [2]. These are only technical costs, since then every implementation of cryptography needs to defend against Spectre-style attacks, in principle.

With this thesis, we want to look into the problems which inhibit developers from defending their software against these known attacks with also published defenses. We want to inspect their problems and highlight what is missing to deploy the sometimes academic defenses in more real-world software installations, to provide better overall security for end users, some of which are the developers in question themselves.

Much of the software that needs to be defended against attacks is not from one commercial entity [287], but from a group of—only sometimes paid [23] for their efforts—volunteer developers. Hence we have to look into the environment which begets the software used in many places in the background [287] first.

1.2. Why Open Source?

In this thesis, we assume that Free/Libre/Open Source Software (OSS) is being used in most software projects, open and closed alike [287]. This trend increased over the years to a high point today [179], consequently we need to ask ourselves how to secure projects that are developed by different groups of developers that interact in different ways, if at all [392].

In the sixties, after the women programmers before [185], programming became a topic for highly educated, highly paid, and highly privileged [185] mainframe programmers with access to computing resources [185]. In the eighties to nineties this changed to include ever more "home programmers", hackers, and other people who are now referred to collectively as parts of the hacker culture and OSS movement [98]. The difference between these decades is widening access to cheaper, readily available computing resources—Personal Computers [175]. This has led to more people being able to develop software at home, in their spare time, for fun, personal, or sociopolitical reasons [183]. These communities started to develop their own operating systems, which were the basis for our modern Internet, with the advent of a widely used open-source TCP/IP implementation for the BSD operating system [232]. These communities have impact on deployed software, commercial and non-commercial alike, but the educational, financial and privilege background can be a lot more diverse in these communities [137], which can lead to different outcomes in security as well [390].

We can always make something more secure if we don't have to change what makes it work [330], like its trust dynamics [392]. We assume OSS is valued for its generally high quality and low opportunity cost [32, 51]. If we change any of the factors that make OSS work in the first place, we might destroy what allowed it to thrive and bring benefit to the software ecosystem [252], with some of its participants not paying forward even if they could afford to [26]. This is not necessarily a detriment, though, since many developers work on OSS in their own free time, as a hobby [341]. It is these hobbyist workers that have to be seen as an integral part, since they founded the whole ecosystem as it is today [98].

Making the software ecosystem more secure by driving up opportunity costs [336], changing how developers interact with their development environment [269], who they trust [114] and how they establish trust [365] has to be done very carefully [392]. As it is today, some developers know and use cryptography to secure their software [114], and we want to look into the quality of that cryptographic software. This applies only to a very small subset of developers, but they are the most important ones [17].

Software packages which have a large set of dependencies [93]—direct and indirect [107]—led to the view to call this a supply chain [242], like the one for other products. This software supply chain needs to be secured as well, and is riddled with transitive dependencies and therefore transitive trust relationships between developers who build something [107], which end users often only see the last development party of, becoming vulnerable to dependency confusion attacks [64]. While software releases may be signed more readily, when single commits in distributed repositories are the main mode of interaction [101], but the prevalence of unsigned commits is not quantified as of yet, so a lot of untrusted changes may end up in our final software we download [331]. Even in the best case, we want our cryptography used for signing software to be implemented securely [395]—what that means will need some more background information, but it is non-trivial for most developers who are not well versed on the topic [180].

Externalized trust relationships are still difficult [215], since the more removed you are from the person whose code you want to trust, the less you know about them or their coding practices [75]. This externalization of trust can be even harder if most of our infrastructure is financed by industry fundamentally at odds with the interests of software developers and users [408]. As stated by Rogaway, cryptographers are motivated to help people anyway, since cryptography was never a politically neutral field [317]. This notion makes cryptographic engineering akin to OSS development, which can be done out of different motivations including moral reasons, but there are many differences as well.

We will now look into those differences.

1.3. Cryptographic Engineering differs from general Software Engineering

Cryptographic developers implement cryptographic algorithms, using tools specific to cryptographic engineering. The security requirements of cryptographic code are different and higher than most other, more general code [395]. The analysis of cryptographic code can be harder due to more manual optimization being done by expert cryptographic engineers as well as many domain specific other details, like the implementations not making secrets inadvertently public [38]. Sometimes it is desirable enough to formally, mathematically prove the correctness of cryptographic algorithms, if at all possible. This is good practice, but proving an implementation correct is a lot harder and proven code may not be good in general [182].

The burden of proving, analyzing, testing, or even just reasoning about the security of their cryptographic implementations by each individual developer in a team, may sometimes get lost in distributed OSS projects [199].

Helping these developers write better code more easily benefits the security of their software implementations and of the infrastructure based on those implementations. We will focus on these developers from now on.

1.4. Why focus on Developers?

Securing the software supply chain requires that we recognize the importance of individual developers [95, 7]. While securing dependencies and build systems is necessary, recent attacks have shown that developers are a link in the chain that is commonly attacked successfully. Therefore, a comprehensive approach that considers the human factor is crucial for effective software supply chain security.

OSS and cryptographic software developers are highly skilled in their own right, but they use software like everyone else. Their understanding of problems does not make them immune to bad usability of software, which in their fields is sometimes from research prototypes that were not meant for public, long-time consumption and maintenance [199].

1.5. OSS and Supply Chain Security

Human factors, and especially developers, play an important role in securing the Software Supply Chain (SSC) [95]. SSC vulnerabilities pose significant risks to organizations, historically being exploited by threat actors who target unpatched systems with known vulnerabilities [221]. Exploits targeting vulnerabilities like Heartbleed [128] and more recently Log4Shell [134] have highlighted weaknesses in both commercial and open-source software, affecting both private and government enterprises and billions of end users. More recently, attackers directly exploited the SSC structure by targeting upstream dependencies and build systems to inject malicious code into downstream software, like in the security incident at SolarWinds.[1]

Several steps to address dependency and build chain problems have been proposed—updating and using trusted dependencies, Software Bill of Materials (SBoMs), securing the build process, and more industry participation [131]. However, securing these dependencies and build systems will likely not thwart all attacks: Recent, headline-making attacks involved breaches of the SSC through one of its largest attack surfaces—individual developers. In January 2023, CircleCI disclosed that a cybercriminal had used malware on a CircleCI engineer's laptop to steal a valid, two-factor authentication-backed SSO session, allowing the attacker to execute session cookie

[1] https://www.mandiant.com/resources/blog/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor

theft and impersonate the employee, gaining access to a subset of production systems.[2] In February 2023, password manager LastPass reported that a hacker stole corporate and customer data by infecting an employee's personal computer with key logger malware, giving them access to the company's cloud storage and resulting in the theft of source code and customer vault data.[3]

These incidents highlight that each individual developer participating in the SSC may have different technology and knowledge stacks, and humans making decisions about security and trust can put the SSC at risk [392]. We explore both how to empower stakeholders to secure the SSC by considering human factors and reducing developer overwhelm, and also how to decrease the attack surface of individual software developers.

1.6. Highly Skilled Software Developers still have usability issues

We see that highly specialized and skilled developers are needed to secure the SSC. We also see that some of the most impactful software in use is made, at least in parts, by volunteers, who do not have a common organizationally enforced minimum standard of institutional knowledge, funding, or onboarding into projects.

These developers need to be supported in their work, so we can raise the security bar for everyone who directly or indirectly depends on their software. Since bad usability of specialized research or other special use software targeted at supporting these developers—software users in that role—harms adoption of that supporting software, we will look at case studies of usability issues in this thesis.

1.7. Thesis Statement

The purpose of this thesis is to explore the impact of usability problems and ways to help developers secure not end-user facing software against attacks, but the infrastructure which all other software is depending on. We will view these specialized developers as users of software themselves, and gain insights as well as offer advice with the lens of usable security, informed by expert matter in these highly specialized topics.

We assume that all developers in these specialized fields—cryptography and supply chain security/build tooling—are highly motivated and highly informed about possible problems in their area, but not necessarily in all possible tools and defenses against those problems. We want to find out how to bridge gaps between these developers and the developers of security automation tools analysing, mitigating, or completely

[2] https://circleci.com/blog/jan-4-2023-incident-report/
[3] https://blog.lastpass.com/2023/03/security-incident-update-recommended-actions/

eliminating specific classes of problems not widely known in the general developer space, which is itself a specialized field among the whole population.

Recognition: For work done in preparation of this thesis, we received a distinguished paper award for the paper "Committed to Trust: A Qualitative Study on Security and Trust in Open Source Software Projects" [392] (see https://www.ieee-security.org/TC/SP2022/awards.html).

1.8. Contributions

For a full list of papers, see Chapter D.

The research that contributed to this thesis has been published as conference papers. This section will provide a summary of the publications that contributed to this thesis, where they were published, and the contributions of each author and how they contributed. Like other scientific work, it would not have been possible to create these works without collaborative effort and significant contributions of co-authors. In this list, § marks the main author, * marks an author with a significant contribution, and $^{+}$ denotes a supervising author with signifcant contribution. The publications have been mildly edited and set in context where appropriate for inclusion in this thesis. The publications contributing to this thesis are the following:

"They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks

This peer-reviewed paper contributes to Chapter 3 of this thesis. We research the current landscape of opinions and practices around cryptographic constant-time verification tools for use by cryptographic library developers

Authors: Jan Jancar§, Marcel Fourné*, Daniel De Almeida Braga*, Mohamed Sabt*, Peter Schwabe^{+}, Gilles Barthe^{+}, Pierre-Alain Fouque^{+}, and Yasemin Acar^{+}.

Published at the 43rd IEEE Symposium on Security and Privacy 2022.

Contributions to this paper: The initial idea was developed by myself with Jan Jancar, Peter Schwabe, Gilles Barthe, and Yasemin Acar. All authors reviewed the literature, conducted the study, and jointly analyzed the data and co-wrote the paper for publication.

Paper Summary Developers of cryptographic software libraries care for software that is important for the security of the whole software ecosystem, which is itself important for all our current industries. Small security problems in those libraries can cause large damage. Attackers can be highly motivated and one class of attacks that is still prevalent are timing attacks. Analysis tools for finding such potential vulnerabilities that can later potentially be exploited in an attack are published, however we found out by asking 44 cryptographic software library developers that their adoption of these tools is small. We asked about their reasons for this, as well as experiences of trying to use

the tools, in order to compile a list of recommendations for different target audiences of the paper.

It's like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security

This peer-reviewed paper contributes to Chapter 5 of this thesis. We research the impact and current status of reproducible builds in the context of build automation and determinism of its results for the security of the SSC.

Authors: Marcel Fourné[§], Dominik Wermke[*], William Enck[+], Sascha Fahl[+], and Yasemin Acar[+].

Published at the 44th IEEE Symposium on Security and Privacy 2023.

This work was mentioned in the monthly report of the Reproducible-Builds.org project at https://reproducible-builds.org/reports/2023-06/.

Contributions to this paper: I developed the main idea with Yasemin Acar, with input from William Enck and Dominik Wermke. The instrument was developed by Yasemin Acar and myself. I conducted the interviews, with the help of Dominik Wermke. I lead the initial round of coding, independently verified by Dominik Wermke, followed by joint affinity diagramming with Yasemin Acar, Dominik Wermke, and Sascha Fahl. I lead writing the paper for publication, with support from all other authors.

Paper Summary

One old attack against software security is the Trusting Trust attack, published during Ken Thompsons 1983 Turing award lecture [354]. Using this attack, a backdoor can be implemented in the binaries that are compiled from unmodified source code of programs, proliferating the backdoor across the whole software ecosystem. Finding this backdoor is hard and only in 2006 David A. Wheeler showed a defense against it, which requires software to be built reproducibly by default, giving bit-by-bit same artifacts. Historically, this was simpler with trivial programs, or older programs made to be simple. Today, there is an effort to achieve this with more modern software, even bootstrapping the whole software ecosystem from an inspectable, clean state to a functional SSC. The security benefits are highlighted through recent attacks on different SSCs, but the main adopters of reproducible builds are projects in the open source community, which also spend the most effort to bring the topic forward. We interviewed 24 developers on current problems and solutions for those problems, compiled a list of recommendations for different target audiences of the paper and show a positive outlook of how far the projects have already come.

A Viewpoint on Human Factors in Software Supply Chain Security: A Research Agenda.

This peer-reviewed article contributes to the introduction and conclusion 6 of this thesis. We identify focus areas promising research problems in supply chain security and usable security for developers and other users.
Authors: Marcel Fourné[§], Dominik Wermke[*], Sascha Fahl[+], and Yasemin Acar[+].
Published in the Special Issue on Secure Software Supply Chain of IEEE Security & Privacy magazine, issue 21.6.
This work was mentioned in the monthly report of the reproducible-builds.org project at https://reproducible-builds.org/reports/2023-11/.
Contributions to this paper: All authors reviewed the literature, developed future research directions, and finalized the paper after an initial draft by me.

Article Summary After conducting the research for Chapter 5, we decided to compile a list of open research problems in the area of software supply chain security (SSCS), where the lens of usable security was still missing to include developers and (expert) users as competent but under-informed practitioners in their fields. We draft different topics for future research projects and outline the problem areas as well as expected benefits for conducting those projects. We provide an overview of how these topics integrate into the main area of SSCS and cite reasons for beginning these projects sooner rather than later in our current software ecosystem crisis.

"These results must be false": A usability evaluation of constant-time analysis tools

This peer-reviewed paper contributes to Chapter 4 of this thesis. We follow up on the previous cryptographic constant-time analysis tool proliferation paper 1.8 by evaluating quantitatively and qualitatively how much certain properties thwart use and adoption of the tools by inexperienced, but highly qualified developers that might start working on cryptographic software libraries.
Authors: Marcel Fourné[§], Daniel De Almeida Braga[*], Jan Jancar[*], Mohamed Sabt[*], Peter Schwabe[+], Gilles Barthe[+], Pierre-Alain Fouque[+], and Yasemin Acar[+].
Accepted for publication at the 33rd USENIX Security Symposium 2024.
Contributions to this paper: The initial idea was developed by myself with Jan Jancar, Peter Schwabe, Gilles Barthe, and Yasemin Acar. We conducted the study with our other co-authors and jointly analyzed the data and co-wrote the paper for publication.

Paper Summary

Since the work of finding out why cryptographic software library developers do not pervasively use cryptographic constant-time analysis tools was not finished by giving concrete, objectively and numerically evaluated reasons for abandoning those tools,

we followed up the first paper with a test setup where we recruited eligible students of European top tier cryptographic and IT-security study programs who did not have prior experience working on production quality cryptographic libraries, but have all necessary skills to do so. 24 of them were given tasks of different difficulty levels for evaluating the ease of use of different cryptographic constant-time analysis tools chosen by us to represent the most common classes of tools being published in research literature. After this, they were given one open source cryptographic library to analyze with the same tool, and another library with a new tool, to compare potential familiarization effects with different tools. Our test setup is public, to make installation of the tools as well as reproduction of the study easier or even possible at all. At the end, we compiled the results, gave our expert opinions on the results, and provide recommendations based on our experimental findings for cryptographic constant-time analysis tool developers to enhance the usability of their research tools to be suitable for real-world adoption. We order the recommendations by the phase of an analysis and give our expert opinions on possible effort required and additional enhancements.

1.9. Related and Concurrent Work

The work in this dissertation supports the statements that (a) the security of the SSC depends on human factors concerning its developers, and (b) better usability of code-analysis tools for cryptographic software implementations benefits developers in helping to find implementation errors in their code. During the time we concluded our studies, other research groups explored several other research questions, either concurrently or after our work.

In 2020, Torres-Arias et al. published in-toto [358], a tool to check code provenance through auditing before publication of code artifacts. The code provenance through cryptographic signing of commits in git repositories available over the web—or the lack of availability to do so at the time—was researched [12] as well. The importance of bootstrapping trust for projects based on a community repository was investigated [371], noting problems for the SSC as well as currently established practices. This led to the design of a new cryptographic artifact signing system called sigstore [267], aimed at better developer usability when a centralized authority is used anyway, foregoing some of the older community projects that sometimes distrust such solutions. By using a fully centralized signing solution under a certificate authority, an even simpler system was published [251]. Online authentication for all users, following developer advice, got a look as well [209].

The deployment of software in the DevSecOps process, orchestrated by a centralized authority, got an initial NIST standard draft in 2023 [87]. Inspecting the security of third party dependencies using the Java ecosystem was investigated and a permission manager was introduced [24], as well the state of finding malicious Java packages in general [223]. This work was repeated for npm and PyPi as well [225], speculating about and giving numbers for detection of these [224]. In contrast, the Rust ecosystem

is more aimed at also enabling embedded software use of its software implementation, so its security state was looked into as well [334]. The quality of Software Bill of Materials (SBoMs), can vary, as one viewpoint article [357] speculated and prompted more research into. Another viewpoint article, concurrent to our own, speculates that the area of SSCS can have limited adoption of best practices through divergent nomenclature [249]. As mentioned before, an SoK paper about the state of SSCS was published to much acclaim [221]. Earlier, as its foundation, software for exploring the risks and potential attack surface of SSCS using OSS was presented [222], together with later developing the taxonomy based on it [222], as groundwork for the SoK paper. Also following the SoK, a lot of other papers emerged. These study the specifics of the security challenges in the open source SSC [391, 254], while others look into package managers and programming language ecosystems [377, 266, 324, 168, 261], statistics-based detection of attacks [202, 270], code review processes [195], just to name a few of the more prominent ones.

The general state of SSCS was also investigated [278] and systematized in a second SoK paper, identifying four stages of supply chain attacks that software is vulnerable to.

The practice of teams developing software and their approaches to security were compared [385] and workshops provided to enhance integration of a security culture. The cybersecurity self-efficacy of users got a literature review [55] and the general stance of employees towards IT security was investigated [250], again with an eye towards interventions. Expert users attitudes towards checking cryptographic signatures [244] found that 52% of these expert users could be made to fall for one of four signature spoofing attacks.

Introduced friction through security practices in organizations was a point of contention for security managers working to better the state of security in their organizations [187], and awareness about the need for it found as the main problem [186].

On the practicality of bringing cryptographic research into general usage [138], new research is slated for publication in 2024, similar to a previous study about general benefits of usable security research not getting into final products [169].

We hope that the results in this dissertation as well as other publications help developers to implement more usable and secure by default software, so the security of our whole SSC benefits in general, but also new research is inspired.

2. Background

After some motivational framing, we want to give a background into the topics around which this thesis gravitates. We want to explain why cryptographic engineering is a specialized discipline, built on a lot of prerequisite knowledge and practical experience, and why this is necessary for high-value software deployments to secure the roots of our current software ecosystem.

This chapter provides a framing on how to read the peer-reviewed background sections as they relate to each chapter thereafter, to give a notion on how this thesis' overarching theme connects back to the goal of securing the software we run on our computers, which is orders of magnitude in interdependent code being deployed in practice. We have to start with the big picture of software supply chains, and will then go into details on how each single part can be secured.

First we will introduce software supply chains, with notions of quality and trust as well as security problems and defenses against them. The Trusting Trust problem will be important and underpins infrastructure changes still necessary to deploy worldwide.

We will follow up with the shortest notion of cryptography, insofar as it is necessary to talk about what we want to secure with its use and what we need to ensure it is secured against, in the form of a still common type of side-channel attack, which code can theoretically defend against, but in practice is not implemented ubiquitously. Finding code parts not hardened against these attacks can be done via specialized code analysis, which we introduce next insofar as we want to highlight why it is not used in the development of each and every deployed cryptographic software.

The art of creating cryptographic software libraries—cryptographic engineering—will be introduced next and set in relation to its need for usable security, to highlight why we did our studies which comprise the main content of this thesis.

2.1. Software Supply Chains, Quality and Trust

Software that is built from different parts, each being a software project in itself, affords less control over each change in those parts. One model to make this work is so-called "blind" trust [113], where developers assume that each project will be managed in a good way, solving a problem for the developer depending that library in their own project and being fixed promptly against security vulnerabilities, which through updates then percolate through the whole chain of dependencies. This model also relies on cryptographic signatures of code artifacts, but we don't need to link every bit of source code to each developer. Instead, we trust that the code artifacts were

generated well. This model is not, however, very secure against attacks [221], due to many developers being burdened by time-constraints and maybe not knowing about potential security vulnerabilities. Even worse, some malicious actors can insert themselves and their code in this software supply chain, their code then being included into code artifacts. That is the reason we need to look into how our software is being made and what we actually execute on our computers. The quintessential attack is the "Trusting Trust" attack presented by Thompson in 1984 [354], where a specifically crafted backdoor is included in a compiler, so that every binary generated by that compiler includes the backdoor. Subsequently compiling the compiler source code with a backdoored compiler binary will include the backdoor, even if it is not present in the source code. This raises questions if our software supply chain is currently backdoored in a similar way. In 2005, David A. Wheeler presented a scientific paper which proposes a defense against this kind of attacks: "Diverse Double-Compiling" [393]. The notion behind this defense is that if two different compiler binaries—not necessarily from different compiler projects, but one of them has to be trusted to be free of backdoors—agree on the result of compiling source code, then we can also trust the previously untrusted compiler binary. For this, a special criterion needs to be true for the compilers: They must build the same source code under the same set of environmental variables bit-by-bit identically. The effort to get this as a default is called "Reproducible Builds" and is the topic of Chapter 5. The trust generated by being able to check how each binary is generated in this setup gives rise to trust relationships, where developers can be identified through cryptographic signatures of their source code and consumers of their work can decentrally check if the resulting code artifacts have been tampered with. This, however, is not current practice, as discussed in Chapter 5. The full bootstrap of a software distribution using a minimal trusted computing base is shepherded into an effort called "Bootstrappable Builds" and the GUIX distribution has achieved such a boostrap [272] from a 357-byte program, which is easy enough to be checked by hand by multiple people.

Checking compiled binaries for potentially malicious behaviour runs into problems with Rice's Theorem [313] which states that for a property p of a nontrivial formal language L, it is undecidable for a given Turing machine if the language recognized has the property p. This is even worse in networks of computers, where performance characteristics pose computational burdens on the analysis [142, 143]. To aleviate this, either reductions in formal language strength to no longer be Turing complete—which cannot be checked when looking at a binary for a general computer—or doing more analysis on the implementation before it gets compiled into a binary file, is needed.

There are languages which even pride themselves on being hard to analyse, some even hard to program in, like Ben Olmstead's 1998 Malbolge [279, 327] (with Ørjan Johansen's dialect Malbolge Unshackled [203] proven Turing complete through implementing a Brainfuck interpreter [241]), which are interesting as counterpoints to good engineering and security practices. This is called "Turing tarpit" [286], where everything can be programmed but it is hard to do so for anything non-trivial.

Sooner or later, to achieve supply chain security we need to check what goes into binaries and treat them as artifacts of that process. In general, trust in computers and

the programs they execute is hard without knowing details about the programs and who wrote them [392]. For making some trust relationships possible to be established, we need cryptography, which has very specific differences from other, less security-critical software.

We now want to give a quick overview of what we need from cryptography, and what it needs to provide this itself.

2.2. Cryptography

Secret key cryptography—also known as symmetric key cryptography, after all communication parties sharing the same secret key—has been known for a long time. Newer systems, like AES, were chosen in public to be secure against the attacks of the best public researchers in the hopes of avoiding crucial weaknesses in their design. Following Kerckhoffs's principle as formulated by Bolovin "design your system assuming that your opponents know it in detail" [45], the secrecy of a message in a cryptographic system should not depend on details of the implementation, but only on secret keys.

A newer and different kind of systems is asymmetric cryptography, publicly described by Diffe and Hellman in 1976 [118]. In this class of systems, it is possible to establish security goals without a shared secret. One of the possibilities is a shared secret, established through a key exchange or using a key encapsulation method (KEM), another popular one is cryptographic signatures. To show how these can be used, we have to look at their properties, their guarantees, assumptions and how to protect them.

Secretness of the content of secret keys and helping developers of cryptographic software implementations is what our research has been about. Therefore, we will look into a criterion that we will see in more parts of this thesis.

2.2.1. Constant Time Criterion

The intuitively misnamed Constant Time criterion for cryptographic algorithm implementations—software and hardware, but in this thesis we will focus on software implementations and their analysis—is a well known [199] baseline for the security against side-channel attacks. Before going into its details, we explain what side-channels and attacks against them are in the context of cryptographic implementations. A side-channel is any kind of measurement of the physical side-effects of a computation, for example time taken for the computation [211], energy emissions in the form of heat [207] and electric fields [220], energy draw—absolute value(Simple Power Analysis) or differences (Differential Power Analysis) [212]—of the computational device, access patterns to memory cells visible through caches [47, 283], even acoustic emanations [153]. In this context, a side-channel is called "silent", if no information from its measurements can be used to break the security assumptions of a cryptographic implementation. The most common of those assumptions is that the secret key of an

algorithm is not made public through the computations done with it. A side-channel attack is an attack on a cryptographic algorithm implementation using a non-silent side-channel for information that makes breaking the implementation feasible, or at least easier. Those attacks get more practical with more research effort put into them, consequently the common byline is "attacks only get stronger" [188]. The Constant Time criterion [211], that could better be called secret-key invariant execution, is based on minimizing the execution timing side-channel through formally provable means:

1. Memory access must not depend on the value of bits of the secret key.
2. Branching in code must not depend on the value of bits of the secret key.
3. Operations in a CPU must not vary in execution time (cycle latency) depending on the value of bits of the secret key.

The last of these three conditions is sometimes omitted, since it is dependent on the exact CPU model under analysis, might only give small side-channel information or could be harder to check for in a general way.

Memory accesses can be timed easier [72] due to caching of memory pages giving vastly different access times dependent of the location of memory pages in either the main RAM or a faster CPU cache. That is why no memory access may depend on a secret value, which in practice means no memory access should be indexed by said secret value.

Branching conditions directly dependent on secret values are a problem due to the branch prediction unit of processors giving different execution times depending on which branch is taken [211], but even without this, two code paths can have different execution times and that is why the choice of branch should not depend on secret values.

To verify if a software implementation exhibits the Constant Time criterion, we have to look into program analysis, and its more specialized form of cryptographic code analysis.

2.2.2. Cryptographic Code Analysis

Analysing generic binaries on modern computing architectures can lead to unsound and incomplete results due to the Turing completeness of the platforms [157]. This is sometimes called "Turing tarpit", in reference to Turing-completeness of programming languages, which is often seen as a benefit, so programmers don't need to prove termination of their non-trivial programs, like to assist an interactive theorem prover [314]. This also makes automatic classification about properties of programs infeasible in general, due to the Halting Problem, or "Entscheidungsproblem", as proven by Church in 1936 [94], which gives rise to the aforementioned Rice's Theorem.

Analysis of a program consists of either restricting to a formal language that *can* be analysed [156], or taking a solid guess as to what will happen if we cannot do that, which can be supplemented by partial information by a human engineering a proof

or automatically solving which branches can be taken depending on inputs to the program via symbolic execution [208]. Normal execution of a program and analysing its behaviour is called dynamic analysis or program testing, and conversely analysing a program without having to execute it is called static analysis [271], but the distinction need not be as clear if we reason about partial programs. Through abstract interpretation [105, 106] we can reason about program semantics. Through symbolic execution, we can reason about values a conditional may exhibit and restrict the tree of possible executions a program may exhibit under them, thereby trying to restrict the possible executions to something that can be analysed exhaustively [271]. This can be combined with partial execution of parts of the program with concrete values, called concolic execution [155].

In this thesis, we focus on the soundness and completeness of its analysis, as known from mathematical logic, not which kind of analysis a program analysing program will do. In this context, an analysis is called *sound*, if no program that may exhibit non-CT behaviour is labelled CT by the analysis [38]. Completeness is not the opposite of that, but the property that every finding during analysis of possible non-CT behaviour is an actual CT-violation, not a false positive [38].

Analysing binaries sooner or later becomes infeasible due to the earlier mentioned Rice's theorem, which the methods mentioned earlier try to work around by using some form of automated theorem proving, which may run into undecidable problems.

Cryptographic code can be easier to analyse than general program code due to specifics of cryptographic algorithm implementations, if done in a high-level programming language [38]: Most often, cryptographic algorithms are sure to terminate, so termination analysis can easily classify this code as total functions, or at least proofs for termination can be made [39]. Highly optimized implementations tend to favour (straight)line-code [132], which looks similar to Static Single Assignment form used in compiler construction for its simplicity and can be analyzed just as well [132]. Loops mostly have statically known bounds, or at least the termination conditions can be proven to be finite. This makes cryptographic algorithm implementations easier to prove properties about than general programs, but not necessarily easy [38].

Since we can theoretically analyze cryptographic code, we need to look into the practice of writing it in the first place, to see what developers are doing.

2.2.3. Cryptographic Engineering

While analysis can sometimes infer the absence of some forms of side-channel attacks, the programmer has to implement the cryptographic algorithms under specific constraints to shape programs amenable to resource and efficiency/speed constraints as well as analysis and security criteria. This practice is part of cryptographic engineering. Constant time coding practice is designed to protect against the most basic of side-channel attacks. Neither is there a formal way to achieve Constant Time code in every programming language, nor is there a way to instruct every compiler to not use optimizations that might destroy this security property in favor of potential execution

speed gains [206].

A given cryptographic algorithm can sometimes be replaced by another one that is free of branching behaviour [49]. If that is not possible, a conversion of the branching condition and result expression can sometimes be converted into arithmetic domain [329], where one result is encoded as conditional bit multiplied by value of the first result expression and the inverse of the conditional bit multiplied by the value of the second result expression. This transformation is no longer CT, when a compiler infers a multiplication by zero not needing to evaluate the result expression for its value, so cryptographic engineering most often is a struggle against compiler engineers having new ideas on how to optimize the cryptographers code against their will [206].

Alternative tool chains for cryptographic code development exist [20, 18, 38, 132, 300, 412, 85]. They are, however, not meant for general purpose programming tasks—targeting either typed dialects of assembly language, domain specific languages or full on dependently typed programming languages with proof assistants—and therefore lack broader adoption. Some provide high assurances against side-channels, for functional correctness, but integrating their code into programs generated through use of a more general purpose programming language is another effort for cryptographic engineering, often achieved through glue code in C and Foreign Function Interfaces [300, 132].

In the end, it is up to the cryptographic engineers to make their code safe and verify that the resulting binaries being executed on deployed hardware for their users are actually safe. This is where usable security can show a gap between theory and practice.

2.2.4. Usable Security

Tying this together is the viewpoint that to achieve security for users of computers and the programs that run on them, we need to look into how to make it easy for cryptographic software developers to test their software against non-CT behaviour more rigorously than we need to check, for example, a general text editor. Usable security often looks at the usability of security software solutions to their users, but in this thesis, every user *is a developer themselves*, and not even every developer is as critically touched by CT problems as cryptography experts, who dedicate their efforts to making sure their implementations can actually keep the promises their algorithms pay forward.

Cryptographic software implementations are *critical* to the software supply chain [233] and need to be tested not only for correctness, but also for their security of not leaking the secret keys while operating [72], so we can build a secure software supply chain on the primitives implemented in, for example, tools that use cryptographic signatures to sign files containing source code, linking them unequivocally to the person we *trust* to keep their secret key to themselves and their code in good, working order.

This is why in this thesis, we preface the two chapters on usability of tools verify-

ing CT to the chapter on supply chain security impacts of reproducible builds, which depend on cryptographic primitives being implemented.

3. What Cryptographic Library Developers Think About Timing Attacks

Disclaimer

The contents of this chapter were previously peer-reviewed and published as part of the conference paper titled "'They're not that hard to mitigate': What Cryptographic Library Developers Think About Timing Attacks", presented at the 2022 IEEE Symposium on Security and Privacy. This research was conducted as a team with my co-authors Jan Jancar, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar; this chapter therefore uses the academic "we". The initial idea was developed from myself with Jan Jancar, Peter Schwabe, Gilles Barthe, and Yasemin Acar. We conducted the study with our other co-authors and jointly analyzed the data and co-wrote the paper for publication. This paper describes the state of cryptographic constant time verification tools up to the year 2021. Research in Chapter 4 builds upon knowledge gained in this ecosystem.

3.1. Motivation

While some of our initial team are cryptographic engineers at heart, we all knew that the tools to verify best practices in cryptographic engineering in any implementation not specifically tailored to verifiability is hamstrung and largely best effort itself. We ventured out to guess that this feeling is universal, but we had no numbers to back this up. From one discussion round, we decided to investigate all the folk tales of cryptographic engineering around constant-time programming and the resulting programs, see what other practitioners experienced and use this as a basis for more development of one at that time still being in development prototype of a new verification tool.

The goal of this study was to get a qualitative, scientifically citable overview of not just sentiments of cryptographic software developers, but also an as-complete-as-possible view of the state of the art of constant-time verification tools, where publicly available or not. We decided to seek help of a usable security research expert and some international colleagues.

Our team decided to give an overview of the whole problem space in as approachable a way as possible, to ask ourselves "why are constant-time violations not eradicated, 25 years after the first scientific paper and 11 years after public release of the

first tool for checking the code?". We found different opinions among the experts, which we will get to after first talking about the problem area as it is still, today.

3.2. Introduction

Cryptographic protocols, such as TLS (Transport Layer Security), are the backbone of computer security, and are used at scale for securing the Internet, the Cloud, and many other applications. Quite strikingly, the deployment of these protocols rests on a small number of open-source libraries, developed by a rather small group of outstanding developers. These developers have a unique set of skills that are needed for writing efficient, correct, and secure implementations of (often sophisticated) cryptographic routines; in particular, they combine an excellent knowledge of cryptography and of computer architectures and a deep understanding of low-level programming. Unfortunately, in spite of developers' skills and experience, new and sometimes far-reaching vulnerabilities and attacks are regularly discovered in major open-source cryptographic libraries. One class of vulnerabilities are *timing attacks*, which let an attacker retrieve secret material, such as cryptographic keys, *"by carefully measuring the amount of time required to perform private key operations"*. Although timing attacks were first described by Kocher in 1996 [211], they continue to plague implementations of cryptographic libraries. There are multiple aspects that make timing attacks *special* in comparison to other side-channel attacks such as power-analysis or EM attacks. First, they can be carried out *remotely*, both in the sense of running code in parallel to the victim code without the need of local access to the target computer, but also in the sense of only interacting with a server over the network and measuring network timings [72] or over the Cloud [410]. As a consequence, unlike many other side-channel attacks, timing attacks cannot be prevented by restricting physical access to the target machine. Second, timing attacks do not leave traces on the victim's machine beyond possibly suspicious access logs, and we do not know at all to what extent they are being carried out in the real world, for example by government agencies: victims are not able to reliably detect that they are under attack and the attackers will never tell.

At the same time, and most importantly for this paper, we know how to systematically protect against timing attacks. The basic idea of such systematic countermeasures was already described by Kocher in 1996 [211]: we need to ensure that all code takes time independent of secret data. It is important here to not just consider the total time taken by some cryptographic computation, but make sure that this property holds for each instruction. This paradigm is known as *constant-time*[1] cryptography and is usually achieved by ensuring that

- there is no data flow from secrets into branch conditions;
- addresses used for memory access do not depend on secret data; and

[1]The term constant-time, often referred as CT, is a bit of a misnomer, as it does not refer to CPU execution time but rather to a structural property of programs. However, it is well-established in the cryptography community.

- no secret-dependent data is used as input to variable-time arithmetic instructions (such as, e.g., DIV on most Intel processors or SMULL/UMULL on ARM Cortex-M3).

Constant-timeness is no panacea, and the above rules may not be sufficient on some micro-architectures or in the presence of speculative execution, but essentially all timing-attack vulnerabilities found so far in cryptographic libraries could have been avoided by following these rules. For this reason, the notion of constant-time has grown in importance in standardization processes and recent cryptographic competitions. For instance, in the context of the ongoing Post-Quantum Cryptography Standardization project, the National Institute of Standards and Technology have stated in their Call for Papers [275]:

> *"Schemes that can be made resistant to side-channel attack at minimal cost are more desirable than those whose performance is severely hampered by any attempt to resist side-channel attacks. We further note that optimized implementations that address side-channel attacks (e.g., constant-time implementations) are more meaningful than those which do not."*

Protection against side-channel attacks, including timing attacks, is also routinely included as a requirement for Common Criteria certification as well as a part of the newly approved FIPS 140-3 certification scheme [31].

Programming highly optimized code that is also constant-time can be very challenging. However, we know how to verify that programs are constant-time. This was first demonstrated by Adam Langley's ctgrind [229], developed in 2010, the first tool to support analysis of constant-timeness. A decade later, there are now more than 30 tools for checking that code satisfies constant-timeness or is resistant against side-channels [198, 38]. These tools differ in their goals, achievements, and status. Yet, they collectively demonstrate that automated analysis of constant-time programs is feasible; for instance, a 2019 review [38] lists automatic verification of constant-time real-world code as one achievement of computer-aided cryptography, an emerging field that develops and applies formal, machine-checkable approaches to the design, analysis, and implementation of cryptography.

Based on this state of affairs, one would expect that timing leaks in cryptographic software have been systematically eliminated, and timing attacks are a thing of the past. Unfortunately, this is far from true, so in this paper we set out to answer the question:

Why is today's cryptographic software not free of timing-attack vulnerabilities?

More specifically, to understand how real-world cryptographic library developers think about timing attacks and the constant-time property, as well as constant-time verification tools, we conducted a mixed-methods online survey with 44 developers of 27 popular cryptographic libraries / primitives[2]. Through this survey, we track down the origin of the persistence of timing attacks by addressing multiple sub-questions:

[2]We refer to both as "libraries" for readability.

RQ1: (a) Are timing attacks part of threat models of libraries/primitives? (b) Do libraries and primitives claim resistance against timing attacks?
RQ2: (a) How do libraries/primitives protect against timing attacks? (b) Are libraries and primitives being verified/tested for constant-timeness? (c) How often/when is this done?
RQ3: (a) What is the state of awareness of tools that can verify constant-timeness? (b) What are the experiences with the tools?
RQ4: Are participants inclined to hypothetically use formal-analysis-based, dynamic instrumentation, or runtime statistical test tools, based on tool use requirements and guarantees?
RQ5: What would developers want from constant-time verification tools?

We find that, while all 44 participants are aware of timing attacks, not all cryptographic libraries have verified/tested resistance against timing attacks. Reasons for this include varying threat models, a lack of awareness of tooling that supports testing/verification, lack of availability, as well as a perceived significant effort in using those tools (see Figure 3.1). We expose these reasons, and provide recommendations to tool developers, cryptographic libraries developers, compiler writers, and standardization bodies to overcome the main obstacles towards a more systematic protection against timing attacks. We also briefly discuss how these recommendations extend to closely related lines of research, including tools for protecting against Spectre-style attacks [210].

3.3. Background & Related Work

3.3.1. Attacks

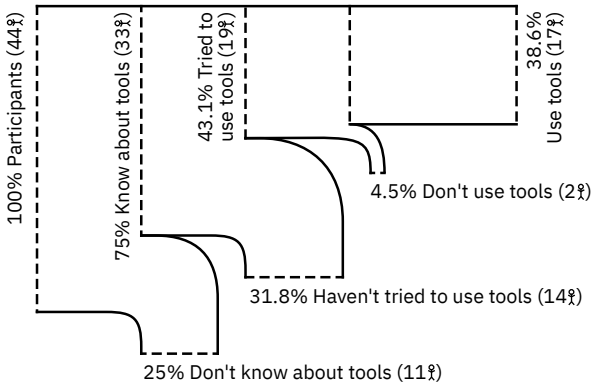

Figure 3.1.: Leaky pipeline of developers' knowledge and use of tools for testing or verifying constant-timeness.

In 1996, Kocher introduced the concept of *timing attacks* as a means to attack cryptographic implementations *"by carefully measuring the amount of time required to perform private key operations"* [211]. He described successful timing attacks against implementations of various building blocks commonly used in asymmetric cryptography like modular exponentiation and Montgomery reduction against RSA and DSS. Since this seminal paper, timing attacks have been further refined and continued to plague implementations of both asymmetric and symmetric cryptography. Successful timing attacks are way too numerous to list all, so we focus on a few relevant examples.

In 2002, Tsunoo et al. [361, 360] were the first to present attacks exploiting cache timing to break symmetric cryptography (MISTY1 and DES); they also mentioned a cache-timing attack against AES. Details of cache-timing attacks against AES were first presented in independent concurrent work by Bernstein [47] and by Osvik, Shamir, and Tromer [283]. In 2003, D. Brumley and Boneh showed that timing attacks can be mounted remotely by measuring timing variations in response times of SSL servers over the network [72]. Canvel et al. showed in 2003 how to recover passwords in SSL/TLS channels using padding oracle attacks [80]. In 2011, B. Brumley and Tuveri showed that such remote attacks are still possible [71]; i.e., that the underlying weaknesses in the OpenSSL library had not been suitably fixed. SSL libraries continued to be the target of timing attacks; examples include the "Lucky 13" attack by AlFardan and Paterson, which exploits timing variation in the processing of padding in the CBC mode of operation in multiple common SSL/TLS libraries [16] similar in principle to the paper by Canvel et al. [80]. In 2015, Albrecht and Paterson presented a variant of the attack targeting Amazon's s2n implementation of TLS [14]. In 2016, Yarom, Genkin, and Heninger presented the "CacheBleed" attack, which showed that the "scatter-gather" implementation technique recommended by Intel [67] and implemented in OpenSSL as cache-timing attack countermeasure, is insufficient to thwart attacks [402]. In the same year, Kaufmann et al. showed that even carefully implemented C code may be translated to binaries that are vulnerable to timing attacks [206].

We conclude this paragraph with a few attacks related to certification and standardization. Certification schemes such as Common Criteria often require certified products to have countermeasures to a range of side-channel attacks, including timing attacks. However, certified hardware did not avoid being a target of timing attacks, as shown by the recent Minerva group of vulnerabilities in ECDSA implementations, including a Common Criteria certified smart card [200]. In recent years, various timing attacks were proposed against implementations of post-quantum cryptography (PQC) including attacks against the BLISS signature scheme used in the strongSwan IPsec implementation [70, 289, 41] and attacks against candidates in NIST's PQC standardization effort [284, 378, 167].

Despite all these academic timing attacks, their practical exploitability is often questioned by practitioners. Security Audit companies try to catch timing vulnerabilities in software [243]. However, they make the following statements: *"Even though there is basic awareness of timing side-channel attacks in the community, they often go unnoticed or are flagged during code audits without a true understanding of their exploitability in practice."*

3.3.2. Tools included in the survey

We provide a brief overview of the tools considered in our survey. We classify tools according to the broad approach they use: runtime statistical tests, dynamic-instrumentation based, or formal-analysis based. Our approach as well as our choice of included tools is based on an earlier paper [38], but amended with tools some authors know to be in

current use.

Broadly speaking, statistical test tools [307] compute the execution time of a large number of runs of the target program and verify whether secret data influences the execution time. These tools are generally easy to install and run, even at scale, and operate on executable code, ruling out the possibility of compiler-induced violations of the constant-time policy. However, they only provide weak, informal guarantees.

In contrast, dynamic instrumentation based tools [111, 53, 381, 66, 229, 181, 386, 387, 363, 346, 396, 264, 109] instrument programs to track how information flows during (concrete or symbolic) execution of programs. They are generally reasonably easy to install and to use, even at scale, and can be implemented at source, intermediate, or assembly levels, and provide formal guarantees. However, as with all tools based on dynamic techniques, these guarantees are generally limited; for instance, dynamic analysis of loops may be unsound, i.e., miss constant-time violations.

Finally, formal-analysis-based tools [21, 29, 125, 19, 384, 85, 316, 213, 30, 91, 42] provide strong guarantees that programs do not violate constant-timeness; in addition, some tools are precise, in that they only reject programs that violate constant-timeness. Their other criterion is soundness, which ensures the absence of constant-time violations. However, these tools are often implemented at source or intermediate levels, frequently require user interaction, and are sometimes hard to install or use at scale.

Table 3.1 presents some key tools and summarizes their main characteristics. Since our focus is not an in-depth technical comparison of the features of the tools, we deliberately keep descriptions simple, and only consider their target and whether they provide some formal guarantees (No, Partial, Yes, Other). For the cognizant, "Partial guarantees" cover tools that perform dynamic analysis, whereas "Guarantees" cover tools that are sound and detect all constant-time violations; in particular, our classification does not reflect if tools are precise. Even for such coarse criteria, classification is sometimes challenging so we err on the generous side. Finally, we tag tools as "Other" if they establish another property than constant-time; comparing these properties with constant-time is often tricky, so we choose not to qualify the difference.

While the CoCo-Channel authors wrote [66]: *"We also evaluate CoCo-Channel against two recent tools for detecting side-channel vulnerabilities in Java applications, Blazer and Themis. Neither are publicly available*[...]*"*, their tool was not found by us either.

We do not claim our list to be comprehensive, especially in this currently active field of research. In particular, we did not ask about Constantine [56], Pitchfork-angr [364], Cachefix [89], and ENCoVer[36], just to name a few.

3.3.3. Libraries included in the survey

Cryptographic libraries have diverse threat models, but with their usual use in protocols like TLS and connected applications often running on shared hardware, resistance against timing attacks is an important property. In our survey, we invited developers of all widely used TLS libraries and other smaller but popular libraries and relevant primitives. We focused on libraries implemented in C/C++ as it is the target language

Tool	Target	Techn.	Guarantees
ABPV13 [21]	C	Formal	●
Binsec/Rel [111]	Binary	Symbolic	◐
Blazer [29]	Java	Formal	●
BPT17 [53]	C	Symbolic	◐
CacheAudit [125]	Binary	Formal	■
CacheD [381]	Trace	Symbolic	○
COCO-CHANNEL [66]	Java	Symbolic	●
ctgrind [229]	Binary	Dynamic	◐
ct-fuzz [181]	LLVM	Dynamic	○
ct-verif [19]	LLVM	Formal	●
CT-WASM [384]	WASM	Formal†	●
DATA [387, 386]	Binary	Dynamic	◐
dudect [307]	Binary	Statistics	○
FaCT [85]	DSL	Formal†	●
FlowTracker [316]	LLVM	Formal	●
haybale-pitchfork [363]	LLVM	Symbolic	◐
KMO12 [213]	Binary	Formal	■
MemSan [346]	LLVM	Dynamic	◐
MicroWalk [396]	Binary	Dynamic	◐
SC-Eliminator [400]	LLVM	Formal†	●
SideTrail [30]	LLVM	Formal	■
Themis [91]	Java	Formal	●
timecop [264]	Binary	Dynamic	◐
tis-ct [109]	C	Symbolic	◐
VirtualCert [42]	x86	Formal	●

Targets: LLVM - intermediate representation, DSL - domain-specific language, WASM - Web Assembly
Technique: † - also performs code transformation/synthesis
Guarantees: ● - sound, ◐ - sound with restrictions, ○ - no guarantee, ■ - other property

Table 3.1.: Classification of tools included in the survey.

of most tools and the most used language for cryptographic libraries. However, we included some libraries implemented in Java, Rust and Python if some tools can analyse them or they contain parts implemented in C.

Our choice of libraries is underpinned not only by our knowledge of them but also by quantitative data of user and developer numbers. We included some newer primitives not (yet) fulfilling this criterion to complement the answers given by the first group. Nemec et al. [265] gave numbers for OpenSSL: *"The prevalence of OpenSSL reaches almost 85% within the current Alexa top 1M domains and more than 96% for client-side SSH keys as used by GitHub users."* We only included libraries with an open development model to allow us to get data for our recruiting choice.

Table 3.2 contains a list of libraries included in the survey and whether at least one of their developers participated in our survey. The table also lists the actions that the libraries perform in their Continuous Integration (CI) pipelines. We draw this information from documentation and the public CI pipelines of the libraries. One author extracted this, a second author double-checked, with disagreements discussed

Library	Particip.	Continuous integration			
		Build	Test	Fuzz‡	CT test
OpenSSL	✔	✔	✔	✔	✖
LibreSSL	✔	✔	✔	✔	✖
BoringSSL	✔	✔	✔	✔	✔
BearSSL	✔	✔	✔	✖	✖
Botan	✔	✔	✔	✔	✔
Crypto++		✔	✔	✖	✖
wolfSSL	✔	✔	✔	✔	✖
mbedTLS	✔	✔	✔	✔	✔
Amazon s2n	✔	✔	✔	✔	✔
MatrixSSL		No public CI			
GnuTLS	✔	✔	✔	✔	✖
NSS	✔	✔	✔	✔	✖
libtomcrypt	✔	✔	✔	✖	✖
libgcrypt	✔	No public CI			
Nettle	✔	✔	✔	✔	✖
Microsoft SymCrypt	✔	✔	✔	✔	✖
Intel IPP crypto		No public CI			
cryptlib	✔	No public CI			
libsecp256k1	✔	✔	✔	✔	✔
NaCl	✔	No public CI			
libsodium	✔	✔	✔	✔	✖
monocypher	✔	✔	✔	✔	✖
BouncyCastle*	✔	✔	✔	✖	✖
OpenJDK		✔	✔	✖	✖
dalek-cryptography†		✔	✔	✖	✖
ring†		✔	✔	✔	✖
RustCrypto†	✔	✔	✔	✖	✖
rustls†	✔	✔	✔	✔	✖
python-ecdsa	✔	✔	✔	✖	✖
micro-ecc		No public CI			
tiny-AES-c	✔	✔	✔	✖	✖
PQCrypto-SIDH	✔	✔	✔	✖	✔
csidh	✔	No public CI			
constant-csidh-c-implementation	✔	No public CI			
ARMv8-CSIDH		No public CI			
SPHINCS+		✔	✔	✖	✖
Total = 36	27 (75%)	27 (75%)	27 (75%)	16 (44%)	6 (17%)

* Java
† Rust
‡ Includes being fuzzed by OSS-Fuzz or cryptofuzz.

Table 3.2.: Libraries and primitives included and the actions they perform in their public continuous integration pipelines.

and resolved.

3.3.4. Additional Related Work

Having already discussed timing attacks and tools for constant-time analysis, we briefly cover other related work.

Foundations of constant-time programming Constant-time programming is supported by rigorous foundations. These foundations typically establish that programs are protected against passive adversaries that observe program execution. However, Barthe et al. [42] show that constant-time programs are protected against system-level adversaries that control the cache (in prescribed ways) and the scheduler. Recently, these foundations have been extended to reflect micro-architectural attacks [78, 84, 166, 82]. In parallel, a large number of tools are being developed to prove that programs are speculative-constant-time, a strengthening of the constant-time property which offers protection against Spectre [210] attacks. We expect that many of the takeaways of our work are applicable to this novel direction of work.

High-assurance cryptography High-assurance cryptography is an emerging area that aims to build efficient implementations that achieve functional correctness, constant-timeness, and security. High-assurance cryptography has already achieved notable successes [38]. The most relevant success in the context of this work is the EverCrypt library [300, 412], which has been deployed in multiple real-world systems, notably Mozilla Firefox and Wireguard VPN. The EverCrypt library is formally verified for constant-timeness (and functional correctness). However, the library is conceived as drop-in replacements for existing implementations, and despite relying on an advanced infrastructure built around the F* programming language, this work does not explicitly target open-source cryptographic library developers as potential users of their infrastructure. Other projects that enforce constant-time by default, such as Jasmin [20, 18] or FaCT [85], target open source cryptographic library developers more explicitly, but rely on domain-specific languages, which may hinder their broad adoption. In contrast, we focus on tools that do not impose a specific programming framework for developers.

Human factor research Researchers have tried to answer the question of why cryptographic advances do not necessarily reach users. In a 2017 study, Acar et al. find that bad cryptographic library usability contributes to misuse, and therefore insecure code [5]. Krueger et al. developed and built upon a wizard to create secure code snippets for cryptographic use cases [217, 219]. Unlike these prior studies that investigate users of cryptographic libraries, we study the developers of cryptographic libraries, their threat models and decisions as they relate to *timing attacks*.

Haney et al. investigate the mindsets of those who develop cryptographic software, finding that company culture and security mindsets influence each other positively, but also that some cryptographic product developers do not adhere to software engineering best practices (e.g., they write their own cryptographic code) [171]. We

expand on this research by surveying open-source cryptographic library developers with respect to their decisions and threat models as they relate to side-channel attacks.

In the setting of constant-time programming, Cauligi et al. [85] carry a study with over 100 UCSD students to understand the benefits of FaCT, a domain-specific framework that enforces constant-time at compile-time, with respect to constant-time programming in C. They find that tool support for constant-time programming is helpful. We expand on their study by surveying open-source cryptographic library developers and considering a large set of tools.

Very recently, there have been calls to make formal verification accessible to developers: Reid et al. suggest "meeting developers where they are" and integrating formal verification functionality in tools and workflows that developers are already using [303]. To our knowledge, ours is the first survey that empirically assesses cryptographic library developers' experiences with formal verification tools.

3.4. Methodology

In this section, we provide details on the procedure and structure of the survey we conducted with 44 developers of popular cryptographic libraries and primitives. We describe the coding process for qualitative data, as well as the approach for statistical analyses for quantitative results. We explain our data collection and ethical considerations, and discuss the limitations of this work.

3.4.1. Study Procedure

We asked 201 representatives of popular cryptographic libraries or primitives to participate in our survey. The recruited developers reside in different time zones and each may have different time constraints. As we were mainly interested in qualitative insights, based on the small number of qualifying individuals and our past experiences with low opt-in rates when attempting to recruit high-level open source developers into interview studies, we opted for a survey with free-text answers.

Questionnaire Development We used our research questions as the basis for our questionnaire development, but we also let our experience with the development of cryptographic libraries, constant-time verification tools (both as authors as well as users), and conducting developer surveys influence the design. Our group of authors consists of one human factors researcher and experts from cryptographic engineering, side-channel attacks, and constant-time tool developers. The human factors researcher introduced and facilitated the use of human factors research methodology to answer experts' research questions posited in this paper. In particular, the human factor researcher explained methods when appropriate, facilitated many discussions and helped the team to develop the survey, pilot it, gather feedback, and evaluate the results. While iterating over the questionnaire, we also collected feedback and input from members of the cryptographic library development community.

Pre-Testing Following the principle of cognitive interviews [399], we walked through the survey with three participants who belonged to our targeted population, and updated, expanded and clarified the survey accordingly.

Recruitment and Inclusion Criteria We created a list of the most active contributors to libraries that implement cryptographic code, including those that implement cryptographic primitives. If a library had any formal committee for making technical decisions, we invited its members. The list of most active developers was extracted from source control by taking the developers with the largest amount of commits down to a cut-off point that was adjusted per library. Table 3.2 gives an overview of projects for which we invited participants. All authors then identified those contributors that belonged to their own personal or professional networks and invited those in a personalized email. All others were invited by a co-author who is active in the formal verification and cryptography community, for whom we assumed that they would be widely known and have the best chance of eliciting responses. All contributors were sent an invitation with a personalized link. We did not offer participants compensation, but offered them links to all the tools we mentioned in our survey, as well as the option to be informed about our results.

3.4.2. Survey Structure

The survey consisted of six sections (see Figure 3.2) detailed below. The full questionnaire can be found in Appendix A.1.

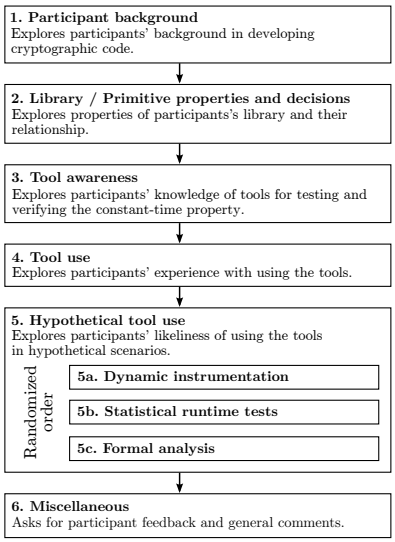

Figure 3.2.: Survey flow as shown to participants.

1. Participant background: We asked participants about their background in cryptography, their years of experience in developing cryptographic code, and their experience as a cryptographic library / primitive developer.

2. Library properties and decisions: We asked about participants' role in <library>'s development, how they are involved in design decisions for <library>. We asked about the intended use cases for <library>, <library>'s threat model with respect to side-channel attacks, whether they consider timing attacks a relevant threat for the intended use of <library> and its threat model, and asked for an explanation for their reasoning. We also asked whether and how <library> protects against timing attacks, and whether, how, and how often they test or verify resistance to timing attacks.

3. Tool awareness: We asked whether they were aware of tools that can test or verify resistance to timing attacks. We then listed 25 tools from Section 3.3.2 and asked them whether they were aware of them, and

how they learned about them.

4. Tool use: We asked participants about their past experience, interactions, comprehension, and satisfaction with using tools to test/verify resistance to timing attacks, including challenges with using them.

5. Hypothetical tool use: We showed participants a description of properties that their code would have to fulfill in order to be able to use a group of tools and given a description of the guarantees the tools would give them, asking them about usage intentions and reasoning. The tools were grouped into dynamic instrumentation based, statistical test based, and formal analysis tools.

6. Miscellaneous: Finally, we asked about any comments on (resistance to) timing attacks, our survey, and whether they wanted us to inform them about our results.

3.4.3. Coding and Analysis

Those who engaged with participant responses came from different backgrounds, with different views, contributing to the multi-faceted evaluation. Three researchers familiar with constant-time verification and open-source cryptographic library development conducted the qualitative coding process, facilitated by one researcher with experience with human factors research with developers. We followed the process for thematic analysis [65]. The three coders familiarized themselves with all free-text answers, and annotated them. Based on these annotations, themes were developed, as well as a codebook. The codebook was developed inductively based on questions, and iteratively changed based on responses we extracted from the free-text answers; all codes were operationalized based on discussions within the team. The three coders then coded all responses with the codebook, iterating over the codebook until they were able to make unanimous decisions. The codebook codifies answers to free-text questions, as well as identifying misconceptions, concerns, and wishes. In some cases where documentation was available, and participant answers were incomplete or ambiguous, or when participants linked to documentation, coders supplemented their code assignment based on the documentation. Our coding process was only one step in the quest for our goal: identifying themes and answering our research questions. All codes were discussed, and eventually agreed upon by three coders[3]. In line with contemporary human factors research, we therefore omit inter-coder agreement calculations [245]. For the comparison of the likelihood of using certain tools with certain requirements in exchange for guarantees (Q5.1, Q6.1, Q7.1), we used Friedman's test with Durbin post-hoc tests [99] with Benjamini-Hochberg multiple testing corrections [46].

3.4.4. Data Collection and Ethics

While our survey was sent to open-source contributors without solicitation, we only emailed them up to twice. During and after the survey, they could opt-out of par-

[3]Our codebook is available at https://crocs.fi.muni.cz/public/papers/usablect_sp22.

ticipation. We do not link participant names to results, nor participant demographics to libraries to keep responses as confidential as possible. We also do not link quotes to libraries or their developers, and report mostly aggregate data. Quotes are pseudonymized. Our study protocol and consent form were approved by our institution's data protection officer and ethics board and determined to be minimal risk. Participant names and email addresses were stored separately from study data, and only used for contacting participants.

3.4.5. Limitations

Like all surveys, our research suffers from multiple biases, including opt-in bias and self-reporting bias. However, we were pleasantly surprised that for 27 out of the 36 libraries we selected, we received at least one valid response. Participants may over-report desirable traits (like caring about side-channel attacks or protecting against them), and underplaying negative traits (like making decisions ad-hoc). However, their reporting generally tracked with official documents and our a priori knowledge about the libraries. The projects represent a selection, and are not representative of all cryptographic libraries. However, we took great care in inviting participants corresponding to a variety of prominent, widely used libraries as well as smaller but popular libraries and primitives, as assessed by multiple authors who work in this space.

3.4.6. Data cleaning & Presentation

We emailed 201[4] listed as most active contributors to 36 libraries/primitives, finding alternate emails for those emails that bounced. 2 emailed us to tell us that they did not think they could meaningfully contribute. In total, 71 started the survey. We removed all 25 incomplete responses. We removed two participants because they gave responses about a project of their own, instead of the library we asked them about. From here, we report results only for the 44 valid participants. For statistical testing and figures for hypothetical tool use, we report results for the 36 participants who gave answers for all three tool groups. We merged answers of participants talking about using a ctgrind-like approach but without the use of ctgrind itself (as it is no longer necessary as Valgrind can directly do this) into the ctgrind tool answers.

Participants spent an average of 32 minutes on the survey, and left rich free text comments. We generally received positive feedback and high interest in our work, and 35 asked to be sent our results, with 33 agreeing to be contacted for follow-up questions. Whenever we report results at the library level, we merge qualitative answers given by all participants corresponding to that library. Whenever the answers are additive, we add them together without reporting a conflict (e.g. when one developer tests a library in one way while another one tests it a different way, we report both). When the answers are claims of a level (e.g. resistance to timing attacks) we

[4]From now on, we use the symbol to denote the participants.

report the highest claimed. Otherwise, whenever we encounter conflicting opinions, we report on this conflict.

3.5. Results

In this section, we answer our research questions based on the results of our survey. Between full awareness and low levels of protection against timing attacks, we identify reasons for (not) choosing to develop and verify constant-time code, including a lack of (easy-to-use) tooling, trade-offs with competing tasks, understandable concerns and misconceptions about current tooling. We identify that participants would generally like the guarantees offered by tools, but fear negative experiences, code annotations and problems with scalability.

3.5.1. Survey Participants

Library developers

We successfully recruit experienced cryptographic library developers, including the most active contributors and decision-makers. We ended up with 44 recruited via direct invitation. Of our participants, 4 were the only developer in their project, 9 were project leads, 11 were core developers, 19 were maintainers, 11 were committers, 3 were contributors without commit rights. These classes are non-exclusive self-reports. 40 said they were involved in the library decision processes, while only 4 were not involved.

Participants had strong backgrounds in cryptographic development, reporting a median of 10 years of experience (sd = 7.75), and qualitatively reporting strong engagement with various projects, for example reporting involvement in security certifications: *"I've worked on open source and closed source cryptography libraries, dealt with various Common Criteria EAL4+ products"* (P1). As for the participants' concrete background in cryptography, 17 reported an academic background, 15 took some classes on cryptography, 32 had on the job experience, 6 teach cryptography, for 15 cryptography is (also) a hobby, 27 have industry experience in cryptography.

Libraries

We ended up with participants from 27 prominent libraries, such as OpenSSL, BoringSSL, mbedTLS or libgcrypt. Participants gave or linked to descriptions of a broad range of use cases for cryptographic libraries. As intended platforms, 23 gave servers, 22 desktop, 14 embedded device (with OS, 32 bit), 4 mobile, and 1 micro-controller (no OS, 8/16 bit). For targets, 7 stated TLS, 12 protocols, 2 services, 1 cloud, 2 operating systems, 1 crypto-currency, and 2 corporate internal purposes. Libraries had varying decision-making processes: 9 made decisions by discussion, 2 by voting, 3 by consensus, and for 11, decisions were made by the project leads who had a final say.

3.5.2. Answering Research Questions

Threat models (RQ1a)

Here, we answer the research question whether timing attacks are part of library developers' threat models (RQ1a). We found that *all participants were aware of timing attacks*. Generally, when a threat model is defined for a cryptographic library, it mostly includes timing attacks. However, strict and absolute adherence to constant-time code is most often not required. In practice, developers tend to distinguish vulnerabilities that are "easy" to exploit (e.g. remote timing attacks) from the others (e.g. locally exploitable attacks). When asked specifically about the library's threat models with respect to side-channel attacks, 20 libraries claimed remote attackers are in their threat model, 16 included local attackers, 1 included speculative execution attacks, 2 included physical attacks and 2 included fault attacks. Some libraries expressed that they consider some classes of attacks in their threat model if they are easy to mitigate, 2 would do so for local attacks and 1 for physical attacks. The general attitude towards side-channel attacks varied, 2 said that all side-channel attacks are outside their threat model and 10 said that their protections against side-channel attacks are best effort. For example, one participant said: *"Best-effort constant-time implementations. CPU additions and multiplications are assumed to be constant-time (platforms such as Cortex M3 are not officially supported)."* (P2) Another one implied a progressive widening of their threat model regarding timing attack in their statement: *"Protections against remote attacks, and slow movement to address local side channels, though the surface is wide."* (P3)

In a follow up question, 23 libraries agreed that timing attacks were considered a relevant threat for the intended use of the library and its threat model, while this was not true for 2 libraries. We did not get this information for 2 libraries.

Reasons for considering timing attacks as relevant for their threat model were given as the ease of doing so (2), the threats of key-recovery in asymmetric cryptography (3), user demands (1), fear of reputation loss (1), use in a hostile environment (6), that attacks get smarter (1), the (rising) relevance of timing attacks (9), personal expectations (5), a connected environment (2), or the large scope of the library/of timing attacks (3).

Reasons for not considering timing attacks as part of their threat model were stated as this not being a goal of the library (2) or that they only consider more "practical" attacks.

Resistance against timing attacks (RQ1b)

Here, we answer the research question whether libraries claim resistance against timing attacks (RQ1b). Many libraries do not have a systematic approach to address timing attacks; they only consider fixing "serious" vulnerabilities that could be exploited in practice. This might result in vulnerable code that can be exploited later with better techniques of recovering leaking information. We also encountered differing answers of different participants regarding suitedness of random delays as a mitigation. Out

of the 27 total libraries, 13 claimed resistance against timing attacks. An additional 10 claimed partial resistance, 3 claimed no resistance, and for 1, we obtained no information.

We also asked how the development team decided to protect against timing attacks. For 4 libraries, participants reported that one person made this decision, for 12 it was a team decision, for 2 it was a corporate decision (where high-level management makes a decision or the team decided locally based on a corporate mission statement), for 14 libraries, participants reported that a priority trade-off caused their decision (e.g., lack of time to fully enact the decision) and 5 inherited the decision from previous projects or developers.

For 6 it was obvious that they needed to protect against timing attacks. For example, one participant stated: *"There was no decision, not even a discussion. It was totally obvious for everybody right from the start that protection against timing attacks is necessary."* (P4) Another one said: *"It's just how you write cryptographic code, every other way is the wrong approach (unless in very specific circumstances or if no constant-time algorithm is known)."* (P5) Another stated

> *"It became clear that these attacks transition from being an "academic interest" to a "real world problem" on a schedule of their own development. If something is noticed we now tend to favor elimination on first sight without waiting for news of a practical attack."* (P6)

Contrarily, another said:

> *"Basically a trade-off of criticality of the algorithm vs practicality of countermeasures. Something very widely used (eg RSA, AES, ECDSA) is worth substantial efforts to protect. Something fairly niche (eg Camellia or SEED block ciphers) is more best-effort"* (P7)

This reasoning of waiting for attacks to justify expending the effort was also reported by another participant: *"For many cases there aren't enough real world attacks to justify spending time on preventing timing leaks."* (P8)

Timing attack protections (RQ2a)

Here, we answer the research question how developers choose to protect against timing attacks (RQ2a). Developers address timing attacks in various ways, for example by implementing constant-time hacks (e.g. constant selecting), implementing constant-time algorithms of cryptographic primitives, using special hardware instructions (CMOV, AES-NI), scatter-gathering for data access, blinding secret inputs, and slicing. Many are interested and willing to invest effort into this - to various degrees, as P9puts it: *"[T]hey're not that hard to mitigate, at least with the compilers I'm using right now"* (P9). Others are deterred by the lack of (easy-to-use) tooling.

We asked developers of the 23 libraries who considered timing attacks at least partially if and how their library protects against timing attacks.

For 2 libraries, participants reported that they use hardware features (instead of leak-prone algorithms) that protect from timing attacks such as AES-NI. For example, P7said: *"AES uses either hardware support, Mike Hamburg's vector permute trick, or else a byte-sliced version."* (P7)

For 21 libraries, participants said that they use constant-time code practices, which should in theory mean that code is constant-time by construction, but may be vulnerable to timing attacks after compilation. For example, P2explained that: *"Conditional branches and lookups are avoided on secrets. Assembly code and common tricks are used to prevent compiler optimizations."* (P2)

For 9 libraries, participants explained that they choose known-to-be constant-time algorithms, but may suffer from miscompilation issues and end up non-constant-time. As an example, P7said: *"If I know of a "natively" const time algorithm I use it (eg DJB's safegcd for gcd)."* (P7)

For 7 libraries, participants said they use "blinding", which means using randomization to "blind" inputs on which computation is performed, thereby destroying the usefulness of the leak. As P7said: *"If blinding is possible [...] it is used, even if the algorithm is otherwise believed constant-time. "* (P7)

For 2 libraries, participants said that they protect through bitslicing, i.e., the implementation uses parallelization on parts of the secrets, hiding leaks. As one participant described: *"For instance, the constant-time portable AES implementations use bitslicing."* (P11)

For 2 libraries, participants reported protecting by "assembly", i.e., they have a specialized low-level implementation for protecting against compilers doing non-constant-time transformations. One participant noted the prohibitive cost of this practice, explaining: *"We do not write all constant-time code in assembly because of the cost of carrying assembly code. It is possible that the compiler may break the constant-time property. We spot-check that using Valgrind."* (P12)

For 1 library, timing leaks are made harder to detect by adding random delays.

Most developers focus on asymmetric crypto. Some do not consider old primitives, such as DES, which is still used in payment systems as Triple-DES. For 5 libraries, participants stated that they only protect a choice of modules: those libraries have multiple implementations, of which only some might be constant-time, maybe even insecure by default. *"Legacy algorithms like RC4 and DES are out of scope. If you use the <libraries'> "BIGNUM" APIs to build custom constructions, it's probably leaky, since bignum width management is complex."* (P13) also mentions bignum libraries being specifically hard to secure. This claim is supported by academic literature as well: *"lazy resizing of Bignumbers in OpenSSL and LibreSSL yields a highly accurate and easily exploitable side channel"*[386].

For 1 library, protection against timing attacks was reported to be still in progress, e.g., they try to use constant-time coding practices throughout the library, but this is still in development due to large legacy code base. *"All decisions in a side project are limited by the available resources. There's a report about a new attack which proposes a new counter-measure: Does someone have the time to implement it? Yes - cool, let's do it. No - fine, let's put it on the ToDo list."* (P15) and *"Very early on in its development these guarantees*

were much weaker, and in a few cases, approaches were used that turned out to be known to be imperfect." (P16) were two answers from participants of libraries being in very different phases of solving this problem.

Testing of timing attack resistance (RQ2b, RQ2c)

In Software Engineering, testing code for the properties it should achieve is commonplace and generally considered best practice [50]. We therefore were interested in the practice of testing and verification for constant-time also. Here, we answer the research questions whether, how, and how often libraries test for/verify resistance against timing attacks (RQ2b, RQ2c).

For 21 libraries, at least some type of testing was done, of which 14 were fully, and 7 were partially tested. 6 were not tested including the 2 libraries which claimed timing attacks are not relevant. 24♀ personally tested their libraries.

Of those, 12 stated they have tested manually, and 11 stated they tested automatically. Those two answers are not exclusive, since 7 libraries which test code automatically have also been tested manually. For manual testing, 6 libraries analyzed (parts of) their source code, 4 libraries analyzed (parts of) their binary, 5 did manual statistical runtime testing for leakage, and 1 ran the code and looked at execution paths, debugging as it ran. *"Originally, me, a glass of bourbon, and gdb were a good trio. But that got old pretty quick. (The manual analysis part – not the whiskey.)"* (P17) conveys the experience quite graphically.

For those who did automated testing, 9 libraries used a Valgrind-based approach, 2 used ctgrind, 1 used Memsan, 1 used TriggerFlow, 1 used DATA, and 1 reported automated statistical testing without specifying further.

For the participants who did at least partial testing for resistance to timing attacks, we asked for testing frequency. For 1, the testing was only done once. For 11, participants reported manual or occasional testing. For 4, participants reported testing on release. For 6 libraries, participants reported that testing for resistance to timing attacks was part of their continuous integration. For 11 libraries, we did not obtain information on testing or testing frequency. These varying answers suggest that despite a common awareness of timing attacks, cryptographic developers never came to a consensus on the best way to address timing attacks in practice.

Tool awareness (RQ3a)

In order to effectively test, developers should be able to leverage existing tooling created for the purpose of testing and/or verifying that source code, or compiled code, runs in constant time. Here, we answer the research question whether participants are aware of the existence of such tooling (RQ3a). We asked participants whether they are aware of tools that can test or verify resistance against timing attacks, also showing them a list of tools from Table 3.1. We asked them whether they had heard about any of those tools with regards to verifying resistance against timing attacks. Table A.1 shows the results with 33♀ being aware of at least one tool and 11♀ being unaware of

any tool. ctgrind was most popular (27👤 heard of it; 17👤 had tried to use it), followed by ct-verif (17👤 heard of it; only 3👤 tried to use it) and MemSan (8👤 heard of it; 4 👤 tried to use it). DATA had been used by 2👤, all others by no more than 1👤. Individual tool awareness and use numbers can be found in Appendix A.2.

For those tools they had heard about, we asked them where they had heard about them. Overall, participants were recommended a tool by a colleague 33 times, heard of a tool from its authors 20 times, read the paper of the tool 27 times, read about the tool in a different paper or blog post 42 times and heard of it some other way 24 times. 2👤 were involved in a development of a tool. A general tool they are already using can also be used for constant-time-analysis, which P18learned through our survey: *"I already use MemSan primarily for memory fault detection. Was not aware of its use for side-channel detection but will try it in future since it is already integrated with my workflow to some extent"* (P18).

Again for the tools they were aware of, we asked which (if any) they had (tried to) use in the context of verifying or testing resistance to timing attacks. Table A.1 displays the results, with 19👤 having tried to use at least one tool and 25👤 having never tried any of the tools.

Tool experience and use cases (RQ3b)

Here, we answer the research question which experiences participants made with tools (RQ3b). As we were anecdotally aware that tools may be hard to obtain, unmaintained, and may be closer to research artifacts than ready-to-use tooling, we were interested in participants' experiences, finding that experience varied by tool, use cases and expectations. We therefore asked participants to describe the process of using the tools.

12👤 reported that they managed to get the respective tool to work at least once, but not necessarily repeatedly, while 3👤 reported that the tool they attempted to use failed to work even once, for various reasons, including excessive use of resources, such as effort, time, RAM, CPU cores, machines etc. One participant said of the DATA tool: *"it uses a ridiculous amount of resources"* (P17).

2👤 reported that they had integrated the tool into CI and were using it automatically. 12👤 reported that they used it manually, of which 6👤 said they use it during development, and 6👤 said they use it after development, on release. A participant said: *"Periodically, and manually, used when altering / writing code to check constant-time property."* (P12)

For those who had heard of specific tools, but had not attempted to use them, we were also interested in their reasoning. The reasons were varied, many including a lack of resources such as time (26👤) or RAM, CPU cores and machine (1👤). Participants also reported on bad availability (4👤), and maintenance (5👤), as well as insufficient language support (4👤), and other usability issues, such as problems with setting up the tool (3👤), or getting it to work properly post setup (1👤). The difficulty or impossibility of fulfilling the required code changes, such as markup for secret/public values, memory regions/aliasing, and additional header files was also a problem (re-

ported by 1), as was the inability to ignore reported issues, once flagged by the tool (8).

Some reported not needing the respective tool (22), using other tools (18), gave reasoning that to our understanding was based on misconceptions of the respective tool (2), or reported having been unaware of the tool's capabilities in the context of resistance to timing attacks (1).

One participant also said that the tool was also used to verify a security disclosure. *"Tried to use to reproduce results, verify disclosures. Tried to use it to discover new defects in existing code."* (P14) — since the tool is later stated as in use by another member of the same project, this confirms that the tool not only verified the initial defect, but works as planned.

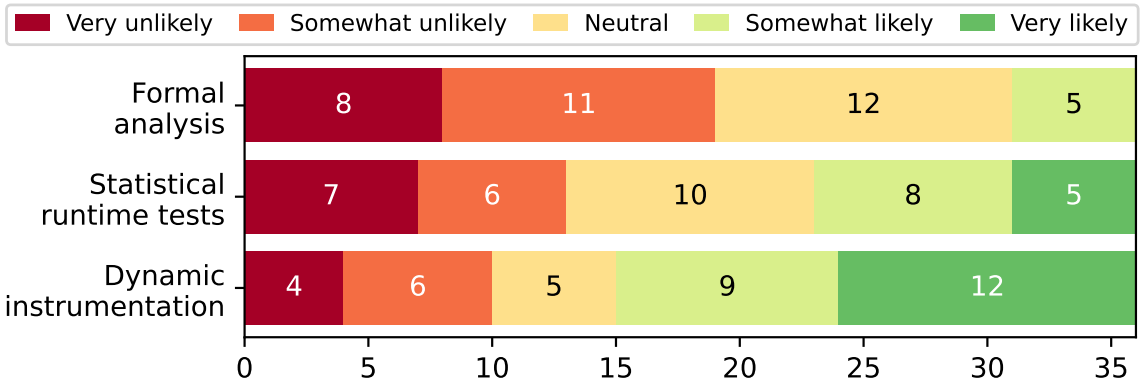

Figure 3.3.: Reported likeliness of tool use based on requirements and guarantees.

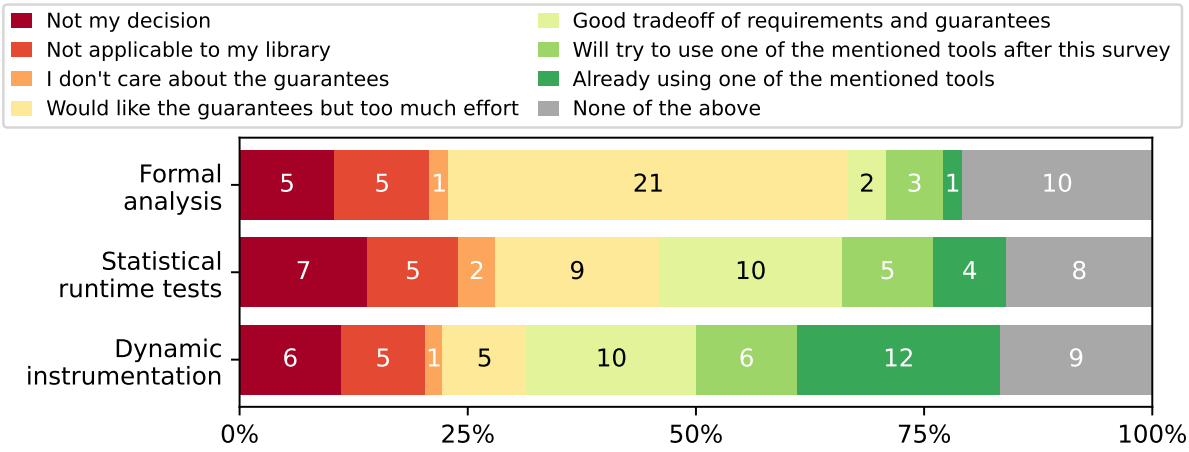

Figure 3.4.: Participant reasoning behind their likelihood of tool use.

Potential Tool use (RQ4)

In addition to understanding participants' current threat models and behaviors concerning constant-time code, we were also interested in what they thought about po-

tential future use of testing/verification tools, and whether they would potentially be willing to fulfill certain requirements in exchange for guarantees (RQ4, see Figure 3.3). Generally, they were most willing to use dynamic instrumentation tools, and also spoke about them the most positively, whereas they mostly mentioned drawbacks when asked about formal analysis tools.

We presented the participants with the requirements and guarantees offered by three categories of tools: dynamic instrumentation based tools, statistical runtime tests and formal analysis tools.[5] We then asked them to rate their likeliness of using the presented group of tools on a 5-point Likert scale from "1=very unlikely" to "5=very likely". Figure 3.3 shows a strong preference for dynamic tools, while formal analysis tools are least likely to be used. We perform statistical tests on these ratings to establish that these differences are statistically significant. We find a significant difference in participants' self-reported likeliness to use tools in the different categories (Friedman Test-Statistic=18.477, p<0.0001). Post-hoc testing showed that participants are significantly more likely to use dynamic instrumentation based tools like ctgrind (mean=3.53, sd=1.38) than statistical tools (mean=2.94, sd=1.31; p=0.023, Durbin-post-hoc (DPH), Benjamini-Hochberg-corrected (BH)) and formal analysis tools (mean=2.38, sd=0.98; p<0.0001, DPH, BH-corrected). The difference between statistical and formal analysis tools was not significant (p=0.18, DPH, BH-corrected). Specifically, while 21 reported being somewhat likely or very likely to use a dynamic testing tool for resistance to timing attacks in the future, 13 reported the same for statistical runtime test tools, and only 5 said they were somewhat likely to use formal analysis tools.

We also asked participants to clarify their reasoning by choosing explanations (see Figure 3.4). Results show that participants would like the guarantees formal analysis tools provide, but perceive them as requiring too much effort (21) compared to the other tools (9 statistical runtime tools, 5 dynamic instrumentation tools). More participants think that the trade-off of effort and guarantees is acceptable for dynamic (10) and statistical tools (10) than formal analysis tools (2). More details on participants' reasoning follow.

Dynamic Instrumentation Tools For dynamic instrumentation tools, some participants were happy with the limited guarantees, understanding the trade-off clearly.

> *"We currently use MemSan and Valgrind because they have very low maintenance since they pretty much come with the operating system, and we could get useful results from them with a few days' work. We are aware of their limitations (they miss non-constant-time parts, and of course they can only test code in the conditions where it is executed as part of the tests)."* (P19)

The approach taken by tools like ctgrind is understandable to developers, so much so that some came up with it independently: *"We independently came up with this approach and were using it [before we] knew ctgrind existed."* (P9)

[5]For the survey questions see the Appendix sections A.1.5,A.1.6 and A.1.7.

One participant specifically commented on the effort required to create and maintain annotations:

> *"A thing this survey might be underestimating is also the cost of code annotations: it's not just about having someone annotating the code properly (which already is quite a lot of effort) but there might be resistance for inclusion of such annotations in the code base as they add a maintenance burden for the project. Maintainers should fully understand the notation syntax and get proficient in it to spot instances where annotations need to be updated, moved, etc."* (P20)

Statistical Tools For tools based on statistical tests of the runtime, 7♀ expressed that the guarantees provided by the tools are limited. One participant explained: *"I am dubious that it would provide much value over existing mechanisms. Also, CI currently runs on shared hosts which are timing noisy. From this noise I would expect [...] false positives [...]"* (P21) Another participant also expressed concern over the guarantees and false positives / negatives: *"The requirements seem straightforward, but a statistical test seems likely to cause both false negatives and false positives."* (P13)

Formal Analysis Tools Participants had strong feelings about the lack of usability of formal analysis tools: *"I'm very interested in these sorts of tools, but so far it seems formal analysis tools (at least where we've tried to apply it to correctness) are not really usable by mere mortals yet. I would be happy to be proven wrong, however!"* (P13)

The fact that compiler optimizations can introduce timing leaks that will not be detected by tools working at the source code level was highlighted by a few participants: *"Static analysis on the source code in most programming languages is NOT sound: it misses compiler optimizations that introduce secret-dependent flows."* (P19) and another one explaining *"I'm much more worried about compilers failing to preserve constant-time code, ..."* (P13)

While 4♀ mentioned their expectation of a higher effort to create the necessary markup for formal analysis-based tools, expectations of the scalability of these tools seem to be in line with other categories of tools.

Additionally, we found that participants were intimidated by the theory-heavy approach by formal analysis-based tools, thinking of formal verification in general. *"I have no experience with formal verification toolchains"* (P23).

More academically focused formal analysis tools also suffer from a maintenance problem if the developers have moved on to other research: *"Who knows if the toolchain is still maintained in a year?"* (P5)

In conclusion, dynamic tools are mostly criticized for requiring code annotation, while statistics ones are viewed critically because of their poor guarantees. However, participants were most critical towards formal analysis tools. Some doubt that such tools would be maintained, or question that fact that they would provide large support for different platforms. While these drawbacks are real, they do concern all tools, but participants point them out mainly for formal analysis tools. Some participants

mention that such tools have steep learning curves, as they are not only unfriendly to use, but they also require specific knowledge. We notice that developers using ct-grind took their time to explain how it actually works (they we were never asked to), while participants remain vague about formal analysis tools. Only one participant actually uses such a tool, only few have tried, but not succeeded. However, many qualify such tools as uneasy to use, inefficient, lacking wide support, unable to verify external code, based only on the code, hard to be CI automated, adding very little confidence, and possibly unmaintained in the future.

Misconceptions

Despite surveying an expert population of cryptographic library developers, our study pointed out some misconceptions and differences of opinion about constant-timeness, timing attacks, and verification/testing tools. Those may deter from analysis tool use, and may contribute to more hidden timing vulnerabilities, ultimately making it harder to solve the timing attack problem in practice.

Some participants seemed to think constant-time is easily achieved. This logic implies if a project has a timing vulnerability, they have made a basic mistake. *"Writing constant-time code, contrary to writing* [...] *memory-safe code, is not hard, if you do it explicitly from start* (*caveat: when there's Gaussian rejection sampling in a lattice system, it _is_ hard*[...]*"* (P11) This ties in with code annotation not being usable when secretness of variables changes, specifically as mentioned with rejection sampling. This misconception is based around most common use of annotations. If the annotations allow for declassification of variables, this problem can be resolved granularly. Not all tools allow this, though, so the misconception that this is true for all tools may have taken root.

One participant suggested that they do not need to test code if they write constant-time code correctly. *"In that sense, the guarantees offered by these tools are not worth putting effort into running them, at least in the case of <library>, where all code was designed to be constant-time"* (P11). This sentiment comes with several problems: on the one hand, humans make mistakes, so testing code is a best practice in software engineering for precisely this reason. Additionally, compiling code that does fulfill the constant-time property may create problems, as the compiler may change the original control-flow while adding some optimizations.

While talking about compilation units and control-flow, a partial misconception can be found in verification scope: *"a lot of code will exist outside of the boundaries of the library. A project using <library> would be more likely to be successful."* (P20) While the library may not know which inputs are secret , looking at an API should make it clear which inputs can be secret, and the constant-time criterion could be tested for all of them without knowing the actual usage patterns.

Furthermore, the different answers about random delays and statistical analysis tools show that there is no universal consensus among the participants. A participant said: *"Anything involving secret data, and in particular private-key data, has the timing dithered and with throttling of repeated attempts to make attacks of this kind difficult."*

(P24) We are skeptical about this due to the results of Brumley and Tuveri [71]. If a side channel signal is measured as a timing difference between executions, adding a random noise distribution to these executions will reproduce a similar difference if enough samples of the executions are obtained. This can be done in parallel from different sources or over a long time, going around the throttling defense. A more practical quote is from P9:*"We once tried to test actual execution timings, but it wasn't reliable. We no longer do that. Now we use Valgrind."*

Lastly, even if cryptography is rather heavy in mathematics, some participants associate math/formal analysis as a barrier to using tools from that research area. *"[P]roving things like loop bounds is often arcane. Also, it's knowledge that would present a barrier to new engineers joining the team."* (P12) This is most likely a misconception, potentially caused by unclear writing in formal analysis tools' documentation, or scientific publications that do not separate tool use from general formal verification and theorem proving.

Developer Concerns and Wishlist (RQ5)

In addition to misconceptions, participants also voiced understandable concerns about constant-time development, as well as wishes for verification tools that would allow them to use these tools more effectively (RQ5). Major concerns were voiced about the tools' resource usage being too high (see Section 3.5.2).

In addition to these issues, P14listed concerns as: *"the execution time of static and dynamic analyzers tailored for SCA, the need for human interaction, the rate of false positives, etc. are usually preventing a systematic adoption"*. The issue with flagging false positives and not linking false positives and negatives was addressed by another participant also: *"We noticed a couple false positive, where there *is* a path from the contents of the buffer to timings, but we decided that doesn't leak any meaningful secret."* (P9). They also mentioned security concerns for tools based only on the source code. These may miss vulnerabilities due to miscompilation, as explained by P13: *"Any "constant-time" code is an endless arms race against the compiler"*.

Interestingly, participants had many precise ideas for what could be done to improve the status quo of testing/verification tools. For example, for better usability, they ask for the ability to ignore some issues and/or some part of the code, as noted by P14: *"Also, expect a lot of "noise" from BIGNUM behavior that is not CT and requires a full redesign to be fixed."*

We saw many wishes for improvements concerning annotations, asking for external annotations. Participants also asked for easy maintenance of code annotations (see 3.5.2), and requested that tools work on complex code, as P14explained: *"even for expert users the chances of exposing something non-consttime to remote attackers are high, especially given the complex nature of <library> under the hood."* They also asked for test cases to be fast to set up, to avoid a *"non-trivial amount of effort to set up comprehensive tests."* (P14)

To address the issue of scale, they want to be able to use tools in CI. Otherwise, when the code changes, the guarantees are lost. This means that error code outputs,

easy CI setup and runtime are important, as explained by P19: *"Static analysis tools tend to have a high engineering overhead: getting the tool to run, deploying it to CI systems, maintaining the installation over the years."* Similarly, participants demanded that tools not require rewrites of their code: P2ruled out an *"awesome tool"*, because it *"cannot verify existing code."* Participants also required no restricted language or environment for their code instead of *"a pretty special-purpose language"* (P26). Similarly, they asked for no use of a specialized compiler; as P4stated: *"Requiring a dedicated compiler sounds like a potential problem."* Generally, they asked for integration into type system and APIs they are already using: *"which values are public and which private, we have flags on APIs to allow the caller to specify this too"* (P28), so the project already has a form of security annotations for the users of their API, which a tool should be able to integrate for its analysis.

They also requested long-term available source code and longterm maintenance. As P25stated, tools being unavailable or unmaintained makes it impossible to use them.

3.6. Discussion

Based on our findings, we make suggestions for four groups of actors who can take action to make cryptographic code resistant to timing attacks: tool developers, compiler writers, cryptographic library developers, and standardization bodies.

3.6.1. Tool developers

In spite of the fact that we selected a subset of well-known tools from the wide diversity of available tools, 25% of the developers who answered our survey did not know about any of them. Some developers learned about the tools from our survey. Only 38.6% actually using any of the tools shows that their adoption is limited. This can be partially explained by the relative youth of the tools, as most tools are less than 5 years old. However, we believe that many other factors come into play: tools may be research prototypes that are difficult to install, not available or not maintained; they may not be evaluated on popular cryptographic libraries, raising concerns about applicability and scalability; they may be computationally intensive, making their use in CI unlikely; they may not be published in cryptographic engineering venues. In addition to the specific recommendations from the previous section, we recommend the community of tool developers to:

1. make their tools publicly available, easy to install, and well-documented. Ideally, tools should be accompanied with tutorials targeted to cryptographic developers; making a tool easier to install by providing Linux distribution packages lowers the barrier to adoption.

2. publish detailed evaluations on modern open-source libraries, creating or using a common set of benchmarks; Supercop [129] is one such established benchmark;

3. focus on efficient analysis of constant-timeness, rather than computationally expensive analysis of quantitative properties, which seem to be of lesser interest. Ideally, tools should be fast enough to be used in CI settings;

4. make their tools work on code with inline assembly and generated binaries to be fully usable by all developers.

5. promote their work in venues attended by cryptographic engineers, including CHES, RWC, and HACS.

Ultimately, we recommend tool developers to follow Reid et al.'s recent advice to "meet developers where they are" [303].

3.6.2. Compiler writers

Developers are very concerned that compilers may turn constant-time code into non-constant-time code. To avoid this issue, developers often use (inlined) assembly for writing primitives. This approach guarantees that the compiler will not introduce constant-time violations but may negatively affect portability and makes analysis more complex. In order to make integration of constant-time analysis smoother in the developer workflow, we recommend compiler writers to:

1. improve mechanisms to carry additional data along the compilation pipeline that may be needed by constant-time verification tools. This would allow cryptographic library developers to tag secrets in source code and use constant-time analysis tools at intermediate or binary levels;

2. support secret types, as used by most constant-time analyses, throughout compilation, and modify compiler passes so that they do not introduce constant-time violations, and prove preservation of the constant-time property for their compilers. This would allow cryptographic library developers to focus on just their source code;

3. more generally, offer security developers more control over the compiler, so that code snippets that implement a countermeasure (e.g. replacing branching statements on Booleans by conditional moves) are compiled securely.

3.6.3. Cryptographic library developers

Cryptographic library developers are aware of timing attacks and most consider them part of their threat model. In order to eliminate timing attacks, we recommend library developers:

1. make use of tools that check for information flow from secrets into branch conditions, memory addresses, or variable-time arithmetic. Ideally the use of such tools is integrated into regular continuous-integration testing; if this is too costly,

a systematic application of such tools for every release of the library may be a suitable alternative;

2. eliminate all timing leaks even if it is not immediately obvious how to exploit them. Attacks only get better and many examples of devastating timing attacks in the past exploited *known* leakages with just slightly more sophisticated attacks techniques;

3. state clearly which API functions inputs are considered public or secret. With a suitable type system, such information becomes part of the input types, but as long as mainstream programming languages do not support such a distinction in the type system, this information needs to be consistently documented. Doing so makes it easier to *use tools* for automated analysis and harder for programmers to *misuse* library functions due to misunderstandings about which inputs are actually protected.

3.6.4. Standardization bodies

A recent paper [38] advocates for the importance of adopting tools in cryptographic competitions, standardization processes, and certifications. We recommend that submitters are strongly encouraged to use automated tools for analyzing constant-timeness, and that evaluators gradually increase their requirements as constant-time analysis technology matures. Standardization bodies should try to avoid the use of cryptographic algorithms leaking timing information. In the case of Dragonfly Password Authenticated Scheme used in WPA3 by the Wi-Fi Alliance, many timing attacks have been discovered [372, 62] as the algorithm leaks timing information. However, many deterministic algorithms with no leaks are known [333].

3.7. Conclusion

We have collected data from 44 developers of 27 cryptographic libraries, and analyzed the data to gain a better understanding of the gap between the theory and practice of constant-time programming. One main finding of our survey is that developers are extremely aware of and generally concerned by timing attacks, but currently seldom use analysis tools to ensure that their code is constant-time. While constant-time testing may not be the most important thing on cryptographic developers' to-do list, it should become best practice. We think that this is only feasible by making tools more usable, supporting developers' current workflows, requiring little work overhead, and giving easy-to-understand outputs. Based on our survey, we have identified recommendations for tool developers, compiler writers, cryptographic library developers, and standardization bodies. We hope that these different communities will take up our recommendations and collectively contribute to the emergence of a new generation of open-source cryptographic libraries with strong mathematical guarantees. Although our recommendations are stated for timing attacks, we believe that many of

our recommendations remain valid in the broader setting of high-assurance cryptography. In particular, all our findings are directly applicable to the many ongoing efforts to protect against micro-architectural side channels, as summarized in [84]. Another interesting topic would be a quantitative analysis of the usability of some of the better known tools collected in this study to gain insight into the exact magnitude of the mentioned usability problems.

In this chapter, we gave an in-depth overview of the landscape of cryptographic constant-time verification and how cryptographic library developers interact with the tools academia publishes for them. We highlighted structural differences between the tools and asked detail questions about these classes and preferences in tool characteristics that the library developers wish to use. This is embedded in operational requirements of open- and closed-source software development projects, which give rise to additional requirements we highlight. In the end, we give only general recommendations for different target audiences, that we will sharpen with the study in Chapter 4.

4. A usability evaluation of constant-time analysis tools

Disclaimer

The contents of this chapter were accepted for publication as part of the conference paper titled "'These results must be false': A usability evaluation of constant-time analysis tools" at 33rd USENIX Security Symposium 2024. This research was conducted as a team with my co-authors Jan Jancar, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar; this chapter therefore uses the academic "we". The initial idea was developed from myself with Jan Jancar, Peter Schwabe, Gilles Barthe, and Yasemin Acar. We conducted the study with our other co-authors and jointly analyzed the data and co-wrote the paper for publication. This paper describes the state of cryptographic constant time verification tools up to the year 2023. It is an extension and logical followup to Chapter 3.

4.1. Motivation

Following Chapter 3 and the general overview of what the developers of cryptographic software libraries think and practice regarding constant-time analysis tools and code practice, some questions were still unanswered. For example, since the tools were seen as a hindrance more than a help, we wanted to understand how this could be changed, so we needed to do a mixed-method study with quantitative as well as qualitative data on how people fail to use these tools.

Our target audience would be the developers of cryptographic constant-time analysis tools, but we wanted to go one step further and publish our study setup, so that other practitioners could easily port our installation scripts forward to their own test setup, making it easier to get some of the tools running. We also decided to write and include documentation for some simple tasks, which we use in our study anyway as basic test of functionality and acclimatization to a tool.

Our participants in this study could not be practitioners themselves, or our results would be tainted by prior experience with some library or constant-time analysis tool. We decided to recruit among associated universities with a strong focus on IT-security, cryptography, and cryptographic engineering. The qualifying students were best in class in relevant courses to our study and recruiting this way took longer than anticipated, but our results were still representative due to the smal population size for cryptographic engineers in general.

As with the previous paper, we first dive in to why the problem space is still important, before we get into the details of the study itself.

4.2. Introduction

Timing attacks [211] are side-channel attacks that measure program execution time to infer information about confidential data. They are practical and can be used by (remote) attackers to achieve full recovery of secrets including cryptographic keys [73]. This makes protection against timing attacks an important goal for developers of cryptographic libraries.

In his seminal work, Kocher [211] observes that making control flow and memory access independent of secret data can help protect programs against timing attacks. Over the years, this guideline has become known as the constant-time discipline, and has become a gold standard for cryptographic libraries. Unfortunately, constant-time programming can be error-prone, especially when programming under stringent efficiency constraints, as is the case for cryptographic libraries. In 2010, Langley developed ctgrind [229], a minimal patch to Valgrind for checking that crypto software is constant-time. Subsequently, the security community has developed a broad variety of tools for protecting against timing attacks. Two recent works [199] and [152] provide an overview of these tools, from complementary perspectives. Jancar *et al.* [199] conduct a survey about the use of constant-time analysis tools with 44 developers of 27 widely deployed open-source cryptographic libraries. Their survey shows that these developers do not leverage constant-time tools despite an interest in writing constant-time code. As reasons, they identify that tools are not ready-to use and their use therefore requires significant time and expertise. Geimer *et al.* [152] presents a systematic evaluation of five selected tools, and identifies several technical roadblocks for the usability of tools. In addition, both works provide a systematic classification of around 40 tools for checking constant-time, and provide recommendations for tool developers and users. Although both [199] and [152] provide valuable insights on these tools, an empirical study to corroborate and deepen their findings has been lacking.

Therefore, in this work, we aim to understand **which factors support and hinder effective use of CT tools** through an empirical usability investigation that analyzes participant strategies while working with CT tools. Our investigation provides a complementary view on the issues discussed in [199]—which predates this work—and [152]—which was published after we completed the developer study. Our developer study is designed to provide deeper insight into usability requirements and how they influence their interaction with CT tools, to determine the features that tools should provide to achieve their full potential. Due to the broad range of tools, we designed a usability study with six CT tools. The participants of the study are 24 advanced CS students who had knowledge in cryptography (including about CT programming) and C programming. Our study comprises two phases: in the first phase, participants work through tasks escalating in difficulty while familiarizing themselves with a tool; in the second phase, they analyze real-world cryptographic libraries for

CT-ness.

We identify usability issues that we group into seven categories that revolve around three high-level aspects: (1) required efforts to setup and start using the tool, (2) barriers and work overhead hindering the use of CT tools, and (3) functionality the developer wants in analysis to identify and fix problems. We aim to answer the following research questions:

RQ1: *What are the pain points when trying to use CT tools?*

A: Installation, setup for analytic use, and (long term) operationalization in a larger library context are challenges for effective CT tool usage.

RQ2: *How helpful are the tools at discovering and fixing problems? Which tool properties help or hinder effective use?*

A: Tools can help cut down the amount of work needed to analyze larger code bases rigorously, but if a tool is too much work to install and get to work, cryptographers might just "eyeball" the analysis without the tool. Meaningful documentation to get a tool working on simple examples effectively helps to overcome this.

RQ3: *How can we support potential users in using the tools?*

A: Easy setup, a set of simple examples to appropriate for the markup (which should be minimal and noninvasive to the source code), a tutorial on how to use the tool and get clear information from the output, and good general documentation were all found to be helpful.

Based on our findings, we suggest how the usability of CT tools can be improved to make CT analysis more accessible to developers. In summary, our contributions in this paper are:

- We concretize the problems mentioned by experts in the Jancar *et al.* survey [199] through a developer study with 24 newly trained potential crypto developers and publish the full procedure material (see footnote 2) for replication.
- We offer a systematization of crypto developer workflow in using CT analysis tools, common to all 49 tools we found (see Table B.1).
- We document pain points and their impact on crypto developer usage of CT analysis, giving an explanation on why the findings of Jancar *et al.* [199] are still prevalent.
- We propose what to consider during development of CT analysis tools by contrasting prior attempts.

Supplementary material and disclosure. We have communicated our results to the authors of the tools included in our study and made the artifacts available to them. We have received four responses; all four expressed interest, one said they plan to link to our study materials in their project. The supplementary material, including tutorials, installation guides, and codebooks is publicly available on a dedicated web page[1] and as an artifact[2].

[1] https://crocs-muni.github.io/ct-tools/
[2] https://zenodo.org/records/10688581

4.3. Background & Related Work

We give an overview over the background and related work to this research by first discussing impacts of timing attacks on security, then describing CT development and CT analysis as defenses. For context, we also discuss a new generation of timing attacks that exploit microarchitectural features of CPUs, and the related efforts to protect against these attacks. Finally, we explain how a lack of consideration of human factors in cryptographic development can hinder widespread effective use of cryptography.

1. Timing attacks. Since Kocher's introduction of side-channel vulnerabilities in 1996 [211], these threats have persisted despite significant efforts to address them. Considering the vast range of side-channel attacks, we will highlight a few pivotal moments with a focus on timing attacks. Kocher's seminal work highlighted vulnerabilities in asymmetric cryptographic algorithms like RSA and DSS through "Timing Attacks", emphasizing the potential for exploitation based on secret-dependent operation times. In 2002, Tsunoo *et al.* [361, 360] expanded timing attacks to symmetric cryptography, noting vulnerabilities in MISTY1, DES, and suggesting AES being vulnerable to cache-timing attacks. Independent work by Bernstein [47] and Osvik *et al.* [283] confirmed these AES vulnerabilities. In 2003, Brumley and Boneh [73] revealed that these attacks could be conducted remotely via network timings. Subsequent vulnerabilities were discovered in the SSL/TLS libraries [71, 16, 80, 14] and on hardware-assisted defenses, such as Yarom *et al.*'s "CacheBleed" [402]. Kaufman *et al.* [206] also warned of persistent vulnerabilities post-compilation.

Despite these vulnerabilities and an emphasis on fixing them, side channels remain common in numerous platforms [240, 112, 63, 150, 149, 151, 176, 372, 62]. Some Common Criteria certified devices, despite their countermeasures, were found vulnerable [200]. Moreover, even recent post-quantum cryptographic efforts are affected [70, 289, 356, 284, 378, 167].

2. Constant-time Analysis. In this paper, we focus on investigating usability aspects of tools that evaluate timing leakages of (cryptographic) software. However, it is worth pointing out that the tools we consider also differ on a technical level in at least four different ways:

First, depending on the approach taken by different tools, they give very different soundness guarantees. Static *formal analysis* can achieve full soundness with regards to some leakage model. Slightly weaker guarantees are offered by tools performing *symbolic execution*; these tools achieve soundness only up to certain upper bounds on loop length. Tools based on *dynamic analysis* typically work with symbolic secret data but concrete public data; they achieve soundness up to code coverage for the concrete public values of the test cases. *Statistical analysis* performs measurements on (large sets of) concrete public and secret data. The advantage is that this approach does not require any leakage model, but on the downside, it also does not provide any soundness guarantees.

Second, the tools work on different levels of compilation. We distinguish tools working on source level, on some intermediate level, or on binary level. An example for a source-level tool would be the information-flow type system implemented by the secret_integers crate[3] in Rust. All tools we study (we will give detailed introductions later in subsection 4.4.2) in this paper work on either intermediate-representation (IR) of the LLVM toolchain [297] or on binary level. Tools working on IR level are inherently limited in the sense that they are unable to find any leakages introduced by the compiler when translating from IR to binary [206, 338].

Third, the tools working on binary level differ in what architectures and extensions they support. In order to be used on production code, they need support not just for the core instruction sets of widely used architectures, but also for vector instructions and dedicated crypto extensions.

Finally—and here is where technical features overlap with usability—the tools differ in terms of performance. For example, for the analysis of Langley's *"donna64"* implementation [230] of Curve25519 [48], the running time of just two of the tools we considered ranges between 0.38 and 225 seconds. This wide range may impact Continuous Integration (CI)/Continuous Deployment (CD) and developer workflows.

Table 3.1 presents the tools we found and categorized according to prior literature [198], appending a few tools previously not included; similar tables are found in [199, 152]. For each tool, we describe the target of analysis, the techniques used and whether the tools claim to provide some form of formal guarantees. We opted to err on the generous side of claimed soundness guarantees of each tool. For some tools the claims do not easily map to the soundness categories we discussed before, so we keep the unqualified "Other" category from the literature. As usual with this kind of classification, the categories are not exclusive, each tool may combine approaches in its design—we opted to continue with best-effort categorization like the established literature.

3. Microarchitectural side-channel attacks and defenses While constant-time programming is still an important and increasingly standard baseline defense against software-visible side channels, research on more advanced microarchitectural attacks in the past few year has shown that this programming discipline is not a sufficient measure. This line of research started with the 2018 Spectre [210] and Meltdown [239] attacks, and has since identified multiple pathways for attacks that often—but not always—exploit speculative execution in modern CPUs. See, e.g., [214, 383, 282, 258].

The notion of constant-time can be extended to protections against more advanced microarchitectural attacks [210], leading to notions of speculative constant-time [82] or more generally of security with respect to a hardware/software leakage contract [262, 189, 259]. Many of the techniques used for analyzing constant-timeness can be extended to reason about speculative constant-time and related notions. In fact, there is already more than two dozen tools that analyze whether a program satisfies (some variant of) speculative constant-time. For an overview of these tools see [83, Fig. 2];

[3] See https://docs.rs/secret_integers/.

they generally suffer from similar usability issues as tools for constant-time.

Recent work [375, 374, 235] shows that aggressive optimizations used by modern CPUs to improve performance can lead to a new class of timing attacks. Many of the leakages are data-dependent and depend on prior execution history, making their detection extremely challenging. As a consequence, there is a strong incentive to develop analysis tools for checking the counterpart to constant-timeness; see [43, 139] for two very recent examples.

In both cases, we believe that the insights gained from [199, 152] and our work will provide valuable input for improving the usability of future tools in this space.

4. Human Factors in Cryptographic Development. There is a large body of work on human factors in cryptographic development. Acar *et al.* establishes in a 2017 study that poor usablitity of cryptographic libraries contributes to misuse and insecure code [6]. Haney *et al.* investigate the mindset of cryptography developers [172], and observe that some developers do not adhere to mainstream software engineering practices.

Krueger *et al.* developed a wizard for secure code snippets for specific cryptographic applications, evaluating its effectiveness and usability in a programming study [217, 219, 218].

In the specific context of constant-time tools, a study by Cauligi *et al.* [85] was carried out with over 100 students to understand the benefits of the FaCT tool introduced in the paper. The tool support by FaCT is found helpful for generating new code that is CT. In extension of this work, we include a diverse set of CT tools, documentations, tutorials, as well as open source libraries in our study.

Unfortunately, while previous research suggests that lack of usability prevents effective use of security tools [92, 121, 158, 320, 376], and specifically for CT [199], the question of how to improve the usability of these tools has been understudied [8].

4.4. Usability criteria and tool selection

In this section wegive a general description of our usability criteria, and explain how they impact users. In addition, we briefly introduce the six included CT tools in our study, organizing our presentation to inspect the previously defined criteria for each tool.

4.4.1. Usability criteria

The main purpose of our evaluation is to assess the usability of current CT tools, and identify features that impact effective use. To expand on Jancar *et al.* [199], we define criteria revolving around three features: (1) the effort required to setup and familiarize, (2) the work overhead for secret designation and target building, and (3) the quality of output to identify and fix problems.

We define our criteria following how users would perform the tasks related to CT analysis [352]: how users might interact with the tool, what information is given to the user, and how analysis outcome is presented to the user. The categorization of CT testing workflow steps was created from our expert team's experience in building CT tools and using them on real-world projects, combined with insights gained from piloting the study. We developed the categorization after all of the study results were gathered.

Installation. Every tool needs to be installed before use. There are two broad ways of installing CT tools. Some come pre-built and bundled for a package manager or in a container. Others involve manual installation by either building from the source, or by grabbing the available binary from a release page. For the latter method, the developer will be in charge of managing the necessary dependencies manually. CT analysis tools mostly come as proof-of-concept artifacts. According to [184], only 3% of artifacts are distributed in containers, while 23% are pre-built and 70% must be compiled from source code. Therefore, we expect that the installation step of CT tools may be very challenging for numerous tools, especially because of unmaintained dependencies, also confirmed by Jancar *et al.* [199], who point out that libraries maintainers do not consider use of hard-to-install CT tools.

Familiarization. Documentation is intended to provide a high-level overview of the tool and offers technical details for expert users. Help materials also include tutorials and examples. In this criterion, we focus on how quickly new users become comfortable running a tool on simple programs.

Building and Secret Designation. CT tools provide a means to tag secret data. This is typically achieved via either *code annotation* or the creation of an external function *wrapper*. Many CT tools operate on instrumented binaries or some abstract intermediate representation that is designed for program analysis. Very often, this implies a custom building and linking process. Usability is negatively impacted whenever manual work is needed during this process. In other words, we look at how much tools modify a project to be analyzed: both in terms of code (for secret designation) and build workflow integration (for target generation). Little work overhead is commonly appreciated [204].

Analysis Runtime. Once the target is built, users can actually run the tool for CT analysis. Here, we look at two sub-criteria, the tool's interface and its runtime. For a command-line interface tool, users may struggle with passing the right options. Importantly, tools are expected to yield results in an acceptable time frame. The longer the runtime of the analysis, the more difficult it is to integrate the analysis into the project workflow [204]. This problem hinders a feedback loop using CT analysis at coding time. This can be important both in CI workflows, which may have an upper time limit, and developer workflows, where each developer may only want to spend a small amount of time waiting for analysis results.

CT Problem Fixing. When the analysis is finished, CT tools display some output to direct the developer's attention to detected issues. The purpose is to provide the

developer with enough information to judge whether or not they care about the issue, and if yes, why the tool reports it. For example, it is not helpful if tools just display that there is an issue without any detail about the origin of the leakage. In addition, it is more productive for developers to be able to navigate and manage the list of reported issues. Otherwise, developers must linearly search through the (potentially large) list of results, making selective fixing more difficult.

Specialized Output Generation. To improve the experience of fixing problems, users might require customizing the generated analysis output. We introduce two features that we identify for CT tools. First, tools should also offer different verbosity in report details to avoid *excess of information* [158]. For example, a summary mode is beneficial in order to quickly skim the reported vulnerabilities to decide which one to inspect. Second, within the context of a CI pipeline, a delta report can be handy in assisting developers to determine whether a specific leakage has been correctly patched, and that the fix has not induced other leakage.

Reliability / False Positives. Ultimately, users need to trust the tool and its analysis. Therefore, any indication of potential false positives or missed issues could undermine user confidence, leading to tool abandonment. Solutions do not necessarily involve sound or complete tools, but also support for filtering user-supplied false positive patterns. This may help the user but can also lead to user filtering actually missing timing leaks, either mistakenly or lazily.

4.4.2. Tools

We selected six tools for use in our study: MemSan, timecop, dudect, ctverif, Binsec/Rel, and haybale-pitchfork. These tools were primarily chosen to include a representative from each analysis type. The selection of the tools was made towards the end of 2022, therefore more recent tools were not considered. We prioritized tools well-recognized in the community, ideally those used by developers, gauging their reputation through a recent survey [199]. Out of the tools, 4 (ctverif, MemSan, dudect and timecop) are 4 out of top 5 most known tools in [199], with the top one being ctgrind, which we replaced with the functionally equivalent and still maintained timecop. haybale-pitchfork and Binsec/Rel were selected as representatives of other tool approaches. The number of tools was also constrained by participant numbers to ensure even distribution. At the end of the subsection we compare our choice of tools with the five tools chosen in [152].

MemSan [346]. MemSan is designed to leverage the Clang built-in memory sanitizer to dynamically analyze binaries for constant-time violations, thereby requiring Clang for installation (which is available in most Linux package managers). Clang sanitizers are well documented, but there is little documentation on how to use MemSan for CT analysis. Concerning secrets, users can declare private variables and/or memory regions containing secrets, and declassify variables within certain code sections if required. To run the tool, developers must compile the program with Clang, using the

appropriate option to enable the memory sanitizer. All parts with no enabled sanitization are ignored—it is easy to get this wrong. Then, the analysis is performed by running the resulted binary. Note that only the executed code is analyzed, leading to different conclusions when running the same binary with different inputs. Indeed, code coverage is essential for MemSan. Upon the binary execution, errors will be displayed on branching or memory access indexing an annotated variable. The output details the path between the annotated variable and the cause of leakage. The output messages will be more related to the source code if the target is compiled in debug mode.

timecop [264]. Similar to MemSan, timecop relies on the Valgrind memcheck module [116] to dynamically analyze binaries for CT violations. Therefore, for installation, it solely requires Valgrind (which is available in most Linux package managers) and an additional C header file that must be downloaded from the project page. The timecop page also contains several tutorials and examples to smooth its first uses by beginners. To analyze code, users need to annotate private variables in the source code and may declassify variables within certain code sections if needed. There is no need for changes in the compilation chain. Concerning the analysis, users can simply run Valgrind on the binary as if they were searching for memory leaks. Valgrind will raise warnings for CT violations just like it would for the use of uninitialized memory in a branching or memory access. The output details the path between the annotated variable and the cause of leakage. timecop relies on the debug information to display the lines of code in its warnings. With its use in SUPERCOP [129], it is widely used.

dudect [307]. Installation for dudect is virtually non-existent as the tool is provided as a simple archive containing the C header file implementing it. The dudect documentation is rather limited. The tool operates via a black-box evaluation of a function, obviating the need for code annotation. The user, however, is required to implement an external wrapper in charge of setting the analysis parameters and options, as well as two functions to initialize the secret input classes and call the code to assess, respectively. Then, the target program must be compiled (with no custom build) and executed for analysis. The dudect approach is statistical, and it thus outputs values of statistics after code analysis. The output does not underline any source leakage, but only some probabilistic conclusion about the target CTness.

ctverif [19]. The installation of ctverif presents a significant challenge, requiring undocumented versions of specific dependencies and manual patches across different projects, such as SMACK and Bam-Bam-Boogieman. Aside from the paper, there is no or little documentation available. As for secret designation, users must declare private and public variables and/or memory regions (arrays) containing secret or public inputs. In addition, they could declassify outputs, and assert the non-overlapping nature of these regions. The tool operation is straightforward, requiring only the source code file as input, in addition to the entry point to analyze. Thus, ctverif does not need any custom build. However, ctverif can process a C translation unit only when all the called functions inside are defined by other input files, otherwise it produces an unknown error. After a run, ctverif only highlights the leakage location in the source

code, without a dependency chain of variables or memory locations that lead to each leaked secret. Surprisingly, ctverif may raise some warnings even after a successful run without CT violations. It is worth mentioning that ctverif, instead of making some approximate analysis, informs developers when it cannot conclude about some leakage, displaying inconclusive output.

Binsec/Rel [111]. Binsec/Rel comes as source code, an extension to the Binsec tool. Some dependencies, such as an SMT solver and the OCaml package manager, shall be installed manually, before compiling the project source available on GitHub. Binsec/Rel offers a comprehensive list of supported command-line options and numerous examples to start with. On the analyzed project, users shall employ markup declarations to annotate the source code, thereby designating public and private data. The analysis of Binsec/Rel operates over binaries. The version utilized in this study only supports ARM 32 and x86_32 architectures, necessitating the target to be compiled accordingly. This might require to add additional compiler flags, since in numerous compilers, the default mode supports 64-bit. Upon completion of the analysis, a report is produced including the number of CT violations and an assembler dump correlating with the violation location. The assembler dump does not point to the leaked secret, but only to the instruction causing the leakage. Note that during our study, Binsec/Rel received a major update that integrated the CT checking functionality into the main tool Binsec.

haybale-pitchfork [363]. Written in Rust, haybale-pitchfork can be installed from source using cargo, although its dependencies must be manually installed beforehand as documented on the project page, which includes multiple examples and different documentation materials. haybale-pitchfork runs its analysis over the LLVM intermediate representation. Thus, users need to modify the compilation chain to produce the corresponding LLVM bitcode of the target. Any symbol in the generated bitcode must be correctly resolved, or haybale-pitchfork stops the analysis, while printing a message raising "other errors". Instead of relying on annotations to mark secrets, users are instructed to implement an external wrapper in Rust, in order to define an abstract signature of the target function. Here, each function parameter can be declared as public or secret using the appropriate Rust type. This wrapper also contains other configurations, such as the bitcode path to inspect. Users carry out the analysis by compiling and executing the Rust wrapper. haybale-pitchfork provides conclusive results, displaying the leakage origin whenever a CT issue is found, together with a tree path to the leaked secret.

Comparison with the tools of Geimer et al. [152]. Geimer et al. [152] explores five tools in depth: Abacus [37], Binsec/Rel, ctgrind, dudect, and MicroWalk-CI [397]. Two of these tools (Binsec/Rel and dudect) are also included in our study. As explained above, we selected timecop and MemSan over ctgrind, because the ctgrind patches are outdated and do not work with recent versions of Valgrind and the Linux kernel anymore. In contrast, timecop and MemSan can be seen as more usable versions of ctgrind. We did not select Microwalk-CI [397], because it was released after we had initiated our study. We also did not select Abacus, because its focus is quan-

titative information flow rather than constant-timeness. We included ctverif for its strong correctness and coverage guarantees. We also included haybale-pitchfork, as an instance of a tool that covers both constant-time and speculative constant-time—however, to our knowledge, the tool was eventually not extended to speculative constant-time.

4.5. Methodology

In this section, we provide details on the procedure and structure of the study we conducted with (initially) 31 participants. We describe the experimental setup including choice of libraries, surveys, and experimental infrastructure. We also describe our coding process of qualitative data, including participant behavior and free-text responses, as well as the approach for statistical analysis of quantitative data, such as success measures and quantitative survey items. Finally, we explain our data collection and ethical considerations, and discuss the limitations of this work.

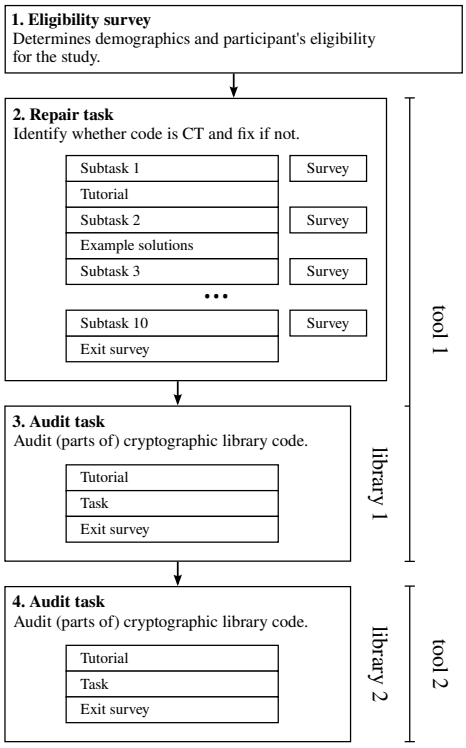

Figure 4.1.: Study flow for each participant.

1. Recruitment and Participants. Due to the challenges of recruiting professional developers with security expertise, we targeted CS students for participation. We engaged Master's and early PhD students from 5 universities across 4 countries, as recommended by [350]. From the 74 students we approached with an eligibility survey, 31 began the study. A tools assignment error led us to exclude one participant.

Eligibility Criteria. We only consider participants having the minimum knowledge necessary to run CT tools. We asked them to self-report their knowledge of the C programming language and the CT paradigm. Naturally, considering our usability criteria, we also verify that they had never used any of the tools that are part of this study. Finally, to distinguish our work from [199], participants needed to lack experience in working on production-quality cryptographic code.

Compensation. Participants were compensated with 200 euros on completion of the study in exchange for 16 hours of participation; this compares to the hourly payment for student research assistants in participating countries.

2. Study Procedure. After a short eligibility and demographics survey which preceded the study, we asked participants to assess code for vulnerability to timing attacks using a CT verification tool across ten tasks that escalated in complexity, as well as to audit (parts of) two cryptographic libraries. Assignment of participants to tools and libraries was done by hand, following a pattern of complete coverage of all possible tool combinations.The study flow is described below as well as visualized in Figure 4.1.

During the study, participants were assigned ten successive *repair tasks*—tasks building upon tool-support specifics tested in prior tasks, which we will explain in detail—in which they were instructed to use a pre-installed CT analysis tool (tool 1) to identify whether a given code snippet is CT regarding a well-defined secret. If the code was not CT, participants were asked to fix it. The repair tasks represent textbook examples of secret-dependent branching and memory access, and their CT variant. After working on the first task, participants were given a tutorial that we had written for the tool. After the second task, we gave them the solutions to previous tasks, to be used as examples. Tasks 3 to 8 added various elements to increase difficulty, such as calls to libc functions (memcmp), reading randomness from the operating system and particular source code designed to trigger optimization during compilation. The goal of these repair tasks is to assess the participants' ability to use the tool to evaluate and fix a rather simple code snippet.

After completing work on the ten repair tasks (or exhausting the allotted time of 8 hours), and so becoming familiar with the tool, we asked them to audit well-known cryptographic libraries using the same tool (tool 1). In this *audit task*, participants were asked to compile the library (library 1) in such a way that enables them to use the tool, and audit a much larger code base. They were pointed at potentially interesting parts of the library, but not at specific functions. After a first library audit with a tool they had used for the entire study up to that point, we provided them with a new tool (tool 2), a tutorial, and a new library (library 2) to start a second audit task. These audit tasks aim at assessing the tools' usability in a setting more closely resembling a real-world use case.

For both parts of the study, we consider that a participant successfully completed a task if they underline the CT violation using the respective CT tool, and recognize it as such to fix it. The task structure was monolithic, simply stating that the task was to find CT violations with the given tool. Participants had to find out the necessary steps themselves.

After each of the repair tasks, participants were given a brief survey asking about their results (was the code CT or not, etc.), their experience with the tool during the task, and issues they encountered. After the last of the repair tasks, we gave participants a longer exit survey, which included the System Usability Scale [68] and questions regarding their overall experience with the tool. Participants were asked, e.g., whether they trust their tool to give them correct results and what their biggest problem was while using it. A similar survey was included after each audit task.

Instrument Development. Our group of authors consisted of experts in cryptographic engineering, side-channel attacks, and CT tool developers, as well as one human-

factors researcher. We based the study development on our usability criteria and related features. We also let our experience with the development of cryptographic libraries and CT verification tools (as authors as well as users) influence the study design. The human-factors researcher introduced and facilitated the use of human-factors research methodology to better explore the identified usability criteria. In particular, the human-factors researcher explained methods when appropriate, facilitated discussions and helped the team to develop the study, pilot it, gather feedback, and evaluate the results.

Pre-Testing. Three co-authors dry-ran the study, followed by one student from the targeted population. Using their feedback we updated, expanded, and clarified the study.

Time Frame. Every participant had a recommended and self-enforced time limit of 8 hours to work on each part of the study (repair and audit, thus a total of 16 hours), within a soft frame of 2 weeks. We allowed extension of the 2-week time frame. Participants, although encouraged to fully use their time, were allowed to hand in their results earlier.

Repair Task Details. The first four of our tasks demonstrate the main points of the CT criterion: Secret-dependent branching and secret-dependent memory access. Repair tasks 01 and 02 are non-CT and CT examples of a selection based on a secret value, once with a branch and once with an arithmetic transformation like the one presented by Schwabe at ShmooCon 2015 [329]. Repair tasks 03 and 04 are likewise memory accesses depending on a secret value or boolean arithmetic for selecting a value loaded from all addresses without depending on the secret for the load address.

Repair task 05 introduces the use of a C programming language standard library function, memcmp, which is non-CT, to compare secret values. Tools which depend on static binaries and cannot inspect dynamically loaded libraries—which are the majority of deployed software today—are expected to fail here and show no CT violation. Repair task 06 includes a system call to read random numbers. System calls are on most operating systems implemented in a way that cannot be seen from user space, the memory area that is analyzable to most CT analysis tools. The tools can work around this, for example by recognizing a set of known system calls and their expected behavior. This task greatly differs from previous tasks, as it does not include secrets, but only checks for support analyzing this code. Task 07 starts to build up problems toward a harder criterion than CT - probabilistic CT, which is a criterion for functions that behave CT by default except for a subset of cases. Indeed, the program reads a random number like in task 06, but in 1 out of 256 cases, it will perform secret-dependent branching like in task 03. This may sound easy to spot manually by most users, but statistics-based CT tools were expected to underperform on this task. Task 08 introduces a different, and on first sight trivial problem: the same function is called, but in two branches based on the value of a secret. This may seem to be CT, but in practice a compiler may transform this into assembly code that does not branch on the secret, even though in the given source code the CT criterion is violated. The intent behind this task is to see if the abstraction level of a tool, whether it works on binaries or instrumented source code, has a measurable impact on the success of participants.

Task 09 makes the compiler transform impossible by changing the branching structure, passing a secret variable as a function input. The called function just returns a constant, which makes the whole program CT. Finally, task 10 is distinguished from previous tasks. It is formally non-CT and can be repaired in two non-obvious ways: users can either make it CT, but only probabilistically correct, or correct, but only probabilistically CT. With this last task, we wanted to see how participants pick up on less trivial code, inspired by techniques used in some cryptographic algorithms recently standardized by NIST, such as Kyber [57, 39] and Dilithium [127].

3. Study Setup. Our tool selection is explained in subsection 4.4.2. Note that one of the included tools (Binsec/Rel) did receive a substantial update during the study, that we did not include as not to invalidate our study.

In order to have similar working environments, we deployed one VM for each tool, and gave SSH access to the participants. Each participant had restricted access to their home directory, with all necessary material (such as instructions and source code of the task and the library) available. For each resource, a clean copy was available as read-only in case they needed a fresh start. We decided to pre-install the tools on the VMs. The reasoning for that choice is twofold.

First and foremost, most tools are the outcome of academic research, and served the purpose of demonstrating new techniques and approaches, without aiming for maintainability. Hence, some tools are not maintained, and rely on specific version of dependencies that are outdated and deprecated. This can make the installation particularly complex and time consuming, especially on recent systems. Second, given that participants using the same tools were co-located on a VM, we could deploy the tool globally to ensure a functional setup, and avoid unintentional corruption of the tool by participants.

As an effort to make the first step easier, we implemented installation scripts for each tool present in our study—for possible difficulties in the installation phase, see Reynolds et al. [312]. We made them publicly available (see footnote 2), along with the repair tasks and a small functional tutorial we provided to the participants. We hope this can prove useful, and motivate tool developers to do the same.

To make sure the participants have something to find in the audit tasks, we needed to include libraries that had problems with CT-ness, therefore we chose the following: two of them—OpenSSL and GNUTLS/Nettle—were chosen because they are in ubiquitous use in open-source software projects. The other—mbedTLS—was chosen because it was common *and* is targeted more for use on embedded devices. Other libraries like BearSSL were not included due to fewer documented CT issues and fewer prior audits of those libraries compared to the first two. We specifically audited the libraries ourselves, first, to see if participants can meaningfully find code that is non-CT in those libraries, either by looking at public documentation and then verifying with a CT verification tool, or direct analysis. All three chosen libraries document which parts of their code bases are not expected to be CT, so our participants could be expected to find them.

4. Coding and Analysis. All qualitative coding and data analysis were done by multiple researchers from a set of four, each coding part done by at least two, from diverse backgrounds and views. All of those researchers were familiar with CT verification, open-source and cryptographic code development practices, while two researchers had additional experience with human factors research with developers. We followed the process for thematic analysis [65]. The four coders familiarized themselves with the free-text answers in their part of the analysis, adding annotations and developing themes as well as codebooks.

Codebooks were first developed deductively based on the questions on each subtask, then changed inductively. The codebooks were iteratively changed while extracting themes from the free-text answers. Coders discussed until agreement was reached to make unanimous decisions; we therefore do not calculate inter-coder reliability [246]. The codebooks codify experiences—good or bad—as well as misconceptions, insecurities, and wishes encountered during the study's surveys.

5. Data Collection and Ethics. Our invitations were sent to participants of thematically fitting courses of five participating universities. We invited students by emailing them individually. During and after the study they could opt-out of participation. We only linked participants names to results for payment, not during analysis and not by members of the research team who had prior contact to those students. We keep the participant responses as confidential as possible and do not link quotes to them by name, only by pseudonyms.The study protocol and consent forms (for study participation and surveys) were approved by our lead institution's data protection officer and ethics board, who determined that the study poses minimal risk. Identifying data of the participants, like names, email addresses, and payment information, were stored separately from study data, and were only used to contact the participants; we did not retain any identifying data in excess of following laws.

6. Data cleaning & Presentation. From the 74 students we invited, 31 started our study, of which we were able to use the results of 24 participants. We only evaluate the results of participants who finished a meaningful part of our study and compared results with and without familiarization with each tool on each library to offset possibly bad pairings, but did not find any meaningful differences between the two groups. As for the 7 incomplete results, we were not meaningfully able to incorporate them in most of the statistics—to not over represent results from simpler tasks—but we used partial results that were complete in appropriate sections.

Participants were paid for and expected to spend two days of eight hours each on the study, leaving rich free text comments in the surveys as well as comments in source code of their task solutions. We received mixed feedback, from disillusioned responses to high interest in further research on CT verification and coding practice. Generally, the feedback to our study was positive, even when the comments about the experience with some tasks were less so.

7. Familiarization. By design of our study, we set our participants up for familiarization with one tool each, then we ask to analyze a common real-world cryptographic library with the same tool. The repair tasks during the familiarization procedure were optimized for familiarization with the tool from simple examples to simplified current research problems.

8. Limitations. Survivorship bias[231] might taint the results, due to the study not reporting all the results of participants which dropped out. Selection bias due to comparatively high requirements in recruiting for the study as well as selective perception due to recruiting from student population who is accustomed to writing exams and tests might both also be relevant, but are both similar to the population which might use one of the CT tools. Participants may have reported more familiarity with the subject matter than they actually had, but due to our recruiting criteria, this was limited to a minimum actually necessary for participation. Our study may also suffer from the typical effects of fatigue in participating in a study, frustrations, and, of course, took place during the later years of the COVID-19 pandemic. Finally, our low participant numbers (due to the significant time investment and prerequisites) does not allow for statistical inference; we report numbers to highlight trends and/or outstanding observations.

Problem Fixing. When participants marked a repair task as already constant-time they were not asked to fix the code.

Library Selection. The projects we included for the audit task represents a selection and are not representative of all open-source cryptographic libraries. We are aware that other libraries might lead to different usability results.

Unknown Code. Our participants were not familiar with the cryptographic libraries used in this study. Annotating and custom-building are likely to be different when analyzing a project the participants are familiar with. Developers might achieve different results if they have a rough overview of the code base. We expect completing the repair tasks to be easier to our participants than the open-ended audit tasks.

VMs, Tutorials, and Examples. By including ready-made virtual machines with each installed CT tool, combined with layered introduction of tasks and documentation, not restricting online documentation and providing some as a backup ourselves, we provided our participants with a best-case scenario to learn how to work with each of the tools. Participants could approach the study as they saw fit, while being able to adapt example solutions and their own prior solutions to everything after a first introduction to the base cases for CT programming practice in minimal examples (tasks 01 to 04). This was a trade-off to gather more data about all CT analysis steps, not being stuck over installation or finding documentation. Nevertheless, this means that our participants had an easier task with the tools than users would have in real world.

Tool (Tech., Guar.)	Repair		Audit 1		Audit 2	
Binsec/Rel (Sy, ◐)	33.5	(3.8)	38.7	(11.8)	45.6	(7.2)
ctverif (F, ●)	30.6	(18.4)	34.4	(8.5)	31.5	(14.8)
dudect (St, ○)	53.1	(29.1)	65	(5.9)	59.4	(23.8)
haybale-pitchfork (Sy, ◐)	64.4	(6.6)	52.5	(13.7)	50.6	(26.6)
MemSan (Dy, ◐)	49.5	(20.3)	41.3	(22)	49.4	(20.1)
timecop (Dy, ◐)	71.2	(6)	69.4	(10.3)	70.6	(24.1)

Table 4.1.: Average and standard deviation of System Usability Scale scores from exit surveys after repair and audit tasks.
Technique: Sy—Symbolic, St—Statistics, Dy—Dynamic, F—Formal
Guarantees: ●—sound, ◐—sound with restrictions, ○—no guarantee

4.6. Results

Table 4.1 showcases the System Usability Scale (SUS) scores [68] for each tool on both repair and audit tasks. The SUS is supposed to give a quick overview of a tool's overall usability; a score above 68 would be *"above average"* across software types. From the scores presented, usability remains fairly consistent between repair and audit, with notable exceptions for haybale-pitchfork, which had a noticeable dip during audits, and dudect, which exhibited enhanced usability in the audit tasks. Among the tools, timecop has the highest and most consistent score, suggesting superior usability. In contrast, ctverif and Binsec/Rel emerge as the least usable. For the correctness of solving the repair tasks, see Table 4.2.

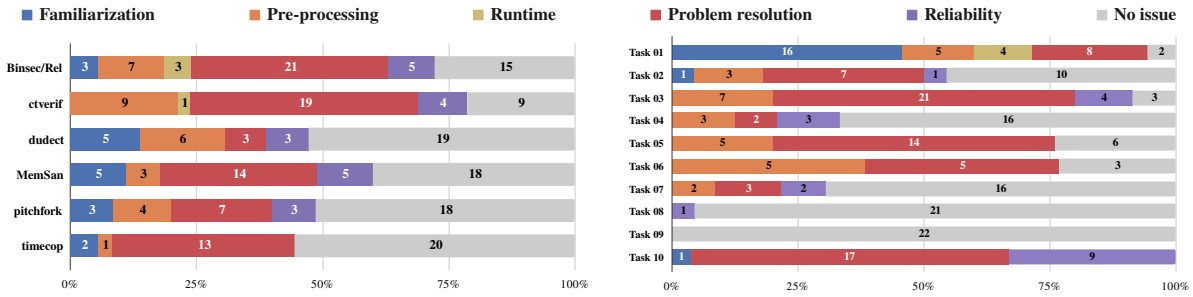

Figure 4.2.: Participant's major issues during the repair tasks. (Left) For tools over all tasks. (Right) For tasks over all tools.

Through thematic analysis of feedback during repair tasks, we identified common usability issues with the tools. Feedback points, categorized according to our codebooks, often overlapped, except for the "no issue" category. The distribution, depicted in Figure 4.2, gave insights into tool perceptions and task challenges.

In section 4.7, we delve into the diverse factors impacting usability, as organized by the criteria introduced in Section 4.4.1, and report on participants' confidence in their results. Each criterion corresponds to a step in detecting/fixing CT violations,

and each subsequent step depends on the success of its predecessor. Those who encountered initial setbacks often did not report in the later stages. This was particularly pronounced when auditing real-world software libraries.

Our findings spotlight factors affecting the tools' usability: Clear and intuitive outputs stood out as extremely important, and a lack of beginner-friendly documentation emerged as a recurrent issue. Though well-structured documentation is invaluable during the familiarization phase, participants reported distinct challenges as they delved deeper into the tools, but mixed with positive feedback as well. This disparity became evident when contrasting feedback between standard textbook examples and real-world audits, emphasizing different stages of tool assimilation.

Although the participants were equipped with the tools for the tasks, their installation and setup experiences could not be included in our data as we set up the tools for them.

1. Familiarization. During the first task, the main issue reported was unclear and non-user-friendly documentation, with 16 complaints (18 overall). Although the tools had associated academic papers, participants felt these didn't serve as effective documentation. They particularly missed step-by-step setup and results interpretation examples. *"[T]he documentation about every command doesn't exist or I didn't find them. Maybe a beginner-friendly aspect of the tool would have been good for me to start."* ()

Notably, participants had no complaints about ctverif documentation—possibly due to its basic user interface—but most of them faced issues with its operational aspects until they consulted our tutorial.

Despite the issues, our study also highlighted successes in the familiarization phase. Concise, beginner-focused documentation was identified as a significant upside in enhancing user engagement. The turnaround is likely a direct result of the tutorial we provide upon the completion (or non-completion) of the first task.

None of the participants managed to solve the second task using ctverif, and all expressed complaints about the output. After the tutorial and solution were provided, 3 out of the 4 participants were able to solve the subsequent task. This improvement persisted through the remaining tasks and can be attributed largely to the alleviation of difficulties in correctly interpreting the output and running the tool.

The effect was similar with Binsec/Rel. While none of the participants solved the first task, 4 out of 5 successfully solved the second task following the tutorial.

We noticed that our tutorials had a particularly strong impact on the usage of dudect, a tool that elicited the most complaints about lack of documentation. One participant even expressed their appreciation with the following: *"Great tutorial about dudect on the previous study page. Why it is not included in the official documentation?"* ()

Overall, the tutorial was appreciated for every tool in the repair tasks, as suggested by the following quote. *"I am just really using the template provided in the tutorial"* (),

During the audit task, 15 struggled to start using the tool, despite our tutorial. This was especially true for Binsec/Rel (3), ctverif (5) and haybale-pitchfork (4). Complaints mainly referred to lack of guidance in more complex tool usage, such as hook-

ing functions, or bypassing some tool limitations. Tools with a more straightforward functioning, such as timecop and MemSan, did not suffer from these complaints.

2. Building and Secret Designation. This crucial preprocessing step is fraught with complexity, leading to 30 complaints from the study's participants during the repair task.

Central to the participants' challenges was the task of designating the secret within the given code snippets. The complexities arose either from the need to annotate the code, leading to 17 complaints, or the requirement to design a wrapper, which received 7 grievances in total. In particular, the annotation APIs provided by Binsec/Rel and ctverif were deemed overly complicated. This perspective was substantiated by 7 and 9 complaints, respectively, suggesting poor usability. In contrast, MemSan and timecop offered more streamlined processes, simply enabling users to flag a memory region as secret. The challenge of implementing external wrappers for tools like dudect and haybale-pitchfork was accentuated by insufficient documentation, evidenced by 4 and 3 feedback reports. A unique challenge presented by haybale-pitchfork was its reliance on the Rust language, which impeded 3 participants. This prerequisite even pushed one to abandon the study. Hesitance to continue, even when participants were provided with ready-to-use tools and monetary encouragement, underscores usability concerns for the target user base.

Interestingly, the audit tasks unveiled a new set of foundational hurdles. A seemingly rudimentary step - local library installation - became a roadblock for 13 participants across all tools. While participants found the compilation of minor repair tasks with specified options manageable, the challenge escalated when they had to adapt intricate compilation chains to enable the tool use. In this regard, 16 participants faced hurdles when gearing up the libraries for suitable compilation to enable analysis. The architectural constraints of Binsec/Rel, especially the need to compile libraries for a 32-bit architecture, caused difficulties for 7 participants (given the study reliance on an older tool version). haybale-pitchfork posed its unique challenge, with 5 participants coping to generate the necessary bitcode of the library. The tools dudect and haybale-pitchfork added another layer of complexity by necessitating external wrappers, proving problematic for 4 and 1 users. The demands of accurate code annotation further intensified the complexities during this phase for 4 participants. This was notably severe for Binsec/Rel users and MemSan, 2 reports each. Overall, 10 participants faced significant hurdles in advancing further in the library audit, and did not manage to run the tool. 6 of them were blocked when using ctverif.

3. Analysis Runtime. In the context of the repair tasks, while many tools were wielded effortlessly on multiple tasks— indicated by the "no issue" category in Figure 4.2—both Binsec/Rel and ctverif manifested signs of a higher barrier, even for tasks that appeared superficially straightforward. Specifically, Binsec/Rel was utilized seamlessly on 15 occasions, whereas ctverif demonstrated hassle-free operation only 9 times. We want to highlight the particular difficulty participants faced with Binsec/Rel during

the first repair task. Users were presented with a multitude of options, some of which tangential to the main task, leading to 3 complaints.

The audit phase, characterized by the need to analyze larger code bases, brought forth a different set of issues. The time-consuming nature of the analysis was a concern, particularly for haybale-pitchfork and dudect. Analysis processes were identified as overly protracted by 1 and 2 participants respectively. This drawn-out analysis underscored concerns over the efficiency and practicality of these tools in real-world settings.

4. CT Problem Fixing. A preliminary glance at the success metrics in utilizing the tools, referenced in Table 4.2, exhibited significant disparities among the tools. Some adopted a tool-reliant strategy, while others, having initially engaged with a tool, later pivoted to manual code analysis. Given the easy nature of most tasks, forcing participants to resort to manual analysis is a witness of poor usability. We recorded these events mostly with ctverif, Binsec/Rel and dudect.

Tool	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7	Task 8	Task 9	Task 10
Binsec/Rel	0%	40%	80%	100%	80%	40%	100%	80%	80%	60%
ctverif	25%	0%	75%	75%	100%	50%	67%	67%	100%	33%
dudect	60%	100%	75%	100%	100%	50%	75%	75%	75%	50%
MemSan	60%	60%	100%	60%	100%	75%	67%	100%	100%	0%
haybale-pitchfork	80%	100%	100%	75%	75%	75%	100%	100%	100%	75%
timecop	75%	100%	100%	75%	75%	50%	25%	75%	75%	50%
Mean	50%	67%	88%	81%	88%	57%	72%	83%	88%	45%

Table 4.2.: Proportion of participants who solved each task per assigned tool (rounded to the nearest percent).

Participants unanimously agreed that discerning the leakage and subsequently mitigating it constituted the principal challenges. These were reflected in 77 grievances. The main subset of these, amounting to 51, expressed that after detecting the leakage, the repair process itself posed difficulties. These difficulties could arise from both details of the tasks and participants' limited familiarity with CT programming. The documentation most consulted by participants was related to CT programming methodologies, suggesting that the primary impediment might be their inexperience in this domain rather than difficulties with the tools themselves. We think this inexperience is not an impediment to use the tools, just in fixing more advanced problems in the code. This observation aligns with our expectations given the demographic we recruited for the study.

Tool outputs and how to interpret them emerged as a recurring concern, in 26 documented instances. Participants grappled with either a lack of comprehensive documentation to interpret the output (15 instances) or ambiguous outputs that did not offer a conclusive determination on the code CTness (11 instances). Here, dudect and haybale-pitchfork stood out for their clarity and precision as seen from little complaints in participant feedback. This likely results from tools concluding their analysis

with a definitive statement about the status of the analyzed code, whereas other tools tend to provide information about possible issues, which can be confusing for beginners. Binsec/Rel and ctverif gathered criticism for occasional vagueness, with 2 and 9 mentions.

The relatively fewer complaints associated with dudect (3 instances) can likely be attributed to its methodology and careful wording of reports.

Even though we knew of pre-existing CT violations, 12 participants reported to be unable to detect any of them. These observations include use of haybale-pitchfork (4 participants) and Binsec/Rel, dudect, and timecop (2 participants). For both MemSan and ctverif it was reported once.

5. Specialized Output Generation. Participants voiced concerns with the verbosity and confusing nature of elaborate error reports. A segment of the study population—3 participants out of 24—grappled with decoding these verbose outputs during the audit tasks. These participants found it challenging to distinguish actual CT violations amid the warnings, and were overwhelmed by the volume of output. Specifically, participant feedback highlighted ctverif as the most problematic in this regard, accounting for 3 complaints. These complaints were directed toward errors preventing its proper usage, and not CT violations. timecop had 2 mentions, while other tools, barring haybale-pitchfork, were criticized once each.

6. Reliability / False Positives. Throughout the repair tasks, participants expressed skepticism regarding the tools, registering 18 complaints centered on perceived reliability issues. Such concerns typically revolved around reports of false positives (recorded 4 times), false negatives (4 times), mistrust in the results (2 instances), or specific tool reasoning limitations (8 times). Among all the tools evaluated, timecop stood out with no reliability complaints. In stark contrast, MemSan found itself at the receiving end of the most criticisms—amounting to 5, predominantly targeting perceived limitations in its analysis. Binsec/Rel follows with the same amount of complaints, but mostly on false positive.

9 participants successfully modified task 10, which was designed to be easily detectable as non-CT yet challenging to fix, in a manner where they discussed their solutions statistically probable CT-ness or correctness compared to the given setting. These successes provide a valuable insight: even when faced with complex tasks, participants can learn and adapt to the nuances of the tools. Due to our provided documentation and the ramp-up of repair task difficulty, we allege that these findings underline the significant role of quality documentation in the tool experience.

Upon transitioning to library audits, participants generally exhibited more restraint in identifying false positives or negatives than in the repair tasks. They mostly expressed low confidence in their analysis, evidenced by 20 self-report on low confidence. *"I have very low confidence in these results that must be false due to my usage of the tool."* () This reticence was particularly pronounced for Binsec/Rel and MemSan, which gathered 5 and 4 reports, respectively. The participants detected few to no

CT violations while analyzing libraries using haybale-pitchfork. This was reported 4 times. With Binsec/Rel, dudect and timecop, only two such cases were reported, and with MemSan and ctverif only one each. Low detection rates are usually correlated with issues during prior steps of the analysis, whether for preparing the library for audit or for using the tool. We conclude this by looking at inter repair task success rates in the first part of the study.

Despite the grievances recorded, most tools, except for ctverif, proved effective in detecting non-CT code during the audit tasks. dudect emerged as the front runner, recording 6 reports of successful detection, followed by Binsec/Rel and timecop, which facilitated 4 reports of discoveries each. Further, haybale-pitchfork accounted for 2 instances, and MemSan contributed 1 finding. We regard these outcomes as practical successes in identifying potential CT-violating bugs in the analyzed open source production code. We did not report these known and (upstream) documented findings.

4.7. Discussion

Building on the results of our study, we discuss the usability of different tools and make a series of recommendations based on the different stages of usage used in our study.

4.7.1. Usability vs verification approaches

Our results provide relevant information on the usability of tools relative to the verification approach they use.

Our study suggests that users found dudect intuitive to use. On the other hand, the underlying approach of statistical time measurement demands a strategic minimization of test parts when dealing with large code bases. Interestingly, the technique that lengthened dudect's processing time might have also contributed to its user-friendliness. The developers of dudect appeared to balance precision with early termination options for less accurate but faster results. Consequently, our participants found the output more intuitive. Whether participants knew they were trading accuracy for speed is unclear. Although it operates as a *"black box"*, a careful balance of precision, speed, and clarity in dudect made it an effective tool for our participants, as seen from their success rates—as seen in Table 4.2—and feedback.

Dynamic instrumentation tools often have a tug-of-war between technical efficiency and user experience, posing challenges during the setup phase. MemSan also faced significant trust issues due to perceived unreliability. timecop stood out with its blend of efficiency and user-friendliness. haybale-pitchfork, proficient yet challenging for some users, hinted at possible issues in the prior analysis steps.

Formal analysis tools, namely Binsec/Rel and ctverif, stand out due to their capacity to offer strong guarantees based on rigorous semantics. While robust, the theoretical foundation of these tools can come at a cost in terms of user experience. Specifically,

ctverif presented a series of usability hurdles, from its initial installation to operational procedures. Many participants encountered challenges despite being provided with a working installation and sample use cases, leading to less successful task resolutions. In addition, our participants did not report particularly more trust in ctverif's output, despite the strong guarantees it claims. On the other hand, Binsec/Rel demonstrated that it is possible to maintain strong analytical guarantees while ensuring a more straightforward setup and operational process. This contrast between the two tools underscores the significance of balancing analytical capabilities with an intuitive user experience when time efficiency and ease of use are highly valued [199].

Our study offers a nuanced perspective on the usability-efficacy spectrum of different analysis tools. While strong guarantees are a primary concern, the trade-offs with usability can sometimes diminish a tool's practical application. The findings emphasize the need for tool developers to prioritize both rigorous analysis capabilities and a seamless user experience, ensuring that state-of-the-art tools are not just theoretically sound but also practically adoptable.

4.7.2. Recommendations

We combine the data from our empirical study with expert insights to curate a suite of recommendations. Our observations indicate that the tool usage is divided into multiple stages. Of particular concern, inhibiting complexities at early stages can deter users from progressing.

Installation. Many tools have a large number of dependencies and require custom building paths. As a result, installing these tools may be highly challenging in the mid-term, even if all the tool's dependencies are maintained. From study setup and piloting we extract the following recommendations:

- Reduce and avoid less maintained dependencies.
- Make tools available via package managers.

A more general recommendation would be to embrace the best practices of open-source software development, which has a long, integrated maintenance period and is often available as native packages in software distributions—native packages through distributions also make for discoverable tools.

Familiarization. After installation, users may run the tool on common examples, in this case crypto libraries, just to make sure that the tool is indeed running, and without caring for the tool's results. However, there are many obstacles to such dry runs. This includes, for instance, the need to compile libraries using specific compilation flags, different from the flags used to produce code, or the need to rewrite libraries to overcome limitations in the coverage of the tool. To avoid such situations and to ensure that tools provide adequate support for beginners, we make the following recommendations:

- Provide support for processing inline assembly and vectorized cryptographic code.
- Provide support for processing precompiled code, statically or dynamically linked.
- Provide user-friendly examples amenable to adaptations.

- Design intuitive tutorials catering to novices, and covering all the aspects of tool-usage.
- Prioritize a comprehensive documentation structure, accentuating essentials before delving into details. Make sure that the documentation avoids overly specialized jargon.

The latter recommendations are based on the feedback from the study participants.

Secret Designation. Most constant-time tools require users to provide security annotations. The annotations are typically given in the code or through some external wrapper. Moreover, many cryptographic libraries require users to declassify computations, for example to make ciphertexts public. From prior literature on different tool designs and their problem areas, we extract the following recommendations:

- Make annotations simple and external in additional files.
- Provide mechanisms to declare internal secrets [152].
- Provide mechanisms to allow to *declassify* computations.

Output generation. Results of analysis tools must be semantically rich, easy to navigate, and exploitable in a broader setting. Based on our interpretation of the results of the study, and our expertise, we make the following recommendations:

- Provide output that is readily understandable by users, including origin of leakage.
- Offer the possibility to report all leakage violations at once. Deduplicate findings in order to avoid repeating violations.
- Offer export formats for integration with bug-tracking tools.
- Have a delta mode for CI.

Analysis Runtime. For integration into users' workflows or CI, analysis should be possible in reasonable time—we think a few minutes are fine even for interactive use, but hours or longer are not. From Jancar et al. [199] as well as our own study participants' feedback and the CPU utilization of our study setup, we make the following recommendations:

- Use progress indicators (progress bar or logs) to ensure the user understands the tool is not stalled.
- If applicable, leverage multiple CPU cores for large tasks.

4.8. Conclusion

We collected data from 24 participants using 6 CT analysis tools to analyze small tasks and audit 3 open-source cryptographic libraries that are documented not to be fully CT. Our broad conclusion is that CT analysis tools have usability shortcomings that prevent them from being integrated into developers' workflows. Although our analysis focused on CT tools, we believe that many of our findings also apply to tools for analyzing microarchitectural side-channels. We believe the community should address these shortcomings by focusing on a handful of easy-to-use and maintained tools that go beyond the CT leakage model and cover a broad range of leakage models.

In this chapter, we follow up on the study in Chapter 3 to give more detailed grievances on different classes of tools and their general behaviour, as well as exemplary differences in project maintenance and different development focus between the specific tools under study. Our test setup is publicly available, so other researchers and other practitioners can use the scripts to simplify getting the tools in working order, but also replicate our test setup for their own tools of interest. We give detailed instructions and documentation for the tasks and code-analysis of open-source cryptographic libraries with known non-constant-time code paths, so success in evaluating the tools is as easy as possible. Still, we found grievances across the tools from usability problems, to performance problems, and even structural inability to find all possible constant-time vulnerabilities. We report no new vulnerabilities, but give detailed recommendations to tool developers on the workflow and possible grievances during each step, combined with our expert opinion advice from cryptographic engineering and code-analysis as a framework on how much work each grievance would approximately take to fix when compared with the others.

5. Reproducible Builds for Software Supply Chain Security

Disclaimer

The contents of this chapter were previously peer-reviewed and published as part of the conference paper titled "It's like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security", presented at the 2023 IEEE Symposium on Security and Privacy. This research was conducted as a team with my co-authors Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar; this chapter therefore uses the academic "we". I developed the main idea with Yasemin Acar, with input from William Enck and Dominik Wermke. The instrument was developed by Yasemin Acar and myself. I conducted the interviews, with the help of Dominik Wermke. I lead the initial round of coding, independently verified by Dominik Wermke, followed by joint affinity diagramming with Yasemin Acar, Dominik Wermke, and Sascha Fahl. I lead writing the paper for publication, with support from all other authors.

5.1. Motivation

Cryptographic signatures and trust between developers secured by high-quality implementations of tools integrating cryptographic signatures in their workflows are necessary, but not sufficient. When we look at the software supply chain, we see many small projects being integrated into larger software, each with a diverse project organization, people working on it, and security practices [391]. Each of these projects generates some form of artifacts that may be secured with cryptographic signatures against malicious third party changes, but one question remains: What if the artifact changes when it is regenerated by the project itself?

In theory, any backdoor attack can change things in the project and nobody would be the wiser if any regeneration of artifacts changes the result anyway. This is why reproducible builds are needed. We link to the first attack that necessitates thinking about supply chain security in 1983, a scientifically published defense against it, and show how they both tie together to form necessary requirements for a good default in building artifacts so the software supply chain *can* be defended in the current setting, where not just one company can dictate what needs to be trusted, but multiple parties can check for themselves if they want to trust a project and its artifacts.

5.2. Introduction

"To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software." (Ken Thompson)

Thompson's 1984 Turing award lecture "Trusting Trust" demonstrated that the security of a program is more than the logic in its source code [354]. It also includes all of the programs used to make the source code logic executable. Thirty-six years later, Thompson's theoretical attack became a pressing concern for nation states across the globe. The 2020 Solarwinds attack did not inject malicious source code into a project. It did not trick users into installing software with an invalid signature. The Solarwinds attack trojaned the *build system*, transparently injecting malicious logic into the product binary signed with Solarwinds' official code signing keys.

The world of software has transformed entirely since Thompson's lecture. His suggestion of "trusting the people" is orders of magnitude harder today than in 1984. Today's software supply chain ecosystem is vast, with software projects often including transitive dependencies that are tens to hundreds of layers deep. And despite much of the software supply chain being open source, end users rarely, if ever, compile software themselves. Thus, the build system has become an ideal target for attackers.

Reproducible Builds (often abbreviated R-Bs) offer a foundation for defending against attacks targeting the build system. "*A build is **reproducible** if given the same source code, build environment and build instructions, any party can recreate bit-by-bit identical copies of all specified artifacts*" [309]. Conceptually, R-Bs allow multiple parties to build the same software package, and assuming the attacker cannot simultaneously compromise the build systems of all parties, arbitrary subversion of an individual build system can be trivially detected. High-profile projects including Bitcoin [117] and Tor [288] use R-Bs to assure users that distributed binary executables match their source code. Recent literature has built upon R-Bs to enhance build provenance guarantees [358] and create verifiable builds [273] more suitable for providing guarantees to end customers. Finally, the US National Security Agency (NSA) identifies R-Bs as an important part of securing the software supply chain [133, 276].

Unfortunately, most software builds are not reproducible. Over the past decade, the Reproducible-Builds.org project has cataloged the many sources of non-determinism that prevent R-Bs, including uncontrolled build inputs (e.g., system time, environment variables, and build location) and build non-determinism (e.g., process scheduling) [227]. Despite industry [119, 311] and academic [304, 305, 306] tools to facilitate R-Bs and extremely high rates of R-Bs in some popular Linux distributions (95.5% for Debian on AMD64 [310] and 88.7% for Arch Linux core [308]), much of the software industry believes R-Bs to be a long-term goal, if achievable at all [76].

The goal of this paper is to help R-Bs more quickly become a commonplace effort in software development. We seek answers to the following research questions:

RQ1: *"What are motivations for, and common themes around, adopting reproducible builds in projects?"* We are interested in our participants' motivations around striving for reproducible builds in their projects, specifically in the case of complex, community-

or industry-driven projects, that likely necessitate a complex, interconnected system of motivations and drivers. We are also interested if some of the motivations involve security, and what specific threat models are applied.
RQ2: *"What experiences and challenges did projects encounter in the context of reproducible builds?"* Most projects were not created with reproducibility in mind. We are interested in what experiences were made, and challenges encountered, on the way towards reproducibility, both by contributors of the project as well as with outside entities such as customers or upstream dependencies. This research question aims at the personal experiences of R-B developers.
RQ3: *"What are commonly encountered obstacles and facilitators in projects' efforts towards reproducibility?"* Some projects stall in their efforts toward reproducibility, while others succeed. We are interested in what facilitators and obstacles our participants encountered during their efforts, how they approached them, and what they would recommend for other projects aiming to become reproducible. This research question aims at external factors encountered by R-Bs developers.

By answering these questions, we hope to guide future industry and academic efforts that target both the technical and human aspects of R-Bs. In this paper, we report the results of a semi-structured interview study with 24 prominent and public members of the R-Bs effort. All participants were experienced developers (5+ years) with R-Bs experience, who could give deep insights into their thought and development processes, and deeply discuss and reflect on the topic. Based on these interviews, we offer the following key insights.

- *Open Source developers are self-motivated to work on software infrastructure.* They see themselves as users as well as developers and want to build better software even without external requests.

- *The Snowden revelations and SolarWinds incident heightened the security awareness.* While some people were interested in R-Bs before, their number grew significantly after those two public events.

- *Caching matters most to businesses.* R-Bs allow for efficient caching of artifacts, which was mentioned as the most important aspect for businesses.

While we specifically choose to interview experts with many invested years in R-Bs for their experience and insights, we also want to highlight that our expert participants are likely positively biased regarding the potential for R-Bs becoming widespread. While this may be substantiated by growing numbers of developers on the R-Bs mailing list as well as growing parts of operating systems being tested as built reproducibly, our sample is still biased towards R-B enthusiasts [231].

The remainder of this paper proceeds as follows. Section 5.3 discusses background information and related work. Section 5.4 describes our interview methodology. Section 5.5 presents our detailed results. Section 5.6 provides discussion and a set of recommendations. Section 5.7 concludes.

5.3. Background and Related Work

This section provides background information for reproducible builds and their relation to overall open source software supply chain security, as well as related work in three key areas: research on reproducible builds, open-source software security, and interview studies with software developers with a focus on software security.

Relationships between open source software projects can be characterized by their order of incorporating software: When one project uses another project's software, it is often called downstream of the project originating the software. Conversely, many project interactions are with upstream projects, which originated the software. A single package that does not have a downstream relationship with another package, instead of just redistributing it without changes, is called a leaf package. Any package that provides some functionality used in the infrastructure of a software build and supply tool chain can also be called an infrastructure package. To make a package build reproducibly, all of its dependencies need to build reproducibly. If a maintainer changes one package to do so, it is often most efficient to upstream those changes as a patch set for incorporation by the upstream developers, so the work is shared with all other downstream projects of that dependency.

5.3.1. Reproducible Builds Background Information

Reproducible builds are a collection of techniques and processes that aim to make the compilation of source to resulting binary code deterministic: the same source code should always be compiled to the bit-by-bit identical binary code. Achieving reproducibility in building software is non-trivial as the software compilation process is susceptible to multiple sources of non-determinism.

Lamb and Zacchiroli [228] provide an overview of common sources of non-determinism during the build process:[1]

Build timestamps are commonly used by C programming language projects and other build tools such as make [355] using similar macros to the __DATE__ C-preprocessor macro. While build timestamps are useful for bug reporting, most version control systems including Git provides alternatives such as offer better solutions to record specific software versions without introducing additional non-determinism [86].

Build paths are commonly embedded using a C programming language preprocessor macro (e.g., __FILE__) as well as assertion macros referencing source code locations or log messages. In most cases, a relative path to the root location of the source code is sufficient [398] and reduces the amount of non-determinism. Build directory name inclusion can be prevented by only including build paths

[1] This list is not meant to be exhaustive.

relative to the root directory of the source code, which will be constant. Non-constant paths can include user-specific changes. To avoid these while maintaining absolute build paths, a statically known directory path can be used for all software builds. This is used in Nix, GUIX, but also Debian using the sbuild software [322], which is the basis for the R-Bs checking tool reprotest [311].

Filesystem ordering as part of the POSIX standard does not define the order when returning the results of a directory listing causing additional non-determinism.

Archive metadata such as the date and ownership information in .zip and .tar archives are commonly used for archival purposes but should be avoided to reduce non-determinism.

Randomness results not only from parallelism and concurrency in the build process. Additionally, some compilers introduce explicit randomness during the build process to generate unique names that do not conflict with those generated for other files such as single compilation unit [325] distinct identifiers.

Unitialized memory adds non-determinism by not always initializing memory to a programmer-defined value and is supported by some popular low-level programming languages including C and C++.

As part of our interview study, we aim to better understand how open source software developers identify and handle these sources of non-determinism.

The Reproducible-Builds.org project [309] aims to support developers and software projects in making their build processes reproducible. They support mainstream Linux distributions such as Debian (currently 95.5% reproducible for AMD64) and Arch Linux (currently 88.7% reproducible for Arch Linux core) in building their software reproducibly.

The project provides several tools to help developers achieve R-Bs, including reprotest to automate the process of building a package multiple times in diverse environments and diffoscope [119] to help find the differences within complex binary packages and directories. Diffoscope receives two files to be compared as input, tries to unpack any data recursively, and displays differences found between the two input files. It is plugin-based and as a wrapper re-uses common file format handling utilities.[2]

As part of our work, we aim to learn if and how developers use support tools for R-Bs.

5.3.2. Research on Reproducible Builds

Most prior research considering reproducible builds addressed technical challenges. In 2005, Wheeler proposed diverse double compilation [394] to address Thompson's

[2] All provided tools can be found at https://reproducible-builds.org/tools/.

Trusting Trust attack [393, 354], connecting software security and R-Bs. Diverse double compilation suggests that source code is compiled by different compilers, potentially mutually distrusting parties, and every additional instance giving the same resulting binary file makes the Trusting Trust attack less likely. However, the software must build reproducibly, giving the exact same binary if compiled at different points in time, space, and for any number of recompilations. To generate a fully trustworthy compile chain, trustworthy root binaries are needed. A popular approach to address this issue is bootstrappable builds [298]: The core idea is to address circular build dependencies of complex software by creating a new dependency path using more simple prerequisite software.

Prior work suggested R-Bs as a primitive to improve software supply chain security including distributed verification [273], transparency logs [238], supply chain workflow integrations [358], and "keyless" signatures using trusted third-party inspectable logs [268].

Prior work has also considered better tool support for R-Bs. Several Linux distributions such as NixOS [123] and GNU GUIX [103, 104, 102] and their corresponding packaging tools are based upon primitives that promote reproducibility. In a series of work, Ren et al. proposed RepLoc [304], RepTrace [305] and RepFix [306] to more precisely locate sources of non-determinism and suggest patches. Recently, containers have been explored specifically for reproducibility [263] and corresponding security impact [409], as well as new build tools [197] which have been reimplemented as open-source tools. In closed- or mixed-source environments, rebuilds can be organized in a more centralized manner [296] that also supports R-Bs.

Closest to our work, Butler et al. interviewed company experts to investigate the value of R-Bs for companies [76]. They identified reasons for their limited adoption of R-Bs like limited awareness and perceived challenges.

In contrast to previous research on reproducible builds, our interview study aims to shed light on enablers and blockers for the adoption of R-Bs in the open source community.

5.3.3. Research on Open Source Software Security

Open source repositories are open to access from outsiders and the security and privacy research community has established use of this data source. From these repositories' commits [293, 292, 13] and contributors [315], but also secondary and related information like vulnerabilities [154, 332] and even torrents [160, 161] as an alternate access method have been used for research in a number of papers.

Big open-source projects like FreeBSD [120], Linux [362], and Mozilla [257] have been the focus of case studies. Topical analysis of vulnerabilities can take the corpus of code and has been done by matching Common Vulnerabilities and Exposuress (CVEs) numbers [130, 190], by using the code base for evaluation of static analysis tools [22, 15, 407, 406], or vulnerability changes [60, 25, 351, 379]. These changes are necessary to secure a codebase, so the patterns and development of fixes have been investigated [340, 234, 302]. In one specific work, 337 CVE entries were linked to the

patches fixing them and the authors found that the developers of those fixes are of higher experience levels [291]. The highly polished Linux Kernel implementations of drivers have also been analyzed multiple times [115, 33].

The social aspects of repository contents researched things like toxic comments [253] and metadata [342, 236] as well as programming languages [237] and their general maintenance [177, 97]. Furthermore, pull requests [159, 359, 140], collaboration [110, 373, 100], and even gamification of the process [260] have been studied in related work.

In contrast to closed-source development, the security challenges for open-source communities are unique, valuable sources of research data and therefore well researched [323, 108, 389]. Open source software repositories contain public commits and issues that can be statistically evaluated [178], mined for emotions [295] and security tactics [321]. Programming language-specific communities were investigated, finding the Python and JavaScript communities to not react quickly to security vulnerabilities [27]. A large-scale analysis of hundreds of thousands code review requests from different open-source projects identified [58] the changes of less experienced contributors to be between 1.8 and 24 times more likely to contain security vulnerabilities.

Automated identification of open-source projects using vulnerability data [285, 3, 411] and toxic comments [301] were investigated. Both problems may weaken trust in the public reception of these software projects, even among collaborators. There has been work on different factors to influence [28, 339, 359] trust and quantification [74, 349] of it. Trust is influenced by open source projects' security, which itself is highly influenced by code quality, which has been investigated by different assessment models [164], the difference between architectural plans and implementation [319], and later code reviews [59, 353]. The base unit of collaboration is the committer, whose motivations [174, 173, 294, 255], barriers to entry [343, 388], and the eventual pull requests [347] have been a focus of different works.

The onboarding [344, 345, 124, 35] and mentoring [79, 35] of new committers have been studied as well. Socialization in the form of pre-existing relationships is an important factor and precursor [81] to joining GitHub projects. Most of such project ecosystems have one central project connecting every part of that ecosystem via software dependencies and connecting to other ecosystems as well [54].

In contrast to previous work on open-source software security, we focus on understanding enablers and blockers for the adoption of reproducible builds as a critical contribution to overall software security.

5.3.4. Interviews with Security Developers

The security and privacy research community uses interviews effectively and as a well-established method for detailed investigations. Prior work utilized interviews to gain detailed information about different kinds of experts, for example: Administrators [44, 40], and overall security professionals [61, 337], but also of communities

that rely on their security, from journalists [247], over editors [248], to victim service providers [90]. Interviews can be used as a part of larger studies and give researchers a view of data that is not readily available from technical systems: e.g., thoughts and procedures. For Tor adoption [148] or how to work with encryption [34], examples are readily available, just like developers' thoughts and plans about security features [170].

In 2017, IT administrators were asked about the usability of deploying HTTPS [216], and programmers about the benefits and drawbacks of outright changing to a different programming language in a study in 2022 [146]. A 2021 qualitative study was performed about developers' struggles with CSP [318], and in 2022 industry practitioners' mental models of adversarial machine learning were investigated [52, 256].

Open-source developers are a special case, their transparency and distributed work being subject of dedicated study [110]. The social barrier of entry for new contributors was studied as part of a larger study containing semi-structured interviews with 36 developers recruited from 14 projects [343]. Their challenges and strategies for overcoming them using tasks recommended for newcomers were studied by interviewing mentors of 10 open-source projects [35]. Recently, Wermke et al. leveraged interviews to investigate behind-the-scene security processes in open source projects [392] and industry projects that utilize open source components [391].

As an extension to previous interview studies with security developers, we focus on reproducible builds.

5.4. Methodology

We conducted 24 in-depth, semi-structured interviews with developers, maintainers, and contributors implementing reproducible builds for their software in the fall of 2022. During these interviews, we discussed reasons for adopting R-Bs and the processes they encountered and used while doing this. Both the interview guide and the codebook are provided in the Chapter C.

5.4.1. Participant Recruitment

We recruited participants by emailing 100 members of the Reproducible Builds Project website's mailing list "rb-general" [299]. We decided on this more focused recruitment approach, because based on our prior experiences we assumed that developers from outside the mailing list would often not be familiar with the concept of R-Bs. Some participants made additional suggestions and/or offered introductions to potential interviewees with insights into reproducible builds; we followed up on all suggestions.

We interviewed a total of 24 participants between August and November 2022. Table 5.1 provides an interviewee demographics summary. All of our participants are software developers (5+ years) with reproducible builds experience. They provided us with insights into their thoughts on R-Bs and development processes, and could

discuss and reflect on the topic. Five participants stated that they are engaging less with R-Bs than at some point in the past, never fully joined the R-B effort, or have not yet started actively working towards R-Bs, while still being interested, to hear their reasons against R-Bs. The set of developers we interviewed contained two non-binary individuals and otherwise men, which is disappointing for diversity, but reflects those active in the mailing list and community we targeted.

5.4.2. Interview Procedure

We conducted semi-structured interviews and topical hints to keep the interviews flowing and otherwise let our participants structure their answers and remarks on their own following established practices [231]. We built our interview guide around our research questions, and discussed and revised it with researchers outside our team with reproducible builds experience. Figure 5.1 illustrates the interview structure and we provide our final interview in the Chapter C.

To pre-test the interview guide, we conducted one mock interview. Interviews generally lasted between 30 to 60 minutes. We scheduled the first batch of interviews with the intention of treating them as pilot-interviews, however, as no major changes were made to the interview guide and the data we collected from them was meaningful, we decided to include theses pilot interviews in our data set. Throughout the interviews, we asked for experiences and opinions and participants' responses indicate that they also reported their (strong) opinions in either direction of R-Bs, including reasons against R-Bs. We specifically iterated over the (various) definition(s) for reproducibility and discussed them with our interviewees.

We offered our interviewees to choose between a locally hosted Jitsi and Zoom for the remote interviews and conducted six of the interviews in person at the Reproducible Builds Summit 2022.[3] We gave them the option to end the interview and withdraw at any time. We started interviews with verbal consent to being audio-recorded, the interview being transcribed by a GDPR-compliant third-party service and the use of the interviews in a scientific publication. One or two authors conducted and recorded the interviews.

1. Context and Definition. The interview guide opened with a section establishing the context for participants, their projects, and their role in the projects. Questions included how, when, and why they got in touch with their projects, their background, and how they decided to focus on reproducible builds. In addition, we established their definition of a reproducible build for their project and in general.

2. Reasons and Decisions. The *"Reasons and Decisions"* section explored the reasons for progressing towards reproducible builds in the project and specific decisions surrounding this process. Questions included internal and external drivers for pursuing

[3] https://reproducible-builds.org/events/venice2022/

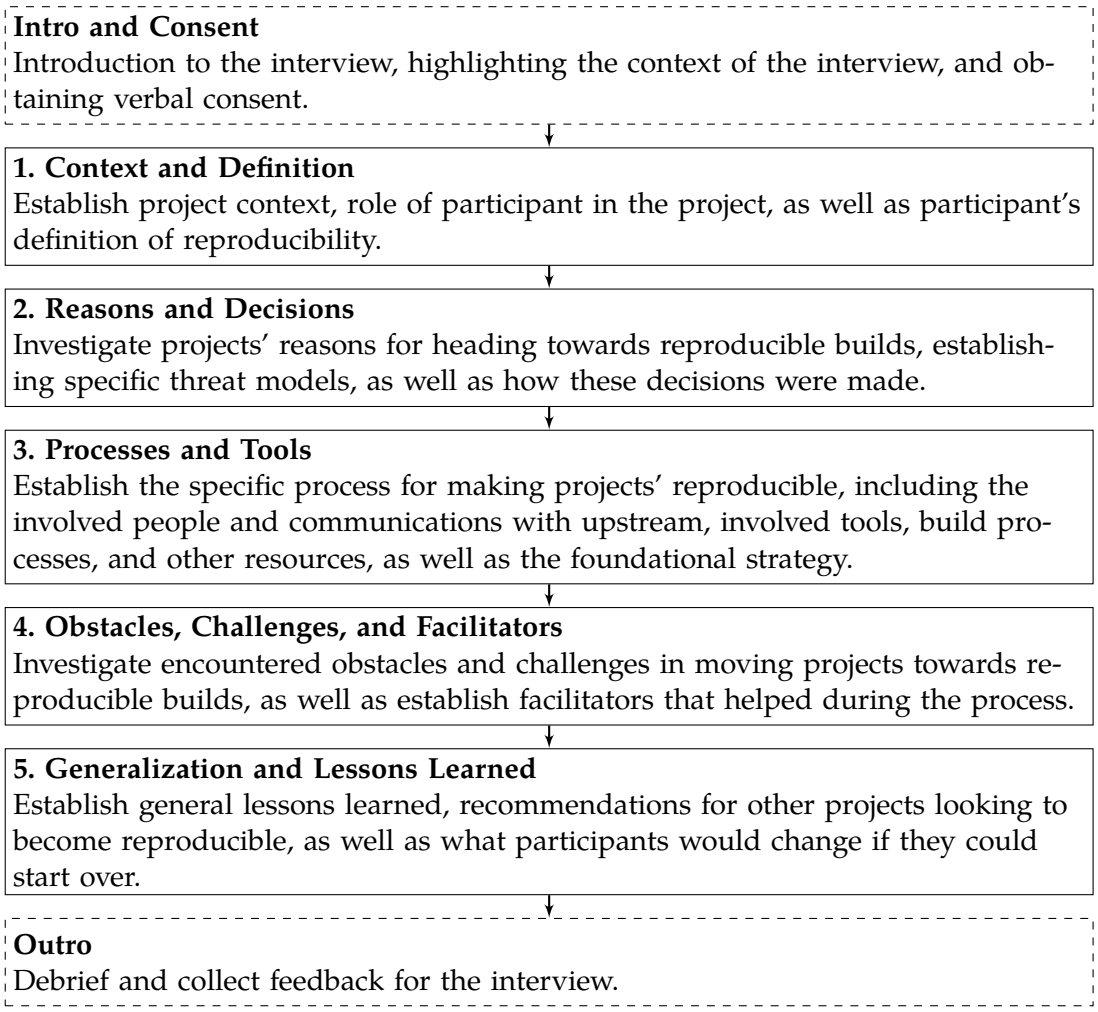

Figure 5.1.: Illustration of topic flow in the reproducible builds interviews. As we conducted semi-structured interviews, participants were presented with general questions and corresponding follow-ups in each section, but were generally free to diverge from this flow.

reproducible builds, threat models and requirements, and how decisions were formed and by whom.

3. Processes and Tools. The *"Processes and Tools"* section established utilized processes and tools for the reproducible builds, as well as experiences with these approaches. Questions included the general process of making the project or parts of it reproducible, the estimated time for these efforts, how the experiences with upstream projects were, and whether additional resources like documentation or websites were any help.

4. Obstacles, Challenges, and Facilitators. The *"Obstacles, Challenges, and Facilitators"* section investigated encountered obstacles, challenges, and positive facilitators during the projects' progress towards reproducible builds. Questions included obstacles of different types (organizational, technical, dependencies / upstream, and communities) and which particular factors were helpful for the project.

5. Generalization and Lessons Learned. The interview closed with a *"Generalization and Lessons Learned"* section, establishing general themes and lessons learned from their reproducible builds attempts. Questions included what they would do differently if they could start over, what worked well and what did not, and what they would recommend for other developers and projects attempting reproducible builds.

After the interview, participants often expressed their interest in our research and in being mailed results (which we promised) and/or recommended other potential interviewees. The authors debriefed with each other after the interviews, discussing new insights.

5.4.3. Reproducible Builds Summit Discussion

We presented preliminary results from this study, mostly on what motivates reproducible builds and how they can benefit stakeholders, to attendees at the Reproducible Builds Summit 2022, Venice. Attendees, together with the main author, then used a collaborative session to iterate over a matrix of motivations and benefits for reproducible builds, which we (with the attendees' consent) discuss with insights into motivations and experiences from our interviews in Section 5.6.

5.4.4. Coding and Analysis

After the 18th interview, new themes ceased to emerge and participants mostly iterated themes we had heard before; therefore, we chose to stop interviewing after 24 participants, reaching theoretical saturation [147]. We analyzed our data using qualitative coding [231]. The main author developed a codebook based on the interview guide and insights from conducting the interviews; the team reviewed and slightly iterated over the codebook. The main author then coded all data with the codebook, discussing insights with the team. The final codebook is provided in the Chapter C.

5.4.5. Ethical Considerations and Data Protection

Our study, including recruitment strategy, data collection, recording methods, and interview guide, was approved by our Institutional Review Board (IRB).

All participants were either recommended by their own colleagues or signed up to a public list of those generally interested in reproducible builds and reacted accordingly to our invitation emails: They generally responded positively, expressed

interest in our research, recommended others, and attempted to schedule interviews. We offered online (and offline) meeting and recording options, including meeting them at the Reproducible Builds Summit that an author attended, as well as using self-hosted meeting software (Jitsi) with local recordings for improved privacy. All participants consented to the interviews, recordings, and transcription by an external, GDPR-compliant service. When participants flagged parts of the interview as too sensitive to transcribe, we removed them before transcription. We de-identified participants and interview transcripts using identifiers such as P01 and de-identified sensitive information in the transcripts. After checking transcripts for correctness, we deleted all audio recordings. We offered no compensation, due to our interviewees being potentially highly paid individuals, motivated to work on R-Bs by their involvement in the Open Source/Free Software project communities; based on our prior experience with this population, they usually reject payments or attempt to re-direct them to donations for OSS projects, which our funding source was unable to accomplish.

5.4.6. Limitations

Our work is affected by limitations common to interview studies, including limited generalizability and biases such as recall and social desirability biases [231]. We accounted for these biases by interviewing a diverse (in projects and experience) sample of those who fit our recruitment criteria. Furthermore, we only interviewed those involved in the Reproducible Builds project; therefore, our sample represented the experiences and perceptions of those who were generally highly aware of and/or working with reproducible builds. While interviewees happened to be concentrated in the Western world and were predominantly male, this is in line with the reproducible builds community [277]. Our sample includes various organizational contexts, from industry-leading companies to single developers. Based on provided answers, we can reason that our sample is broad and diverse in R-B adoption, experiences, and organizational contexts, but we refrain from comparing smaller and larger companies quantitatively due the limited sample sizes and the more qualitative nature or our research approach.

5.5. Results

In this section, we describe the findings from 24 semi-structured interviews with developers with experience in R-Bs for software projects. First, we illustrate our interviewees' motivation to implement R-Bs for their software projects. Second, we explore supporting factors and obstacles for R-Bs. We de-identified participant quotes, made minor grammatical corrections, and highlighted omissions using brackets ("[...]"). German interview quotes were translated into English by native German speakers.

Counts in our reporting should be interpreted as the number of interviewees that touched on the specific topic at least once during their interview. As qualitative interview study, reported counts are not necessarily representative for the wider developer

Table 5.1.: Overview of our interview participants

Alias	Interview			Project		
	Duration	Codes[1]	Recruitment Channel	Position	Area	Software Stack[2]
P01	44 minutes	40	rb-general mailing list	Developer	Operating systems	Ocaml
P02	31 minutes	15	rb-general mailing list	Developer	Desktop Environments	C
P03	42 minutes	29	rb-general mailing list	Developer	Operating systems	C/C++
P04	39 minutes	27	rb-general mailing list	Developer	Operating systems	Assembly, C, and others
P05	39 minutes	31	rb-general mailing list	Developer	Operating systems	C, Tcl
P06	51 minutes	40	rb-general mailing list	Developer	Operating systems	diverse
P07	45 minutes	29	rb-general mailing list	Developer	Operating systems	C, Python
P08	58 minutes	14	rb-general mailing list	Developer	Graphics processing	C/C++
P09	50 minutes	28	rb-general mailing list	Developer	Operating systems	Python, a little C/C++
P10	55 minutes	36	rb-general mailing list	Developer	Operating systems	C
P11	57 minutes	24	rb-general mailing list	Developer	Privacy preservation	C/C++, Rust
P12	54 minutes	20	rb-general mailing list	Developer	Build systems, GUIs	Python, C and others
P13	64 minutes	19	Personal recommendation	Developer	Operating systems	C and others
P14	50 minutes	15	rb-general mailing list	Project lead	Electronic currencies	diverse
P15	34 minutes	22	rb-general mailing list	Advisor	Privacy infrastructures	-
P16	42 minutes	26	Personal recommendation	CEO	Build systems	Python and others
P17	39 minutes	20	rb-general mailing list	Developer	Operating systems	C/C++, Python, Scheme, and others
PS18[3]	45 minutes	24	RB Summit 2022, Venice	Developer	Embedded software	C, Assembly, and others
P19	31 minutes	11	RB Summit 2022, Venice	Developer	Privacy preservation	C/C++, Rust, and others
PS20[3]	48 minutes	27	RB Summit 2022, Venice	Developer	Operating systems	Scheme, C, and others
P21	49 minutes	34	RB Summit 2022, Venice	Developer	Operating systems	diverse
PS22[3]	46 minutes	24	RB Summit 2022, Venice	Developer	Operating systems	diverse
PS23[3]	58 minutes	29	RB Summit 2022, Venice	Developer	Build systems	Java
P24	31 minutes	35	rb-general mailing list	Developer	Operating systems	C/C++ and others

[1] Total number of codes assigned to the interview after resolving conflicts.
[2] Abbreviated. Common among all participants was some amount of shell script use.
[3] Participant aliases: P means participant was recruited by email, PS indicates recruitment at Reproducible Builds Summit.

population, but are included to give some general idea about the distribution of codes and to highlight especially prevalent or underrepresented themes in the interviews.

Table 5.1 provides an overview of project demographics. We conducted 18 remote interviews and six in-person interviews at the Reproducible Builds Summit 2022. Four of those interviewees were recruited at the summit. We mark those interviews "PS" (vs. "P" for all other participants), as their attitude towards R-Bs might be particularly positive. Most (21) interviewees worked in some capacity as developers on projects that strived to build reproducibly. All interviewees were software developers with between 5 and more than 20 years of experience in general software development, specifically between 2 and 12 years on R-Bs.

5.5.1. Why and How Projects Started to Work on Reproducible Builds

In this section, we illustrate reasons for and against reproducible builds that the interviewees mentioned.

Reasons for and against adopting Reproducible Builds. We identified technical and non-technical themes related to our participants' motivations for making their

builds reproducible, both related to security and as an intuition of how compilation should behave according to their own mental model of software compilation.

Ten participants reported encountering a misunderstanding of the mechanics of current software compilers: They brought up that they commonly encounter the expectation that compilers generally produce the same binary output given the same source code without outside interference.

> *"I have an input and some computation, so I expect the output to be the same.* [*Like a mathematical function.*] *And like a scientific function. It's computations; you put something in it, and the same output should get out. Except if the function randomizes,* [...] *or it is broken. I think unreproducible builds are illogical. Conversely, reproducible builds are logical."* (P21)

Their main motivation to work on reproducible builds was to bring the mechanics of software compilation closer to their assumptions. Aside from compiler mechanics, broken expectations were brought up by eleven interviewees. Beside their expectation for compilation working deterministically, they also want software to work the same in the future. Two participants reported beneficial (better run-time performance in some cases or security fixes) but still unexpected compiler behavior:

> *"It's more like things aren't fixed. You do a deployment one day with some source code, and you come back a week later, and you do the deployment again. You repeat the process with the same source code. Your source code is the same, but because you haven't engineered the process to be reproducible, what you actually deploy is something different.* [...] [*T*]*oo often, one, it's not understood what's changing and two, you don't have control over it."* (PS22)

Some interviewees (6) mentioned the importance of constantly maintaining a high level of build quality to limit increased effort later in the development process. For 18 interviewees, improved software quality was the main motivator. One participant compared the effects of improving software quality by making it build reproducibly to dental flossing:

> *"At every summit you have people show up there because they want the hands-on support to get their thing into a more reproducible status so* [...] *I have this ongoing analogy I use around dental flossing and the dentists tell you how important it is to the dental floss and people do not very often floss as much as their dentist tells them to. I think Reproducibility is falling in that same spectrum of really important things* [*and*] *people treat it like that."* (P15)

A similar sentiment was reported for company resources:

> *"The only reason why we ever moved into this direction of reproducible builds, for the company, was because it was causing us issues in losing time. People would forget to declare dependencies and that would fail the build. People merge small changes that change the dependency ordering when executed massively parallel. We noticed that when this happened, it would take us half a day to fix. During this half of the day, there were about 500 people who couldn't build anymore. That costs a lot of money and time."* (P05)

As described earlier, many participants started working on R-Bs due to intrinsic motivation; they began with the parts of their software projects that best fit their motivation. Once they had worked on their chosen package, they explored more complex components required for R-Bs in their software packaging work.

The main motivation for 16 interviewees was working on infrastructure reproducibility, while six worked on single software packages. While building infrastructure solves more problems compared to leaf packages, individual configuration specific R-Bs problems are specific to leaf packages.[4] Overall, participants reported complex, interconnected motivations to implement R-Bs; some motivations are related to intuition for how builds should behave, and some are grounded in explicit security concerns, both for developers and software and its users. One developer mentioned working on *version pinning*. Version pinning is an approach of describing the exact version of each software dependency, as an easy, but necessary area to implement R-Bs by documenting a set of dependency versions that produced a working artifact. Another two interviewees mentioned that they worked on difficulties with specific compiler versions, citing specific versions of well-known compilers as problematic for R-Bs. Older versions of those compilers generated more reproducible binaries than current versions. Finally, two participants mentioned to have worked on *transitive dependencies*, i.e., dependencies of dependencies, which may influence reproducibility. These participants reported having investigated if indirect dependencies broke their reproducibility and figured out how to fix this.

Four participants mentioned interactions with upstream projects. While they had made a version of their software that depended on upstream projects being reproducible, they needed the upstream project to incorporate and maintain their changes. One interviewee reported that they had an upstream project rewrite a suggested patch from scratch, and were amazed at the commitment to R-Bs by the upstream project. The ideal goal for all participants was full bit-by-bit reproducibility. However, projects have considered weaker reproducibility criteria as more realistically achievable along the way.

Two participants started with the build process by manually debugging unreproducibility introduced by it. After achieving reproducibility manually, 15 participants continued their efforts toward R-Bs through more automation in Continuous Integration (CI) and other infrastructure.

With R-Bs, a build of the same source code version results in the exact same binary. Without R-Bs, a rebuild can change the binary of the package, which can still be cached each time but leads to waste and incompatibilities. Ten developers, across industry and open source projects, mentioned that slow build speeds are frustrating for developers, and can be caused by inefficient caching that is sensitive to small changes. Building reproducibly would solve this problem.

> *"We recently got a new build machine which is I think 64 cores and [lots] of memory or something like that, terabytes of this as a benchmark. How long does it take to build*

[4] See https://github.com/bmwiedemann/theunreproduciblepackage.git for a list of known problems that can occur in a leaf package.

> *everything from scratch? It took about a day on that machine to build everything.* [...] *Fortunately, that's not the typical dev experience because our particular system caches build outputs.*" (P11)

The caching strategy mentioned by P11 can apply if build systems cache compilation artifacts during build-time, instead of complete artifacts. In the case of build-time artifacts to be included into full compilation artifacts, they themselves can be built in a repeatable way that affords caching, without the whole build process necessarily being fully reproducible. If all of those artifacts afford this building in a repeatable way that gives bit-by-bit identical artifacts and those artifacts are combined in a way that does not break reproducible builds, then the build system produces reproducible builds. This is different from the notion that, if the compilation toolchain works like a mathematical function, giving the exact same result to an unchanged input, the compilation builds reproducibly, which is a harder precondition not necessary for build reproducibility.

Reasons against working on Reproducible Builds. Thirteen interviewees also mentioned reasons against R-Bs. They reported that, due to decreasing enthusiasm and repetitive work, they decreased the amount of time and effort they invested in R-Bs. This decrease in enthusiasm and effort corresponded with a perceived impracticality of fully reproducible builds due to workload, missing organizational buy-in, unhelpful communication with upstream projects, or the goal being perceived as only theoretically achievable. Relatedly, the frustrating experience of "moving goalposts" was described as follows: Projects that had achieved bit-by-bit R-Bs might "lose" that status when someone found a previously unchecked source of mutation in the environment that broke full reproducibility.

Seven participants mentioned discussions about which mutations of the build environment should be checked for fully reproducible software packages, including a feeling of unclear goals, as noted by P01: "[*Projects*] *also have various definitions of what reproducibility means. It means you have some mysterious cloud, and in the end, you get the very same binaries, the bit-wise identical binary. That is the output, but what is considered as part of the input is not very clear.*" (P01) Seven participants, again across industry and OSS, mentioned detractors to implementing R-Bs, such as missing organizational buy-in, as mentioned by P10: *"To say, 'Reproducibility is stupid; go away.' That happens very rarely. We just remember it a lot because it's interesting. The most common interaction with upstream is silence. They just don't merge the patches,"* (P10) or due to their unwillingness to change their build process, as mentioned by P06: "[*project*] *is an old project and some areas are very conservative.*" (P06) One interviewee compared a lack of reaction to suggestions that contribute to R-Bs to a general quality problem in projects: *"*[*E*]*arly warnings that you know this and that upstream has some problems with their definition or they don't want to accept the patch about certain epoch, specification, or something like that."* (P07) The notion of unclear goals that change over time when a new source of unreproducible behavior occurs, was echoed by P08:

> *"Because we are taking a 'fix when we find' approach, I don't think we are doing anything*

> *to evaluate whether a package has achieved 100% reproducibility. Usually, when we fail to register a package to Reprepro, we know there may be a reproducibility issue and will look into it. Once we fix it, we'll do a clean build a few times to make sure the same binaries are produced. It sounds more like 'we make sure it's reproducible for now'.*" (P08)

Personal reproducibility target changes were discussed in 13 interviews. These changes in goals include being content with repeatable builds (i.e., compiling the same source code at different times into a functional artifact, not necessarily with the same bit-by-bit result) instead of fully R-Bs. One participant expressed hope that the community would value and work on the guarantees fully R-Bs provide over e.g., repeatable builds: *"It's one of those things where if more people in the community value this, they would get solved. Right now, they're happy with the builds being reproducible-ish."* (P04)

> Summary: Reasons for and against adopting Reproducible Builds. Interviewees were mostly driven by improved software quality. The unclear impact of R-Bs on the overall security of deployed software systems, together with high effort, were reasons against R-Bs.

Reproducible Builds as a Protection Mechanism. By design, R-Bs can serve as a protection mechanism for software projects against security and quality problems that might be introduced—maliciously, through coercion, or through mistakes—by software developers. Many interviewees (14) mentioned R-Bs as a security measure against coercive attacks against developers by malicious actors:

> *"The time where you have a room full of Debian developers and you tell them, 'If your computer is compromised in a way, then you might unknowingly compromise millions of machines'. People were like, 'I am this kind of target.'* [...] *It was a way to remove some leverage for a malicious actor to actually go at the people directly. If you would try to kidnap my kids and say, 'You need to plant this malware.' I can say, 'I can't. It's going to be seen, so probably you should do it differently and give me back my kids.' If you don't have reproducible build systems in place, then they have leverage because it's going to go unnoticed if you release a binary that doesn't match the actual source code.*" (P06)

R-Bs were also mentioned as a requirement for checking OSS, and making explicit trust in individual maintainers redundant. Some participants (13) mentioned that explicit trust in open source software project maintainers was not needed since open source code can be audited by anyone, at any given time. This possibility of having one's work audited was discussed as an incentive for honesty, and not wanting to be publicly seen as dishonest, which in turn was discussed as a powerful motivator motivator to safeguard the security of open-source software projects against powerful (non-)government agencies and private security actors. R-Bs were discussed as one strong mechanism to facilitate public scrutiny. In turn, participants discussed that for R-Bs to be an effective public security measure, openness is a requirement.

Eleven participants mentioned that their user base requested improved security, including specific requests for R-Bs, based on an awareness of one or more highly impactful, publicized security incidents, including the Snowden leaks, the Heartbleed vulnerability, the introduction of the GDPR, or the SolarWinds incident.

> *"It was clearly a need. There were a lot of the NSA, Snowden revelations coming out, and various things like that. Nothing specifically from those revelations, but the gist of a general vibe of, oh, we can trust even less than we previously thought as a very general idea, and the world, post-2015, is moving to a more data-conscious and privacy security."* (P10)

Although many participants mentioned requests for reproducibility due to highly visible security incidents, they also explained that these requests originate from only a small part of their user base—security-affine power-users—that reported concerns and requested R-Bs as a preventative measure against software supply chain attacks. *"It is perceived as one of those very rarely visited corners and only in cases like breaches and things like SolarWinds and attacks against suppliers."* (P07) Most of our interviewees (23) could not name a security incident related to reproducible builds, either thwarted or caused by it. However, one interviewee mentioned the following case:

> *"I had a package that was not reproducible when I checked. The difference between my rebuilds and the reference builds that are in [binary software repository] is the passphrase of a GPG key. During the builds, it was recorded because the command line that he used for signing, he added it in the parameter. I don't know why he's recorded it into a file that was then merged into the archive, and it is in [binary software repository]."* (PS23)

In the above example, R-Bs helped to detect a secret leak and contributed to the overall security of the project.

> Summary: Reproducible Builds as a Protection Mechanism.Attacks on software supply chains were an important driver to build software reproducibly.

Software Project Decision Structures. For both, open source and commercial software projects, interviewees reported that decision structures had a strong impact on R-Bs.

Although many OSS projects make important decisions by consensus, only seven participants reported a joint decision process to start moving the project to implement R-Bs. Some interviewees (11) reported that individual developers in their projects had independently started work on R-Bs due to intrinsic motivation.

For commercial projects, decisions were driven by commercial product deadlines, and decisions were generally made at the management level. Five participants reported that their project started moving towards implementing R-Bs through management decisions, heavily influenced by developers' insistence on what they perceived to be a contribution towards product quality.

> Summary: Project Decision Process.Individual developers have an influence on R-Bs adoption. They either drove R-Bs adoption by starting the process themselves or influencing project decisions.

5.5.2. Experienced Obstacles

In this section we report obstacles our interviewees experienced while working on R-Bs. Obstacles include unexpectedly large efforts, unsupportive upstream projects, and development processes that might be unusable, undefined, chaotic, or inconsistent. We discuss challenges and opportunities relating to community support in Section 5.5.3.

Lack of Good Communication. In general, eleven interviewees mentioned the relevance of strong communication skills to implement R-Bs in open source software. This communication involved heavy discussions of the concept of R-Bs, the goals achieved by R-Bs, and the need to adopt R-Bs. A common theme related to communication was a lack of outreach. The Reproducible Builds project's outreach consists of maintaining a website and mailing list, as well as paying one developer to post monthly progress reports, as they report on reproducible-builds.org. This included building the website's infrastructure for reporting successes in R-Bs, including testing OSS projects for reproducibility criteria themselves. However, to move beyond outreach and towards a wide adoption of R-Bs, participants discussed the need for wider support beyond those already involved in the R-Bs project. Participants communicated that more people working on R-Bs would be beneficial and would move the needle on software supply chain security. Eleven participants reported feeling not having done enough outreach-related work themselves, having instead focused on solving individual reproducibility changes in the code for which they felt responsible. One interviewee discussed the issue of transferring research advances into code, a problem also seen in prior research [199]: *"I think there's this big gap between scientific research and programming.* [...] *I think bootstrappability and reproducible builds could be an enormous boost for free software, and inspire people to move towards free software and free software practices."* (P17) Eight interviewees mentioned a lack of and a higher need for more helpful documentation for R-B efforts in a software project. The documentation on the R-Bs central website was also mentioned as needing more work.

Seven of our interviewees mentioned experiences of unhelpful interactions with other developers or users who were unsupportive or had difficulties understanding the concepts and benefits of R-Bs and the required effort. This led to delays in achieving R-Bs, including projects for which the upstream communication is still ongoing or stuck in a bug tracker. A total of eleven developers were astonished by the amount and importance of good communication for R-Bs. Eleven interviewees mentioned patience as a virtue in communicating with upstream projects that were less motivated to make reproducible builds part of their project. Participants explained that "good etiquette" in the open-source ecosystem is to proliferate bug reports upstream, ideally accompanied by a patch. The goal behind this approach is to fix issues at the source, getting rid of the burden of maintenance of the local changes. Our interviewees often went beyond that: five told us that they iterated with upstream projects and "polished" their patches until the upstream projects accepted them.

> Summary: Lack of Good Communication.The impact of interaction with other developers was often initially underestimated; participants discussed the importance of patience and good communication.

Technical Obstacles. Binaries including dates or other point-of-time information were the most common issues for reproducible builds reported by our interviewees. Reasons our interviewees gave included debugging artifacts, and different software version commits. eight interviewees mentioned the SOURCE_DATE_EPOCH standard [226] as an effective solution to the above problems. The SOURCE_DATE_EPOCH standard replaces random build time information with epoch.[5]

Five interviewees mentioned build directory name inclusion (cf. Section 5.3) that hindered their adoption of R-Bs.

> *"There has been some pitch to also support build path prefix, someway, somehow, but I don't know how to use it. From my approach, the OCaml compiler is not relocatable at the moment. It will be in the future eventually, but I'm not too concerned about it because I don't think there's any threat model that contains the build path, and in the end, I'm fine with recording the operating system packages, and the environment variables that led to that binary. Then I'm conducting the builds in a container, or in jail."* (P01)

Three interviewees mentioned that compilers include randomness explicitly during the compilation process, but also that a deterministic initial value can be supplied (for example via gcc -frandom-seed-string): *"I think we set the python seed. I think we set the python seed with Python. Um. The sort of state I'm trying to think of the other languages that they have, other weird things like that."* (PS20) Only one interviewee mentioned the potential issues around Profile-Guided Optimization [290], which change optimizations based on execution on the compiling machine. four interviewees reported (embedded) cryptographic signatures as implemented in the Apple ecosystem with enforced cryptographic signatures on binaries as a problem for R-Bs. Seven interviewees mentioned an unclear definition of build reproducibility as an obstacle to adopting R-Bs. Discussions in the community range from full, bit-by-bit R-Bs down to repeatable builds.

Overall, most interviewees mentioned that the technical obstacles above are manageable for people interested in R-Bs, but there is a long tail of problems to fully R-Bs.

> Summary: Technical Obstacles.Interviewees reported a wide range of technical obstacles including embedded timestamps, signatures, and build directories, which they assessed as intellectually simple, but cumbersome and repetitive to solve.

5.5.3. Helpful Factors

Helpful factors most mentioned were being self-effective, which interviewees defined as being determined, possessing the skill-set to progress R-Bs, and having good communication with other developers.

[5]The epoch value is the UNIX system time 01 Jan 1970 00:00:00 UTC

Self-effective Participants. 14 interviewees implemented R-Bs by themselves, through trial and error with some software that they tried to reproduce, which they described as the most efficient pathway to R-Bs. They described that a self-effective work environment, including having ownership of a large part of a project, being able to implement changes themselves, and prioritizing tasks as well as work packages themselves greatly contributed to effective work on R-Bs. We did not discover any different path to R-Bs in our interviews. Relatedly, interviewees explained hardships in increasing community efforts towards R-Bs, as the reserve of developers productively contributing to R-Bs. Anyone who could meaningfully contribute would need a high level of specialized knowledge and familiarity with the project; however, working on R-Bs might not be the most attractive work that these highly skilled open source developers might want to work towards.

> Summary: Self-effective Participants.Interviewees reported that specialized knowledge about projects, as well as the enthusiasm and ability to take broad action made their R-Bs work possible. These circumstances seems hard to scale to volunteers from the broader community.

Successful Community Communication. We identified factors that support R-Bs, often centering around effective community interaction. 14 participants mentioned positive interactions with upstream projects. These interactions included benign disinterest:

> *"Well, the first thing I do is ask them if they're aware of the problem because there's still a decent percentage of even developers that aren't fully aware of the reproducibility problem. I think it's solved at the package manager level. You know, they think that there's some sort of layer of it and it gets kind of resolved. Then from there, you know, I help them to understand the types of things that like this scope can help them understand the ways to ability can be compromised."* (P15)

Positive interactions also included encouraging cooperation and work on upstreamed patches: *"We got in contact with upstream and they fixed it, and now it's working fine."* (P19) In addition to interacting with upstream projects, communication with compiler authors was mentioned as helpful. Six interviewees reported positive interactions with compiler authors regarding R-Bs. Like other upstream requests, interviewees reported providing patches to compiler authors to address R-Bs issues, which were later incorporated.

> Summary: Successful Community Communication.Being helpful to upstream projects helps create goodwill for R-Bs in the open source community. Upstream projects may spend time and effort on R-Bs if they receive help from the R-Bs community for their own software project.

Helpful Resources. Different resources that helped with R-Bs were discussed in the interviews, but the most helpful resource for implementing R-Bs can be summarized

as "good tooling." A total of 13 people mentioned that they found tooling particularly helpful, specifically the diffoscope tool [119]. Eight interviewees mentioned that projects should work on the future seamless integration of R-Bs into the build process. Eight interviewees mentioned additional resources they used for R-Bs, including the R-B's website and mailing list. Eight interviewees stated that documentation should be expanded to make onboarding new R-Bs enthusiasts easier. In contrast, P16 stated that existing documentation was sufficiently helpful for R-Bs, and that efforts should increase community awareness and buy-in toward more effective support for R-Bs.

5.6. Discussion

In the previous results sections, we establish the importance of effectively disseminating the benefits of R-Bs to a broader community and that communication with upstream projects is crucial.

Outreach

To better communicate the procedural, monetary, reputational, and general benefits that stakeholders can gain from employing reproducible builds, we discussed preliminary findings and potential benefits with participants of the Reproducible Builds Summit 2022 in Venice with additional related material provided in the Chapter C, and highlight key points from that discussion below.

The goal of an outreach effort is to describe benefits that a stakeholder of fully reproducibly built software may want. The benefits can be classified as time savings, monetary savings, reputational gains, and generally better results of the work with the software. A non-exhaustive list of stakeholders includes non-university research groups, universities, development corporations, security organizations (which can themselves be in a corporation), open-source projects, end users, and governmental organizations. For any of the potential beneficiaries of reproducible builds, multiple benefits may apply. While any user of the software may be an end user individually, most organizations have different needs as a whole.

Almost any software project can benefit from caching build results, giving decreased build times and better turnaround times for changes. A published reproducible build will not change and can be reproduced exactly as-is, so no retesting is needed since all previous tests for that build directly apply for a rebuild. Build debugging is simpler since any singular build that fails for some users can be reproduced for finding the bugs in it. Developers can work a lot faster and with more confidence that bugs they introduce can be bisected and figured out, while at worst, any previous version can be used. The open-source project can also save on hardware resources, since build artifacts can be deduplicated effectively, meaning that only parts that changed need to be saved again. This can also be used for updates, where only the differences between updated versions need to be transferred to each user, saving bandwidth (cost). For their reputation, the open source project can fulfill more parts of the OpenSSF

scorecard [281]. While the project does not directly gain a reputation from higher software quality in its dependencies, however, choice of dependencies has an effect on the reputation of the open-source project in question. Incorporating reproducibly built dependencies may therefore indirectly increase a project's reputation.

In general, open source projects gain faster builds and the ability as well as a guarantee that they can build their software at any point in the future. All of this applies with one additional benefit: They learn which software components were used in building their software. This binary introspection can be a hard task in itself, but R-Bs can give this information almost for free using ".buildinfo"-files or a Software Bill of Materials (SBOM), as elaborated in analogy by P21:

> *"I don't grow plants, I don't create food, I don't write recipes, I don't prepare meals, but I do research on how to tell people to wash their hands. This does not make the food taste any better, make it better in general, more healthy. Maybe it makes the food a little bit less unhealthy or poisonous, but most times you'll be fine eating unwashed food. We try to change the way food is prepared. This does not influence the food at first but makes it better, safer, more healthy in general. We want to change the way people prepare food, just like water sewage, and treatment systems have been installed before. We want to change the mindset. We want to remind people, that reproducible, deterministic software is possible and reasonable.."* (P21)

Answers to RQs

Our 24 semi-structured interviews with experts involved with reproducible builds projects provided the following answers to our research questions:

RQ1: *"What are motivations for, and common themes around, adopting reproducible builds in software projects?"*

Our interviewees mentioned complex, interconnected motivations in the context of R-Bs. Some motivations are related to intuition for how builds should behave such as being deterministic as well as motivations grounded in explicit security concerns such as a compromised maintainer account.

RQ2: *"What experiences and challenges did projects encounter in the context of reproducible builds?"*

Many interviewees mentioned positive interactions with upstream projects and other developers, although some specifically noted that upstream communication required patience.

RQ3: *"What are encountered obstacles and facilitators in projects' efforts towards reproducibility?"*

Commonly encountered obstacles to reproducible builds include build directory name inclusion and cryptographic signatures on the technical side, as well as patience and good social communication on the interaction side.

Additional Insights

Some interviewees suggested that the overall awareness and buy-in for R-Bs was lacking, and that even with the increase of prevalence of software supply chain attacks, R-Bs is not yet widespread. Participants reported that in the early days of the R-Bs effort, most work appears to have been invested into infrastructure, including the upstreaming patches and creation of tooling, which should in theory provide a foundation for developers to make leaf packages build reproducibly.

Based on our participants' answers there appears to be a still ongoing public discussion about which criteria need to be fulfilled to call a package reproducible. The clearest criterion is bit-by-bit identical build results, which we opted to use in this paper. However, even bit-by-bit identical builds is subjective with respect to the mutators used to evaluate packages. Currently, the most used deployment of reprotest does not check all available mutators while testing a package for its build reproducibility.[6] A test with those or other not yet found sources of unreproducible behavior may change some bits in the artifact and give way to a stricter definition. Definitions of R-Bs that allow for differences in files from embedded signatures make R-Bs on Apple devices possible. These definitions specify elements in files that may be otherwise bit-by-bit reproducible.

Many groups do not enforce R-Bs. The Debian policy suggests rather than enforces reproducibility, which is understandable: users want to use software even if it is not built in a reproducible way. The OpenSSF scorecard also only cites R-Bs inside a high-risk criterion named "non-reviewable code," a detail that is fairly buried in the documentation.[7] In the case of open source projects, missing organizational buy-in comes in the form lack of support to mark R-Bs bug reports as blocking for a new software release. Untested changes from R-Bs can break build systems, so projects being conservative about patches is understandable.

For transitive dependency problems, concrete technical documentation could be achieved by the pervasive use of Software Bills of Materials (SBOMs) [96] to indicate all software included in building an artifact, so the transitive dependencies could be traced over a dependency graph.

Neither of President Biden's Executive Orders for Cybersecurity [136] or Supply Chains [135] mentions R-Bs. While this promises some eyes on the criticality of (Software) Supply Chain Security, we are worried that neither adequate funding for the work of mostly hobbyist open source developers nor real changes or competent help are to come in the foreseeable future. Lack of funding and organizational buy-in for their work was one of the major detractors mentioned in our interviews.

While we do not have insights into governmental regulation efforts, at some point we expect to see some regulation about software, similar to regulations about mandatory fitness of purpose and non-toxicity for other products. As also mentioned in one of our interviews, there are legislative efforts underway towards requiring an SBOM,

[6] https://reproducible.debian.net/, which now points to https://reproducible-builds.org/citests/ and is used by different OSS distributions

[7] https://github.com/ossf/scorecard/blob/main/docs/checks.md

which can only be reliably generated by having the depth of information as is needed for R-Bs.

Recommendations

A significant effort by a small number of individuals has laid the groundwork for R-Bs, fixing hard-to-find non-determinism in common build infrastructure. Despite these efforts, a significant number of participants regretted not spending more time on outreach. The knowledge and frameworks around R-Bs have reached a level of maturity such that now is the time for broader consumption. In this light, we conclude with the following recommendations.

1. We urge the industry to give their engineers leeway to work on what they deem necessary for software quality. They were the ones hired for their expertise and to know what is necessary for this and they should be empowered to work on it. Missing organizational buy-in by supporting their developers who may already want to get rid of some technical debt was one of the main detractors for R-Bs. Industry funding and engineering freedom were specifically mentioned in our interviews as wished for items regarding buy-in.

2. The open-source community should join the R-Bs effort and make it the new standard, so new releases are reproducible by default. Newly released unreproducible software should be permissible in distributions only if sufficient reasons and a plan to change are provided, creating more security for their users and themselves. Help with upstream interaction, R-Bs developers' small numbers, fatigue, and clarifying the status and importance of packages on the last mile to 100% R-Bs could help a lot. R-B's goal was seen as not clearly communicated, which led to some burnout with part of our interviewees. Better communication and avoiding "moving goalposts" would minimize the reported loss of emotional investment.

3. Not based on our interviews, but rather related work on the Trusting Trust attack and Software Supply Chain Security, we see R-Bs as a potentially greater interest to the security research community. We hope more buy-in from security organizations and researchers could be achieved by treating unreproducible software builds as a serious threat for software supply chain security and better support reproducible builds. We see some similarity of the R-Bs effort and security concerns, hinted at by one participant's remark about the search for mutators that make builds unreproducible, just like security vulnerabilities are searched for.

4. Governments should mandate some level of R-Bs as part of a general effort to strengthen software quality. Liability for last-level commercial, for-profit entities should be a necessary precondition for being allowed to profit from software products, just like it is common with physical products. This would create financial incentives for companies to provide R-Bs as a part of their software quality and security.

5.7. Conclusion

While R-Bs offer a strong foundation for securing the software supply chain, much of the industry believes it is out of reach. We conducted a series of 24 semi-structured expert interviews with participants from the Reproducible-Builds.org project with the goal of identifying insights that could lead to R-Bs becoming more commonplace in software development. Our findings include that the collaboration between highly motivated developers and upstream projects over long periods of time is a key aspect for the success of R-Bs. We identified a range of motivations for adopting R-Bs (**RQ1**), including indicators of quality, security benefits, and more efficient caching of artifacts. Discussions around process (**RQ2**) and obstacles (**RQ3**) confirmed many of the challenges discussed in this work.

The R-Bs effort to date has operated under the mindset of "infrastructure before leaf packages." It has required active and self-guided bug hunting to root out problems in the build infrastructure that have been long overlooked. While this approach has brought R-Bs far with very limited resources and persons, progress was in most cases achieved with only limited organizational buy-in, specifically by motivated individual open-source developers.

When companies *do* care for R-Bs, it is mostly seen as a cost-saving measure and only sometimes as a safeguard against wasting the time of highly-paid software engineers. The goals of quality and robust security motivate open source developers a lot more than corporate developers, though this may change due to the current geopolitical climate. In particular, new US initiatives have made R-Bs very tacitly, but mostly indirectly named, important to (inter-)national security. However, while the software supply chain security is seeing generous amounts of funding, nothing yet has been earmarked towards R-Bs.

In this chapter, we explain the influences of the Trusting Trust attack, the Diverse Double-Compilation defense against it, and the requirements both imply for a secure software supply chain. We interview experts from the reproducible-builds.org community and find themes of rather simple to solve technical problems, but rather hard economies of scale and fixing the last remaining infrastructure problems with a limited set of developers who care enough about them. We present possible solutions for gaining more awareness of the problem space and the solutions already partly in place in widely known open source projects, but are almost unknown in the wider software development community, if not derided as unnecessary. We follow up with recommendations for different addressees who may want to invest in a secure software supply chain and could get involved in reproducible builds to solve the problem of hidden, self-replicating backdoors in compilation tool chains in a systematic way.

6. Conclusions and Future Work

Disclaimer

The content of this conclusion was expanded and adapted from a previously peer-reviewed and published work. Please see the disclaimer in Chapter 1 for more details.

6.1. Cryptography is a Cornerstone of Security, but not Universally Checked

We have shown in Chapter 3 and Chapter 4 that the constant-time criterion has big implications on the security of cryptographic software implementations. Checking code for this is nontrivial and the tools to do so have a range of problems.

As verified through qualitative and quantitative studies, the state of the art of program verification of cryptographic algorithm implementations is not adopted widely in practice. Even if some form of verification is used, developers who select a cryptographic algorithm implementation for their project should look into the details about presumptions and difficulties in adopting a specific implementation. Upstream developers, who are cryptography experts, not necessarily have time to work around difficulties in program analysis frameworks and tools and may opt to favor a general purpose programming language codebase with easier integration over one that is easier to secure.

The practical security of widely used cryptographic libraries lags behind the most advanced cryptographic primitives from dedicated research groups. This may also be due to a gap between research groups and OSS developers, who may be working in their spare time as a hobby or who may be paid by a company to work on projects as a part of their duties.

6.2. Human Factors in Supply Chain Research

Supply chain security (SCS) is a crucial area of concern for businesses operating in today's connected economy. However, as with many aspects of security, we think the human factors involved in the design, implementation, and use of SCS measures have not received the attention they deserve. We believe that the usability of SCS measures for the people who will use them, especially developers, is a critically under-investigated

area of research. To address the need for more human-centered SCS approaches, we propose multiple aspects for a high-level research agenda below.

Motivations, Challenges, and Personal Risks. Integrating external software components into software projects presents unique challenges and risks for developers. The development of the external components often involves different developers with a diverse set of technology stacks, motivations, and development as well as communication cultures.

There is a responsibility that comes with relying on external build components and processes—in principle transparent, in practice infeasible to completely vet across fragmented ecosystems. The lack of clear hierarchies and trust relationships across different software components can also make it easier for attackers to target and exploit vulnerabilities.

To address these challenges, we think it is necessary to research and promote systematic, usable communication across creators of different components along the software supply chain.

Usable Tooling. Ideally, new functionality is integrated into versatile tooling that developers are already using; it remains unlikely that tools with unique interfaces and functionalities will be widely adopted to check for individual security properties.

To create build artifacts that can be trusted in an informed manner, not through organizational authority, one needs to know everything included in the creation of said artifact; see the Trusting Trust attack [354].

While we applaud [144] the work of the reproducible-builds.org project, this should only be the baseline and software build reproducibility—bootstrappability[1] and diversified checking à la David A. Wheelers "Diverse Double-Compiling" [393] should be further security targets. Their benefit needs to be prevalent in deployed software and not limited to academic study to gain practical advantages for every end user. This prevalence may yet entail publishing the source code and complete build recipes of all software that may have an effect on user security, to have them automatically recompiled and the results checked after variation in the environment, bootstrapping the necessary dependencies cleanly.

All of this needs to be low effort for a wide range of developers with different technology stacks. Further research is needed to explore which problems can be solved with tooling, and how tooling can communicate with its users in a way that allows for wide adoption.

Build Processes and CI/CD. Today's build systems and CI/CD pipelines can interact and chain with other systems and third-party services, allowing for the creation of complex, multi-step build and distribution processes for software. But this complexity also increases the risk of misuse, misconfiguration, or leakage of secrets, and

[1] https://bootstrappable.org/

as software is increasingly being built and deployed using third-party services, these services are becoming high-value targets for attackers seeking to infect all customers and compromise their SSCs. This is highlighted by a recent CircleCI security incident, where an attacker used malware on an engineer's laptop to gain unauthorized access to CircleCI's production systems.

As build systems and CI/CD pipelines are becoming increasingly complex, it is essential to not overlook the human factor in setting up, maintaining, and using them. Usability plays an important role in ensuring that developers can effectively and securely utilize and manage these complex systems: By establishing what makes these usable for stakeholders, such as clear documentation, user-friendly interfaces, and effective training materials, we as researchers can reduce the risk of misconfigurations and other vulnerabilities while allowing developers to maintain productivity and workflow efficiency.

Dependencies. Allowing developers to leverage external dependencies as building blocks for their software (e.g. from package repositories like npm or PyPI) is an important advantage of the SSC. However, the reliance on external repositories also introduces new attack surfaces, as recent typo-squatting and account-takeover attacks have shown. In response, Pythons PyPI and Microsofts GitHub have begun requiring two-factor authentication (2FA) for developer accounts with critical projects. While such approaches may increase the overall security of package repositories and dependencies, it is crucial to also consider their usability. I.e. if 2FA is required, but the authentication process is too complicated or time-consuming, developers may find ways to bypass it, which would undermine the intended security benefits. Developers need to keep track of changes in dependencies, so good communication practices about those are necessary.

By examining the common challenges and usage patterns involved in using and providing external dependencies, researchers can identify ways to improve the adoption of security processes, ultimately enhancing the security of the SSC.

Usable and Acceptable Authentication for Developers. Developers create our digital tools, but how do we know that the tools we have on our computer are actually created by developers we trust? This is a problem that has been well-studied and can be solved with cryptographic signatures. With the most common form of digital signatures in open source projects being the venerable OpenPGP signature and Web of Trust, both for authenticating developers as well as the artifacts of their labor. Some projects have sprung up thinking about how to solve this use-case in a more user-friendly way without bringing in the possibility of impersonation by some (trusted) third party. One of those projects favored by the industry is sigstore with ideas from certificate infrastructures, while more security-minded software distributions have opted for simpler tools like OpenBSD's signify.

This divide seems to be irreconcilable without investigating this area to find a universally acceptable solution to the authentication problem that includes all potential

user concerns while being more user-friendly than currently (at least partially) established practices. Having a universally accepted standard for authenticating developers and their work would heighten the level of security in SSC against impersonation, and can only be established by research that centers the users of this mechanism.

Metrics and Frameworks. In the context of a secure SSC, metrics, and frameworks that classify security vulnerabilities (CVE, CVSS, VEX), weaknesses (CWE), or coding practices (OpenSSF Scorecards) play an important role in communicating between stakeholders. Adoption is a critical factor in the effectiveness of any metric or framework. If these tools become widely adopted, they create a network effect, whereby stakeholders become familiar with the metric and share a common understanding of what constitutes secure software development practices. As more stakeholders adopt and utilize a particular metric or framework, they build a collective understanding of what it takes to develop secure software (according to the metric), leading to a higher level of standardization and consistency in secure software development practices.

Designing these tools around the metrics stakeholders actually care about, and centering usability are key in making these tools effective and widely accepted; without proper consideration of the human factor, these tools may not be utilized to their fullest potential or even adopted at all. Additionally, when more stakeholders utilize these tools, they can provide feedback on how to further improve their usability, resulting in a continuous improvement cycle. By investigating and improving their usability, researchers can increase the likelihood that stakeholders will utilize these tools and ultimately establish a common understanding towards a more secure software ecosystem.

Open Source vs. Closed Source Conflicts. Large companies sometimes adapt Open / Libre Source Software (OSS) projects, and may have internal changes to them that they maintain, update, develop, and never contribute back into the OSS space [391]. This may also be true for dealing with vulnerabilities and incurs costs for companies as well as the OSS ecosystem where the software originated.

However, there are legitimate reasons for forking and maintaining in a closed environment: oftentimes, the OSS community may not want to develop in the direction that a large company requires (e.g., Google, boringSSL). A company may look at its plate first instead of the whole upstream bowl.

There are trade-offs, due to the work required for maintaining each internal fork a company may keep internally—each one has to be kept up to date with security patches, inventoried, and kept on watch for vulnerabilities and plain errors being fixed upstream. While having an internal cache of external dependencies brings benefits like keeping each dependency available even facing upstream disaster, they are certainly not for free and tend to break if left without care. That care may even be a full internal fork with in-house patches, but they too need to be maintained, making mid-term costs skyrocket. Costs to interact with OSS—one-time setup, repetitive—may be different on each project.

A deeper understanding of better cooperation possibilities may be more economical, provide more prompt reaction to security incidents, and be in the best interest of all parties overall. Centering the needs, challenges, and decisions of stakeholders in human factors research can help improve cooperation in this space.

"One Guy in Kansas" Some open source software is so ubiquitous in the development and operation of IT systems that its existence is hardly noticed. Its absence, or even just unfixed bugs, could incur large costs and other damages for big organizations however. Surprisingly, some of that software was developed, or is maintained by very few developers, colloquially known as "that one guy in Kansas." This is specifically true for many open source projects, which are often done by default as hobbies, not contributing significantly to the income of their main developers. Research into how to better support these small projects may have a significant impact on the security of the SSC overall.

In conclusion, we need to consider human factors to secure the SSC, dependencies, and build systems. Recent attacks have demonstrated that developers are working on every link in the chain, making approaches that consider the human factor an important step for effective SSC security.

6.3. Outlook

In this thesis we have shown the importance of quality and checking for its criteria in high-value code that comprises some of the most critical parts for securing our software supply chain security. This cannot be done without high quality implementations of cryptographic software libraries, which have their own implementation pitfalls, to be checked for. To get those implementations not just deployed, but also developed, we need to look more into the human factors which hinder adoption of academic solutions into real world code.

With that cryptography, we can implement trust relationships and assign trust to artifacts, checked for reproducibility by different, mutually trustful—secured by cryptographic signatures—entities, without any need for centralized trust to any authority. Those trust relationships are a huge, underinvestigated part of how software is generated by humans. Getting those trust relationships into a shape implemented by simple and usable tools that even non-expert users can benefit from still seems to be an open problem, to be researched just as more ergonomic forms of cryptographic authentication to surpass nineties era cryptographic tools for general use in all OSS projects. We must look into usable security to get security benefits to people who may have other priorities before that, by finding out where additional effort is enough of a roadblock to adoption without being necessary for the intended benefits.

While some projects are distrustful of corporate industry developments, others try these out to see if at least some form of security and usability advancements can be gained over not doing any artifact signing and checking at all. This area should be researched further to find a simple and ubiquitous solution that can be held to the

absolutely highest standards of verifiable trust, so a common standard for software supply chain security can emerge from it.

Appendices

A. What Cryptographic Library Developers Think About Timing Attacks

A.1. Survey

A.1.1. Background

Q1.1: How many years have you been developing cryptographic code?
[Numeric field]
Q1.2: What background do you have in cryptography?

- ☐ Academic
- ☐ Took some classes
- ☐ On the job experience
- ☐ Teach it
- ☐ Hobby
- ☐ Industry
- ☐ Prefer not to say

Q1.3: Can you tell us a little bit more about your background as a developer who works on cryptographic libraries/primitives?
[Free text field]

A.1.2. Library / Primitive

Q2.1: What's your role in the development of *library*? (E.g., maintainer, project lead, core developer, commit rights, no rights, etc.)
[Free text field]
Q2.2: How are you involved in design decisions (e.g., concerning the API, coding guidelines and style, security-relevant properties) for *library*?
[Free text field]
Q2.3: What are the intended use cases of *library*? (E.g., embedded use, servers, etc.)
[Free text field]
Q2.4: What is the threat model for *library* with regards to side-channel attacks? (E.g., local/remote attackers, etc.)
[Free text field]

Q2.5: Do you consider timing attacks a relevant threat for the intended use of *library* and its threat model? Please give a brief explanation for why / why not. (If the execution time of a program depends on secret data, a timing attack recovers information about the secret by computing the inverse of this dependency. The two most notorious sources for such dependencies are secret dependent control flow and secret-dependent memory access. Timing attacks include cache attacks where the attacker uses the cache to infer information about memory accesses of a target.)
[Free text field]
Q2.6: Does *library* claim resistance against timing attacks?

- ○ Yes
- ○ No
- ○ Partially
- ○ I don't know
- ○ Not yet but planning to

Q2.7: How did the development team decide to protect or not to protect against timing attacks? (We are interested in the decision process and not the protection mechanisms themselves (if any).)
[Free text field]
Q2.8: [only shown if **Q2.6** is "Yes" or "Partially"] How does *library* protect against timing attacks?
[Free text field]
Q2.9: Did you personally test for or verify the resistance of *library* against timing attacks?

- ○ Yes
- ○ No
- ○ Partially
- ○ Not yet but planning to
- ○ Not me but someone did
- ○ I don't know
- ○ Prefer not to say

Q2.10: [only shown if **Q2.9** is "Yes" or "Partially"] How did you test or verify the resistance against timing attacks? (E.g. using which tools, techniques, practices.)
[Free text field]
Q2.11: [only shown if **Q2.9** is "Yes" or "Partially"] How often do you test or verify the resistance of *library* against timing attacks?

- ☐ Only did it once
- ☐ Do it occasionally
- ☐ During releases
- ☐ During CI
- ☐ Don't know
- ☐ Prefer not to say

A.1.3. Tooling

Q3.1: Are you aware of tools that can test or verify resistance against timing attacks?

◦ Yes ◦ No

Q3.2: Please tell us which of these you've heard of with regards to verifying resistance against timing attacks.
[List of tools from Table 3.1.]
Q3.3: How did you learn about them? (Check all that apply)
[Matrix question with subquestions being the tools the participant selected in **Q3.2** and the following answer options:]

☐ Recommended by colleague

☐ Heard from authors

☐ Read the paper

☐ Referenced in a blog/different paper

☐ Was involved in the development

☐ Other

Q3.4: Which of these (if any) have you tried to use in the context of resistance against timing attacks?
[Multiple choice question among the tools selected by the participant in **Q3.2**.]
Q3.5: Why have you not tried to use these?
[Multiple free text fields for all of the tools the participant did select in **Q3.2** but not in **Q3.4**.]

A.1.4. Tool use

[All of the questions in this group are matrix questions with subquestions for all of the tools the participant did select in **Q3.2** and **Q3.4**, i.e. those tools that the participant knows and tried to use.]
Q4.1: Please describe the process of using the tools.
[Free text field]
Q4.2: I was satisfied with the installation process. (Please rate your agreement with the above statement.)

◦ I quit using the tool before I got to this point

◦ I quit using the tool because this was a problem

◦ Strongly disagree

◦ Disagree

◦ Neither agree or disagree

- Agree
- Strongly agree

Q4.3: I was satisfied with the prerequisites that the tool needed to work with my code. (Please rate your agreement with the above statement.)
[Same answer options as **Q4.2**]
Q4.4: In my understanding the tool is sound. (Please rate your agreement with the above statement. A sound tool only deems secure programs secure, thus has no false negatives.)
[Same answer options as **Q4.2**]
Q4.5: In my understanding the tool is complete. (Please rate your agreement with the above statement. A complete tool only deems insecure programs insecure, thus has no false positives.)
[Same answer options as **Q4.2**]
Q4.6: I understood the results the tool provided. (Please rate your agreement with the above statement.)
[Same answer options as **Q4.2**]
Q4.7: I was satisfied with the documentation of the tool. (Please rate your agreement with the above statement.)
[Same answer options as **Q4.2**]
Q4.8: I was satisfied with the overall usability of the tool. (Please rate your agreement with the above statement.)
[Same answer options as **Q4.2**]
Q4.9: I was satisfied with the tool overall. (Please rate your agreement with the above statement.)
[Same answer options as **Q4.2**]

A.1.5. Tool use: Dynamic instrumentation based

Q5.1: Use of dynamic instrumentation based tools like ctgrind, MemSan or Timecop requires:

- Creating test harnesses.
- Annotating secret inputs in the code.
- Compiling code with a specific compiler (in the MemSan case).

and in return detects non-constant time code that was executed (e.g. branches on secret values, or secret-dependent memory accesses). However, it does not detect non-constant time code that was not executed (in branches not executed due conditions on public inputs).

Do you think you would fulfill these requirements in order to use this type of tool?
[1 = Very unlikely, 2 = Somewhat unlikely, 3 = Neutral, 4 = Somewhat likely, 5 = Very likely]
Q5.2: Can you clarify your reasoning for the answer?

☐ Not my decision

☐ Not applicable to my library

☐ Would like the guarantees but too much effort

☐ Good tradeoff of requirements and guarantees

☐ Already using one of the mentioned tools

☐ Will try to use one of the mentioned tools after this survey

☐ I don't care about the guarantees

☐ None of the above

Q5.3: Please expand on your answer if the above question didn't suffice?
[Free text field]

A.1.6. Tool use: Statistical runtime tests

Q6.1: Use of runtime statistical test-based tools like dudect requires:

- Creating a test harness that creates a list of public inputs and a list of representatives of two classes of secret inputs for which runtime variation will be tested.

and in return provides statistical guarantees of constant-timeness obtained by running the target code many times and performing statistical analysis of the results.

Do you think you would fulfill these requirements in order to use this type of tool?
[1 = Very unlikely, 2 = Somewhat unlikely, 3 = Neutral, 4 = Somewhat likely, 5 = Very likely]

Q6.2: Can you clarify your reasoning for the answer?
[Same answer options as **Q5.2**]

Q6.3: Please expand on your answer if the above question didn't suffice?
[Free text field]

A.1.7. Tool use: Formal analysis

Q7.1: Use of formal analysis-based tools like ct-verif requires:

- Annotation of the secret and public inputs in the source code.
- Running the analysis via a formal verification toolchain (i.e. SMACK).
- Might not handle arbitrarily large programs or might require assistance in annotation of loop bounds.

and in return provides sound and complete guarantees (no false positives or negatives) of constant-timeness (e.g. no branches on secrets or secret-dependent memory accesses or secret inputs to certain instructions).

Do you think you would fulfill these requirements in order to use this type of tool?

[1 = Very unlikely, 2 = Somewhat unlikely, 3 = Neutral, 4 = Somewhat likely, 5 = Very likely]

Q7.2: Can you clarify your reasoning for the answer?

[Same answer options as **Q5.2**]

Q7.3: Please expand on your answer if the above question didn't suffice?

[Free text field]

A.1.8. Miscellaneous

Q8.1: Do you have any other thoughts on timing attacks that you want to share?

[Free text field]

Q8.2: Do you have any other thoughts on or experiences with those tools that you want to share?

[Free text field]

Q8.3: Do you have any feedback on this survey, research, or someone you think we should talk to about this research (ideally an email address we could reach)?

[Free text field]

Q8.4: Do you want to allow us to contact you for:

- ☐ sending you a report of our results from the survey
- ☐ asking possible follow-up questions

Q8.5: [Only shown if some of the options in **Q8.4** was selected] To allow us to contact you, please enter your preferred email address. (If at any time you want to revoke consent to contact you and ask us to delete your email address, please email [de-identified for submission])

[Free text field]

A.2. Tool awareness

Tool	Aware	%	Tried to use	%
ctgrind [229]	27	61.4%	17	38.6%
ct-verif [19]	17	38.6%	3	6.8%
MemSan [346]	8	18.2%	4	9.1%
dudect [307]	8	18.2%	1	2.3%
timecop [264]	8	18.2%	1	2.3%
ct-fuzz [181]	7	15.9%	1	2.3%
CacheD [381]	6	13.6%	1	2.3%
FaCT [85]	6	13.6%	0	0.0%
CacheAudit [125]	5	11.4%	0	0.0%
FlowTracker [316]	4	9.1%	1	2.3%
SideTrail [30]	3	6.8%	0	0.0%
tis-ct [109]	3	6.8%	0	0.0%
DATA [387, 386]	2	4.5%	2	4.5%
Blazer [29]	2	4.5%	0	0.0%
BPT17 [53]	2	4.5%	0	0.0%
CT-WASM [384]	2	4.5%	0	0.0%
MicroWalk [396]	2	4.5%	0	0.0%
SC-Eliminator [400]	2	4.5%	0	0.0%
Binsec/Rel [111]	1	2.3%	0	0.0%
COCO-CHANNEL [66]	1	2.3%	0	0.0%
haybale-pitchfork [363]	1	2.3%	0	0.0%
KMO12 [213]	1	2.3%	0	0.0%
Themis [91]	1	2.3%	0	0.0%
VirtualCert [42]	1	2.3%	0	0.0%
ABPV13 [21]	0	0.0%	0	0.0%
None	11	25.0%	25	56.8%

Table A.1.: Tool awareness and use

B. A usability evaluation of constant-time analysis tools

B.1. Summary of known CT analysis tools

Tool	Target	Tech.	Guar.	Available
Abacus [37]	Binary	Stat	○	Github
ABPV13 [21]	C	Fo	●	no
ABSynthe [162]	Leakage	Dyn	■	Github
ANABLEPS [382]	Binary	Dyn	○	Github
BINSEC/REL [111]	Binary	Sym	◐	Github
Blazer [29]	Java	Fo	●	no
BPT17 [53]	C	Sym	●	irisa.fr
CacheAudit [125]	Binary	Fo	■	Github
CacheAudit2 [126]	Binary	Dyn	●	Github
CacheD [381]	Trace	Sym	○	no
CacheFix [89]	Trace	Sym	◐	BitBucket
CacheQL [404]	Binary	Dyn	○	Github
CacheS [380]	Binary	Fo	◐	no
CANAL [348]	LLVM	Fo	●	Github
Cache Templates [165]	Binary	Stat	■	Github
CaSym [69]	LLVM	Sym	●	no
CaType [201]	Binary	Fo	●	no
CHALICE [88]	LLVM	Sym	■	BitBucket
COCO-CHANNEL [66]	Java	Sym	●	no
Constantine [56]	LLVM	Dyn	◐	Github
ctgrind [229]	Binary	Dyn	◐	Github
ct-fuzz [181]	LLVM	Dyn	○	Github
ct-verif [19]	LLVM	Fo	●	Github
CT-WASM [384]	WASM	Fo†	●	Github
DATA [387, 386]	Binary	Dy	◐	Github
DifFuzz [274]	Java	Dyn	○	no
dudect [307]	Binary	Stat	○	Github
ENCIDER [403]	LLVM	Sym	◐	Github
ENCoVer[36]	Java	Fo	●	kth.se
FlowTracker [316]	LLVM	Fo	●	ufmg.br
haybale-pitchfork [363]	LLVM	Sym	◐	Github
KMO12 [213]	Binary	Fo	■	no
Manifold [405]	Binary	Stat	○	Zenodo
MemSan [346]	LLVM	Dyn	◐	llvm.org
MicroWalk [396]	Binary	Dyn	◐	Github
MicroWalk-CI [397]	Binary	Dyn	◐	Github
mona-timing-[lib|report]	Network	Stat	■	Github & Github
PinCEC [196]	Binary	Dyn	◐	Github
Pitchfork-angr [364]	Binary	Sym	○	Github
SC-Eliminator [400]	LLVM	Fo†	●	Zenodo
Shin et al. [335]	Binary	Stat	○	no
SideTrail [30]	LLVM	Fo	■	Github
STACCO [401]	Binary	Dyn	○	no
STAnalyzer [326]	C	Fo	●	no
Themis [91]	Java	Fo	●	Github
timecop [264]	Binary	Dyn	◐	blog
tis-ct [109]	C	Sym	◐	no
TLSfuzzer [205]	Network	Stat	■	Github
TriggerFlow [163]	Binary	Dyn	○	Gitlab
VirtualCert [42]	x86	Fo	●	edu.uy

Targets: LLVM—intermediate representation, DSL—domain-specific language, WASM—Web Assembly, Network—network-reachable TLS implementation
Technique: Sym—Symbolic, Stat—Statistics, Dyn—Dynamic, Fo—Formal, †—also performs code transformation/synthesis
Guarantees: ●—sound, ◐—sound with restrictions, ○—no guarantee, ■—other property

Table B.1.: Classification of CT tools.

C. On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security

C.1. Codebook

1. Background
    a. Professional Experience
    b. Joining project
        i. Interests
2. Reasons
    a. (non-)technical
        i. Internal and External drivers
            A. Compilers work like mathematical functions
            B. Broken expectations
            C. Want to do/have quality
            D. Want to build infrastructure
            E. Wants to work on leaf packages
            F. Self-guided exploration
            G. Source code is out in the open, so anybody can see it
            H. Build speed/caching
        ii. Community requests
            A. Snowden leaks/security incident/privacy protection
        iii. Security for developers
    b. Threats
        i. Specific threats
        ii. Incidents or requirements
            A. Specific incident found by reproducible builds

   c. Project decision process
      i. Started themselves independently
      ii. Consensus
      iii. Corporate decision/management
3. Process
   a. People involved
      i. Communication
      ii. Detractors
   b. Starting topic
      i. Build process
         A. CI or other build infrastructure
         B. Version pinning
            1. compiler version
            2. transitive dependencies
      ii. Upstream interaction
         A. Community building
            1. Documentation work
         B. Communication
            1. Patience necessary
            2. Good communication skills more necessary than expected
         C. Reproducible Builds buy-in
         D. Common community requests
         E. Receptive upstream
         F. Receptive compiler authors
         G. Patch polishing was necessary
         H. Upstream rewrote patches themselves
      iii. Decision criterion for reproducibility
4. Tooling
   a. Helpful tools used
      i. Integration into build process
   b. Other resources used
5. Obstacles

    a. Technical
        i. Build date included, SOURCE_DATE_EPOCH
        ii. Build directory included in full, BUILD_PATH_PREFIX_MAP
        iii. Compiler included randomness (symbols, ...)
        iv. Profile Guided Optimization (PGO)
        v. Cryptographic signatures included in binaries
6. Helpful factors
    a. Self-effective participants
7. Target changes
8. Changes on starting over
    a. Regret not doing more outreach
9. How specific
    a. Direct port of procedures and tools
10. Misc
    a. Bootstrappable Builds

C.2. Questionnaire

Intro

- **Thanks**: Thank you very much for offering your valuable time for this interview. We are very grateful for your contribution.
- **Ready**: Are you ready to start the interview?
- **Structure**: First off, I am going to talk about the context and data handling, and if you agree with everything, we would then start with the interview.

Context

- **We**: We are researchers at [anonymized for submission]
- **Our research**: focuses on the area "Security impact of and experiences with reproducible builds".
- This interview is a start/exploration of internal processes and decisions often not visible at the technical level.
- For this interview:

  - We are not judging security or technological decisions of a project, we are just interested in the underlying structures and processes.
  - Projects are often very complex, if you don't know the answer, or cannot speak about a question for any reason, just say "next".
  - We are not just interested in structures, but also your personal opinions and experiences.

- **Questions?** Any questions about the interview context so far?

Consent

- Voluntary: Your responses in this interview are entirely voluntary, and you may refuse to answer any or all of the questions in this interview.
- Duration: Duration of the interview depends a bit on the duration of your answers,in our experience so far about X to Y minutes.
- We will de-identify you and your projects in any publication and only include short quotes.
- We will send you a preprint before a potential publication, if you want.
- Recording: We would like to record this interview so that we can transcribe the answers later
  - The recording will be destroyed when we transcribed the interview
- Questions? Any more questions about data handling or recording?
- I will now start the recording
- "The recording is now on" ***SWITCH ON RECORDING***
- Restate consent question

Section 1 - Intro [Personal / General / Project]

1. To start, we are interested in your background and that of [project we are interested in re: reproducible builds]. Please tell us a little bit about how you got involved?
   a. Coursework?
   b. Professional experience?
   c. How did you get into [project]?
      i. What did you find interesting about [project]?
      ii. And how long have you been working on [project]?

iii. Which programming languages are commonly used in [project]?

d. [if [project] uses more than on PL:] Specific programming language background/experience?

i. Do you have experience with some/all of the programming languages used in [project]?

2. Could you please elaborate a bit about your role in [project]?

a. What packages do you work on, what do they do?

b. How did you get from working generally on [project] to working on reproducible builds?

3. Could you explain what reproducible builds mean to you?

a. In the context of project

b. In general

In the context of our research, we'll be talking about reproducible builds in this interview.

When we say reproducible builds, we mean that for each version of the project, anyone can take the source artifacts and in the best case build a package that is bit-for-bit identical at any point now or in the future.

Comparison that two packages are built from the same source should, at the very least, be possible via human inspection.

Ideally, the comparison should be as automated as possible.

We want to allow different parties to determine if a binary package matches its source code, eliminating possible backdoors during the build process.

Section 2 - Reasons, Decisions

1. [project] has[/has not yet] made progress towards reproducible builds. We'll be very interested in the process in a moment. Before we start on that, we are interested in your reasons for making [project] reproducible?

a. Technical and also non-technical?

i. Internal/external drivers?
ii. Community requests (users vs. developers)?
iii. Security for developers? (by getting the software out of your sole control, making the build environment on your machine less of a target for attackers)

b. What threats are you protecting against?

i. Are there any specific threats?
ii. Specific incidents/requirements?

c. What were the reasons against making [project] or individual packages fully reproducible as of now? (not yet, or not at all)

d. How did the project decide? Who made these decisions, what roles did they have?

Section 3 - Process, Tools

1. What [was/is/would be/will be] your process of making [project] or individual packets reproducible?
   a. When did you start?
   b. Who were the people and roles involved in this effort?
      i. How did they communicate with each other? (*e.g., mailing lists, conferences/Bug Squashing Parties, issues, calls*...)
      ii. Were there detractors?
   c. What is your estimate of your time invested into it?
      i. Did that change over time?
   d. Where did you start?
      i. What was the strategy for choosing which packages to work on first?

(e.g., easy leaf packages first vs. important upstream dependencies first)

1. a. i. What is your build process like?
      A. Do you use some form of CI or other build infrastructure?
      B. What is the level of version pinning that you do for releases?
         1. Upstream compiler version subreleases?
         2. Transitive dependencies?
   ii. How do you interact with upstream projects used in [project]?
      A. Was there active community building,
      B. communication,
      C. buy-in into reproducible builds,
      D. any insights on successful / unsuccessful communication
      E. (common) community requests [rb or upstream]?
   iii. How did you decide that a package is reproducible?

(*prompt: 100% reproducible artifacts vs. specific test for reproducibility; explainable differences in certain data fields*)

1. [for libraries] Do you (want to) use other programming languages in the library and if so, why?
    a. Can you tell us about any interactions between programming language changes and reproducibility efforts? Did one of them impact the other?
2. What is your tech setup/specific tooling or other resources to help you make [project] reproducible?

    (*e.g., prompt for diffoscope*)

1. a. Please tell us about the tools or libraries you built or used to help make [project] reproducible?
        i. Did you integrate any of them into the CI or packaging utilities?
            A. How did that go?
    b. Were there any approaches, tools, that you abandoned on your way to make [project] reproducible?
        i. *Example: Binary diffing died*
    c. What type of tool – whether it exists or not – would you have liked to have to help with making your builds reproducible, or checking for reproducibility?
    d. Were there any other resources you used? (*e.g., documentation, knowledge bases, websites*)

Section 4 - Obstacles/Challenges, Facilitators

1. If there were any, please tell us about the obstacles involved in making [project] reproducible?
    a. Organizational (buy-in from different developers, not being able to do systemic changes in different parts of the project)
    b. Technical (hard dependencies on timestamps for identifiers etc.)
    c. Dependencies (upstream not being willing to take patches, marking change requests as invalid/WorksForMe)
    d. Of the differences between different programming languages and their communities, which influenced your effort in [project]'s reproducibility? And was that influence positive or negative? (bug reporting, error culture towards compiler changes, community responses to questions)
2. Which factors were particularly helpful in making your progress more manageable?

3. Did your target change due to difficulties in getting [project] to be reproducible?

   (*e.g. any compromises*)

Section 5 - Generalization / Lessons Learned

1. "If you had to start over, what would you do differently?"
   a. Would using current tools solve a lot of the problems you encountered?
   b. Are there any programming language specific things you wish you could have used?
2. What worked well, what didn't work?
   a. Can you tell us about the role of upstream patches and how they help or hinder reproducibility?
      i. How did your effort change over time while other packages worked on their reproducibility?
3. How specific would you say your process was to your project? We are interested in what can be generalized or already benefits other projects, and what's specific to yours.
   a. In your personal opinion, do you think that other projects could follow the same or similar procedures? (direct port of technical measures or organizational structures, similarities between biggest hurdles)
4. What would you recommend to other developers and projects?

Outro

1. Is there anything else you would like to tell us about reproducible builds, within or outside the scope of [project]?
2. Is there something that we did not cover during the interview but you would like to talk about?

Debrief

- ***SWITCH OFF RECORDING*** "The recording is now off"
- Could you recommend other projects or persons we could invite for an interview? They should have attempted to make their project reproducible.
- Thank the participant again for their valuable time
- Do you want to get a preprint of our paper?
  - If interested: what is your preferred contact data?
    * We will be in contact for a preprint

C.3. Motivational Matrix

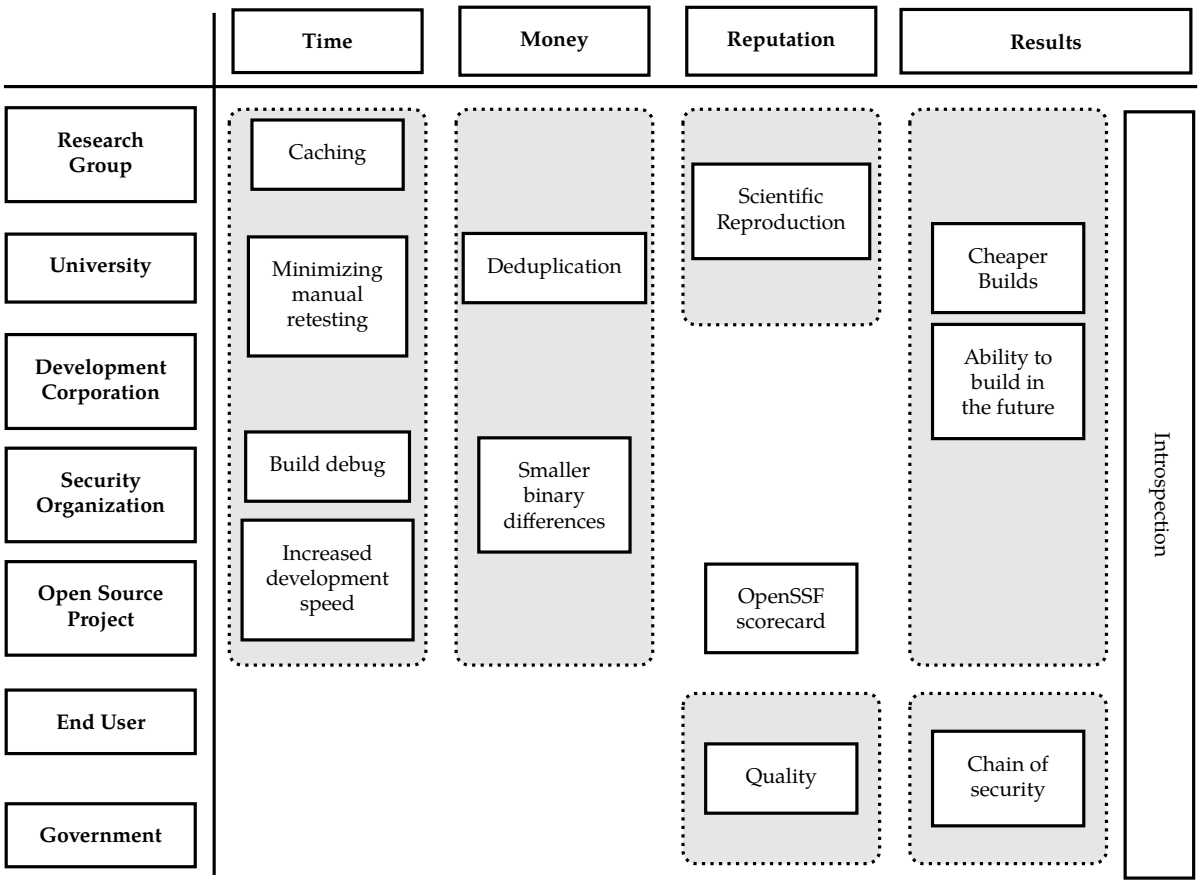

Figure C.1.: Motivational matrix, joint work from a discussion session with attendees of the Reproducible Builds Summit 2022.

D. Publication History

Prior to working on this thesis, the following papers were published:

1. Marcel Fourné, Dominique Petersen, and Norbert Pohlmann. *"Attack-test and verification systems, steps towards verifiable anomaly detection"*. In: *INFORMATIK 2013 – Informatik angepasst an Mensch, Organisation und Umwelt*. Gesellschaft für Informatik eV, 2013, pp. 2213–2224. ISBN: 978-3-88579-614-5

2. Marcel Fourné, Kevin Stegemann, Dominique Petersen, and Norbert Pohlmann. *"Aggregation of Network Protocol Data Near Its Source"*. In: *Information and Communication Technology: Second IFIP TC5/8 International Conference, ICT-EurAsia 2014, Bali, Indonesia, April 14-17, 2014. Proceedings 2*. Springer. 2014, pp. 482–491. ISBN: 978-3-642-55031-7. DOI: 10.1007/978-3-642-55032-4_49

In order of publication, the following peer-reviewed works were accepted for publication during the work leading up to this thesis:

3. Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. *""They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks"*. In: *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2022, pp. 632–649. DOI: 10.1109/SP46214.2022.9833713

4. Dominik Wermke, Noah Wöhler, Jan H. Klemmer, Marcel Fourné, Yasemin Acar, and Sascha Fahl. *"Committed to Trust: A Qualitative Study on Security & Trust in Open Source Software Projects"*. In: *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P'22)*. May 2022, won the distinguished paper award (see https://www.ieee-security.org/TC/SP2022/awards.html) but is not included in this thesis.

5. Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. *"It's like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security"*. In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 1527–1544. DOI: 10.1109/SP46215.2023.10179320

6. Marcel Fourné, Dominik Wermke, Sascha Fahl, and Yasemin Acar. *"A Viewpoint on Human Factors in Software Supply Chain Security: A Research Agenda"*. In: *IEEE Security & Privacy* 21.6 (2023), pp. 59–63. DOI: 10.1109/MSEC.2023.3316569

7. Marcel Fourné, Jan Jancar, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "*'These results must be false': A usability evaluation of constant-time analysis tools*". In: *33th USENIX Security Symposium (USENIX Security 2024)*. USENIX Association, 2024

List of Figures

Bibliography

[1] https://www.wired.com/story/worst-hacks-breaches-2020-so-far/.

[2] https://www.statista.com/statistics/800258/worldwide-meltdown-spectre-potential-mitigation-cost-by-device-type/.

[3] Ibrahim Abunadi and Mamdouh Alenezi. *"Towards Cross Project Vulnerability Prediction in Open Source Web Applications"*. In: *Proceedings of the The International Conference on Engineering & MIS 2015*. ICEMIS '15. Istanbul, Turkey: Association for Computing Machinery, 2015. ISBN: 9781450334181. DOI: 10.1145/2832987.2833051. URL: https://doi.org/10.1145/2832987.2833051.

[4] Yasemin Acar. *Human Factors in Secure Software Development*. Ed. by Bernd (Prof. Dr.) Freisleben. Philipps-Universität Marburg, 2021. URL: /diss/z2021/0231/pdf/dya.pdf.

[5] 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.

[6] Yasemin Acar, Michael Backes, Sascha Fahl, Simson L. Garfinkel, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. *"Comparing the Usability of Cryptographic APIs"*. In: *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 2017, pp. 154–171. DOI: 10.1109/SP.2017.52.

[7] 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.

[8] 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: *IEEE Cybersecurity Development, SecDev 2016*. Boston, MA, USA: IEEE, Nov. 2016, pp. 3–8. DOI: 10.1109/SECDEV.2016.013. URL: https://doi.org/10.1109/SecDev.2016.013.

[9] Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, eds. *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Dallas, TX, USA: ACM Press, Oct. 2017.

[10] Giovanni Vigna and Elaine Shi, eds. *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Virtual Event, Republic of Korea: ACM Press, Nov. 2021.

[11] Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, eds. *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Los Angeles, CA, USA: ACM Press, Nov. 2022.

[12] Hammad Afzali, Santiago Torres-Arias, Reza Curtmola, and Justin Cappos. *"Towards adding verifiability to web-based Git repositories"*. In: *Journal of Computer Security* 28.4 (2020), pp. 405–436.

[13] Abdulkareem Alali, Huzefa Kagdi, and Jonathan I. Maletic. *"What's a Typical Commit? A Characterization of Open Source Software Repositories"*. In: *Proceedings of the 16th IEEE International Conference on Program Comprehension*. 2008, pp. 182–191.

[14] Martin R. Albrecht and Kenneth G. Paterson. *"Lucky Microseconds: A Timing Attack on Amazon's* s2n *Implementation of TLS"*. In: *Advances in Cryptology – EUROCRYPT 2016*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9665. LNCS. Springer, 2016, pp. 622–643. URL: https://eprint.iacr.org/2015/1129.

[15] Mamdouh Alenezi and Yasir Javed. "*Open source web application security: A static analysis approach*". In: *2016 International Conference on Engineering & MIS (ICEMIS)*. 2016, pp. 1–5. DOI: 10.1109/ICEMIS.2016.7745369.

[16] Nadhem J. AlFardan and Kenneth G. Paterson. "*Lucky Thirteen: Breaking the TLS and DTLS Record Protocols*". In: *2013 IEEE Symposium on Security and Privacy*. Berkeley, CA, USA: IEEE Computer Society Press, May 2013, pp. 526–540. DOI: 10.1109/SP.2013.42.

[17] Said Ali, Oscar Nierstrasz, and Mohammadreza Hazhirpasand. "*Profiling Cryptography Developers*". In: (2020).

[18] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. "*Jasmin: High-Assurance and High-Speed Cryptography*". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 1807–1823. DOI: 10.1145/3133956.3134078.

[19] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. "*Verifying Constant-Time Implementations*". In: *USENIX Security 2016: 25th USENIX Security Symposium*. Ed. by Thorsten Holz and Stefan Savage. Austin, TX, USA: USENIX Association, Aug. 2016, pp. 53–70.

[20] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Benjamin Grégoire, Adrien Koutsos, Vincent Laporte, Tiago Oliveira, and Pierre-Yves Strub. "*The Last Mile: High-Assurance and High-Speed Cryptographic Implementations*". In: *2020 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2020, pp. 965–982. DOI: 10.1109/SP40000.2020.00028.

[21] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. "*Formal verification of side-channel countermeasures using self-composition*". In: *Sci. Comput. Program.* 78.7 (2013), pp. 796–812. DOI: 10.1016/j.scico.2011.10.008. URL: https://doi.org/10.1016/j.scico.2011.10.008.

[22] Kemal Altinkemer, Jackie Rees, and Sanjay Sridhar. "*Vulnerabilities and patches of open source software: an empirical study*". In: *Journal of Information System Security* 4.2 (2008), pp. 3–25.

[23] Ron Amadeo. *Linux gives up on 6-year LTS kernels, says they're too much work*. https://arstechnica.com/gadgets/2023/09/linux-gives-up-on-6-year-lts-thats-fine-for-pcs-bad-for-android/. 2023.

[24] Paschal C. Amusuo, Kyle A. Robinson, Santiago Torres-Arias, Laurent Simon, and James C. Davis. *Preventing Supply Chain Vulnerabilities in Java with a Fine-Grained Permission Manager*. 2023. arXiv: 2310.14117 [cs.CR].

[25] P. Anbalagan and M. Vouk. "*Towards a Unifying Approach in Understanding Security Problems*". In: *Proceedings 20th International Symposium on Software Reliability Engineering (ISSRE'09*. 2009, pp. 136–145.

[26] Chris Aniszczyk. *8 ways your company can support and sustain open source*. https://opensource.com/article/19/4/ways-support-sustain-open-source. 2019.

[27] Gábor Antal, Márton Keleti, and Péter Hegedűns. "*Exploring the Security Awareness of the Python and JavaScript Open Source Communities*". In: *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*. 2020, pp. 16–20.

[28] Maria Antikainen, Timo Aaltonen, and Jaani Väisänen. "*The role of trust in OSS communities — Case Linux Kernel community*". In: *Open Source Development, Adoption and Innovation*. Ed. by Joseph Feller, Brian Fitzgerald, Walt Scacchi, and Alberto Sillitti. Boston, MA: Springer US, 2007, pp. 223–228. ISBN: 978-0-387-72486-7.

[29] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. "*Decomposition instead of self-composition for proving the absence of timing channels*". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*. Ed. by Albert Cohen and Martin T. Vechev. ACM, 2017, pp. 362–375. DOI: 10.1145/3062341.3062378. URL: https://doi.org/10.1145/3062341.3062378.

[30] Konstantinos Athanasiou, Byron Cook, Michael Emmi, Colm MacCárthaigh, Daniel Schwartz-Narbonne, and Serdar Tasiran. "*SideTrail: Verifying Time-Balancing of Cryptosystems*". In: *Verified Software. Theories, Tools, and Experiments - 10th International Conference, VSTTE 2018, Oxford, UK, July 18-19, 2018, Revised Selected Papers*. Ed. by Ruzica Piskac and Philipp Rümmer. Vol. 11294. LNCS. Springer, 2018, pp. 215–228. DOI: 10.1007/978-3-030-03592-1_12. URL: https://doi.org/10.1007/978-3-030-03592-1_12.

[31] Melissa Azouaoui, Davide Bellizia, Ileana Buhan, Nicolas Debande, Sébastien Duval, Christophe Giraud, Éliane Jaulmes, François Koeune, Elisabeth Oswald, François-Xavier Standaert, and Carolyn Whitnall. "*A Systematic Appraisal of Side Channel Evaluation Strategies*". In: *Security Standardisation Research - 6th International Conference, SSR 2020, London, UK, November 30 - December 1, 2020, Proceedings*. Ed. by Thyla van der Merwe, Chris J. Mitchell, and Maryam Mehrnezhad. Vol. 12529. Lecture Notes in Computer Science. Springer, 2020, pp. 46–66. DOI: 10.1007/978-3-030-64357-7_3. URL: https://doi.org/10.1007/978-3-030-64357-7_3.

[32] Salem S. Bahamdain. "*Open Source Software (OSS) Quality Assurance: A Survey Paper*". In: *Procedia Computer Science* 56 (2015). The 10th International Conference on Future Networks and Communications (FNC 2015) / The 12th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2015) Affiliated Workshops, pp. 459–464. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.07.236. URL: https://www.sciencedirect.com/science/article/pii/S1877050915017172.

[33] Jia-Ju Bai, Julia Lawall, Qiu-Liang Chen, and Shi-Min Hu. "*Effective static analysis of concurrency use-after-free bugs in Linux device drivers*". In: *Proceedings of the 2019 USENIX Annual Technical Conference (ATC'19)*. 2019, pp. 255–268.

[34] Wei Bai, Moses Namara, Yichen Qian, Patrick Gage Kelley, Michelle L Mazurek, and Doowon Kim. "*An inconvenient trust: User attitudes toward security and usability tradeoffs for key-directory encryption systems*". In: *Proceedings of the 12th Symposium on Usable Privacy and Security (SOUPS'16)*. 2016, pp. 113–130.

[35] Sogol Balali, Umayal Annamalai, Hema Susmita Padala, Bianca Trinkenreich, Marco A. Gerosa, Igor Steinmacher, and Anita Sarma. "*Recommending Tasks to Newcomers in OSS Projects: How Do Mentors Handle It?*" In: *Proceedings of the 16th International Symposium on Open Collaboration (OpenSym'20)*. 2020.

[36] Musard Balliu, Mads Dam, and Gurvan Le Guernic. "*ENCoVer: Symbolic Exploration for Information Flow Security*". In: *CSF 2012: IEEE 25th Computer Security Foundations Symposium*. Ed. by Steve Zdancewic and Véronique Cortier. Cambridge, MA, USA: IEEE Computer Society Press, June 2012, pp. 30–44. DOI: 10.1109/CSF.2012.24.

[37] Qinkun Bao, Zihao Wang, James R. Larus, and Dinghao Wu. "*Abacus: A Tool for Precise Side-Channel Analysis*". In: *43rd IEEE/ACM International Conference on Software Engineering: Companion Proceedings, ICSE Companion 2021*. IEEE, 2021, pp. 238–239. DOI: 10.1109/ICSE-Companion52605.2021.00110. URL: https://doi.org/10.1109/ICSE-Companion52605.2021.00110.

[38] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. "*SoK: Computer-Aided Cryptography*". In: *2021 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2021, pp. 777–795. DOI: 10.1109/SP40001.2021.00008.

[39] Manuel Barbosa and Peter Schwabe. *Kyber terminates*. Cryptology ePrint Archive, Paper 2023/708. https://eprint.iacr.org/2023/708. 2023. URL: https://eprint.iacr.org/2023/708.

[40] Rob Barrett, Eser Kandogan, Paul P Maglio, Eben M Haber, Leila A Takayama, and Madhu Prabaker. "*Field studies of computer system administrators: analysis of system management tools and practices*". In: *Proceedings of the 2004 ACM conference on Computer Supported Cooperative Work*. 2004, pp. 388–395.

[41] Gilles Barthe, Sonia Belaïd, Thomas Espitau, Pierre-Alain Fouque, Mélissa Rossi, and Mehdi Tibouchi. "*GALACTICS: Gaussian Sampling for Lattice-Based Constant- Time Implementation of Cryptographic Signatures, Revisited*". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019*. ACM, 2019, pp. 2147–2164. URL: https://doi.org/10.1145/3319535.3363223.

[42] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, Carlos Daniel Luna, and David Pichardie. "*System-level Non-interference for Constant-time Cryptography*". In: *ACM CCS 2014: 21st Conference on Computer and Communications Security*. Ed. by Gail-Joon Ahn, Moti Yung, and Ninghui Li. Scottsdale, AZ, USA: ACM Press, Nov. 2014, pp. 1267–1279. DOI: 10.1145/2660267.2660283.

[43] Gilles Barthe, Marcel Böhme, Sunjay Cauligi, Chitchanok Chuengsatiansup, Daniel Genkin, Marco Guarnieri, David Mateos Romero, Peter Schwabe, David Wu, and Yuval Yarom. *Testing side-channel security of cryptographic implementations against future microarchitectures*. arXiv preprint 2402.00641. https://arxiv.org/abs/2402.00641. 2024.

[44] Lujo Bauer, Lorrie Faith Cranor, Robert W Reeder, Michael K Reiter, and Kami Vaniea. "*Real life challenges in access-control management*". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 2009, pp. 899–908.

[45] Steven M. Bellovin. http://catless.ncl.ac.uk/Risks/25.71.html#subj19. 2009.

[46] Yoav Benjamini and Yosef Hochberg. "*Controlling the false discovery rate: a practical and powerful approach to multiple testing*". In: *Journal of the Royal statistical society: series B (Methodological)* 57.1 (1995), pp. 289–300.

[47] Daniel J. Bernstein. *Cache-timing attacks on AES*. http://cr.yp.to/antiforgery/cachetiming-20050414.pdf. 2005.

[48] Daniel J. Bernstein. "*Curve25519: New Diffie-Hellman Speed Records*". In: *PKC 2006: 9th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin. Vol. 3958. Lecture Notes in Computer Science. New York, NY, USA: Springer, Heidelberg, Germany, Apr. 2006, pp. 207–228. DOI: 10.1007/11745853_14.

[49] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. "*High-Speed High-Security Signatures*". In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Nara, Japan: Springer, Heidelberg, Germany, Sept. 2011, pp. 124–142. DOI: 10.1007/978-3-642-23951-9_9.

[50] Antonia Bertolino. "*Software testing research: Achievements, challenges, dreams*". In: *Future of Software Engineering (FOSE'07)*. IEEE. 2007, pp. 85–103.

[51] Vieri del Bianco, Luigi Lavazza, Sandro Morasca, Davide Taibi, and Davide Tosi. "*An Investigation of the Users' Perception of OSS Quality*". In: *Open Source Software: New Horizons*. Ed. by Pär Ågerfalk, Cornelia Boldyreff, Jesús M. González-Barahona, Gregory R. Madey, and John Noll. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 15–28. ISBN: 978-3-642-13244-5.

[52] Lukas Bieringer, Kathrin Grosse, Michael Backes, Battista Biggio, and Katharina Krombholz. "*Industrial practitioners' mental models of adversarial machine learning*". In: *Proceedings of the 18th Symposium on Usable Privacy and Security (SOUPS'22)*. Aug. 2022, pp. 97–116.

[53] Sandrine Blazy, David Pichardie, and Alix Trieu. "*Verifying Constant-Time Implementations by Abstract Interpretation*". In: *ESORICS 2017: 22nd European Symposium on Research in Computer Security, Part I*. Ed. by Simon N. Foley, Dieter Gollmann, and Einar Snekkenes. Vol. 10492. Lecture Notes in Computer Science. Oslo, Norway: Springer, Heidelberg, Germany, Sept. 2017, pp. 260–277. DOI: 10.1007/978-3-319-66402-6_16.

[54] Kelly Blincoe, Francis Harrison, and Daniela Damian. "*Ecosystems in GitHub and a Method for Ecosystem Identification Using Reference Coupling*". In: *Proceedings of the 12th Working Conference on Mining Software Repositories* (*MSR'15*). 2015, pp. 202–207.

[55] Nele Borgert, Jennifer Friedauer, Imke Böse, and Malte Elson. "*The Study of Cybersecurity Self-Efficacy: A Systematic Literature Review of Methodology*". In.

[56] Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. "*Constantine: Automatic Side-Channel Resistance Using Efficient Control and Data Flow Linearization*". In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Ed. by Giovanni Vigna and Elaine Shi. Virtual Event, Republic of Korea: ACM Press, Nov. 2021, pp. 715–733. DOI: 10.1145/3460120.3484583.

[57] Joppe W. Bos, Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, John M. Schanck, Peter Schwabe, Gregor Seiler, and Damien Stehlé. "*CRYSTALS-Kyber: A CCA-Secure Module-Lattice-Based KEM*". In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*. IEEE, 2018, pp. 353–367. DOI: 10.1109/EuroSP.2018.00032.

[58] Amiangshu Bosu, Jeffrey C Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. "*Identifying the characteristics of vulnerable code changes: An empirical study*". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, pp. 257–268.

[59] Amiangshu Bosu and Jeffrey C. Carver. "*Impact of Developer Reputation on Code Review Outcomes in OSS Projects: An Empirical Investigation*". In: *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (*ESEM'14*). 2014.

[60] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. "*When Are OSS Developers More Likely to Introduce Vulnerable Code Changes? A Case Study*". In: *Open Source Software: Mobile Open Source Technologies*. Springer Berlin Heidelberg, 2014, pp. 234–236.

[61] 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.

[62] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. "*Dragonblood is Still Leaking: Practical Cache-based Side-Channel in the Wild*". In: *ACSAC '20: Annual Computer Security Applications Conference*. ACM, 2020, pp. 291–303. URL: https://doi.org/10.1145/3427228.3427295.

[63] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. "*PARASITE: PAssword Recovery Attack against Srp Implementations in ThE wild*". In: *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security*. Ed. by Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi. ACM, 2021, pp. 2497–2512. DOI: 10.1145/3460120.3484563.

[64] Tohar Braun and Lidor Ben Shitrit. *Dependency Confusion Supply Chain Attacks: 49% of Organizations Are Vulnerable*. https://orca.security/resources/blog/dependency-confusion-supply-chain-attacks/. 2023.

[65] Virginia Braun and Victoria Clarke. "*Using thematic analysis in psychology*". In: *Qualitative research in psychology* 3.2 (2006), pp. 77–101.

[66] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Pasareanu. "*Symbolic path cost analysis for side-channel detection*". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by Frank Tip and Eric Bodden. ACM, 2018, pp. 27–37. DOI: 10.1145/3213846.3213867. URL: https://doi.org/10.1145/3213846.3213867.

[67] Ernie Brickell. *Technologies to Improve Platform Security*. Invited talk at CHES 2011. 2011. URL: https://www.iacr.org/workshops/ches/ches2011/presentations/Invited%5C%201/CHES2011_Invited_1.pdf.

[68] John Brooke. "*SUS: A quick and dirty usability scale*". In: *Usability Eval. Ind.* 189 (Nov. 1995).

[69] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut T. Kandemir. "*CaSym: Cache Aware Symbolic Execution for Side Channel Detection and Mitigation*". In: *2019 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2019, pp. 505–521. DOI: 10.1109/SP.2019.00022.

[70] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. "*Flush, Gauss, and Reload - A Cache Attack on the BLISS Lattice-Based Signature Scheme*". In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2016, pp. 323–345. DOI: 10.1007/978-3-662-53140-2_16.

[71] Billy Bob Brumley and Nicola Tuveri. "*Remote Timing Attacks Are Still Practical*". In: *ESORICS 2011: 16th European Symposium on Research in Computer Security*. Ed. by Vijay Atluri and Claudia Díaz. Vol. 6879. Lecture Notes in Computer Science. Leuven, Belgium: Springer, Heidelberg, Germany, Sept. 2011, pp. 355–371. DOI: 10.1007/978-3-642-23822-2_20.

[72] David Brumley and Dan Boneh. "*Remote Timing Attacks Are Practical*". In: *USENIX Security 2003: 12th USENIX Security Symposium*. Washington, DC, USA: USENIX Association, Aug. 2003.

[73] David Brumley and Dan Boneh. "*Remote Timing Attacks are Practical*". In: *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*. ACM, 2003.

[74] Sven Bugiel, Lucas Vincenzo Davi, and Steffen Schulz. "*Scalable trust establishment with software reputation*". In: *Proceedings of the sixth ACM workshop on Scalable trusted computing*. 2011, pp. 15–24.

[75] David Burke, Joe Hurd, John Launchbury, and Aaron Tomb. "*Trust Relationship Modeling for Software Assurance*". In: *Proceedings of the 7th International Workshop on Formal Aspects of Security & Trust*. 2010.

[76] Simon Butler, Jonas Gamalielsson, Björn Lundell, Christoffer Brax, Anders Mattsson, Tomas Gustavsson, Jonas Feist, Bengt Kvarnström, and Erik Lönroth. "*On business adoption and use of reproducible builds for open and closed source software*". In: *Software Quality Journal* (Nov. 2022).

[77] Rina Diane Caballar. *The Move to Memory-Safe Programming*. https://spectrum.ieee.org/memory-safe-programming-languages. 2023.

[78] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. "*A Systematic Evaluation of Transient Execution Attacks and Defenses*". In: *USENIX Security 2019: 28th USENIX Security Symposium*. Ed. by Nadia Heninger and Patrick Traynor. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 249–266.

[79] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. "*Who is Going to Mentor Newcomers in Open Source Projects?*" In: *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering* (*FSE'12*). 2012.

[80] Brice Canvel, Alain P. Hiltgen, Serge Vaudenay, and Martin Vuagnoux. "*Password Interception in a SSL/TLS Channel*". In: *Advances in Cryptology – CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 2003, pp. 583–599. DOI: 10.1007/978-3-540-45146-4_34.

[81] Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. "*Developer Onboarding in GitHub: The Role of Prior Social Links and Language Experience*". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*. 2015, pp. 817–828.

[82] Sunjay Cauligi, Craig Disselkoen, Klaus von Gleissenthall, Dean M. Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. "*Constant-time foundations for the new spectre era*". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*. Ed. by Alastair F. Donaldson and Emina Torlak. ACM, 2020, pp. 913–926. DOI: 10.1145/3385412.3385970. URL: https://doi.org/10.1145/3385412.3385970.

[83] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. "*SoK: Practical Foundations for Software Spectre Defenses*". In: *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2022, pp. 666–680. DOI: 10.1109/SP46214.2022.9833707.

[84] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. "*SoK: Practical Foundations for Spectre Defenses*". In: *CoRR* abs/2105.05801 (2021). arXiv: 2105.05801. URL: https://arxiv.org/abs/2105.05801.

[85] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. "*FaCT: a DSL for timing-sensitive computation*". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*. Ed. by Kathryn S. McKinley and Kathleen Fisher. ACM, 2019, pp. 174–189. DOI: 10.1145/3314221.3314605. URL: https://doi.org/10.1145/3314221.3314605.

[86] Scott Chacon and Ben Straub. https://git-scm.com/book/en/v2/Git-Basics-Tagging. 2014.

[87] Ramaswamy Chandramouli, Frederick Kautz, and Santiago Torres Arias. *Strategies for the Integration of Software Supply Chain Security in DevSecOps CI/CD pipelines*. Tech. rep. National Institute of Standards and Technology, 2023.

[88] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. "*Quantifying the Information Leakage in Cache Attacks via Symbolic Execution*". In: *ACM Trans. Embed. Comput. Syst.* 18.1 (Jan. 2019). ISSN: 1539-9087. DOI: 10.1145/3288758.

[89] Sudipta Chattopadhyay and Abhik Roychoudhury. "*Symbolic Verification of Cache Side-Channel Freedom*". In: *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 37.11 (2018), pp. 2812–2823. DOI: 10.1109/TCAD.2018.2858402. URL: https://doi.org/10.1109/TCAD.2018.2858402.

[90] Christine Chen, Nicola Dell, and Franziska Roesner. "*Computer security and privacy in the interactions between victim service providers and human trafficking survivors*". In: *Proceedings of the 28th USENIX Security Symposium (Sec'19)*. 2019, pp. 89–104.

[91] Jia Chen, Yu Feng, and Isil Dillig. "*Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic*". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 875–890. DOI: 10.1145/3133956.3134058.

[92] Maria Christakis and Christian Bird. "*What developers want and need from program analysis: an empirical study*". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016*. Ed. by David Lo, Sven Apel, and Sarfraz Khurshid. ACM, 2016, pp. 332–343. DOI: 10.1145/2970276.2970347.

[93] Ching-Chi Chuang, Luís Cruz, Robbert van Dalen, Vladimir Mikovski, and Arie van Deursen. *"Removing dependencies from large software projects: are you really sure?"* In: *2022 IEEE 22nd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 2022, pp. 105–115. DOI: 10.1109/SCAM55253.2022.00017.

[94] Alonzo Church. "*A note on the Entscheidungsproblem*". In: *The Journal of Symbolic Logic* 1.1 (1936), pp. 40–41. DOI: 10.2307/2269326.

[95] CISA. *Securing the Software Supply Chain: Recommended Practices for Developers*. https://www.cisa.gov/sites/default/files/publications/ESF_SECURING_THE_SOFTWARE_SUPPLY_CHAIN_DEVELOPERS.PDF. 2022.

[96] CISA. *Software Bill of Materials (SBOM)*. https://www.cisa.gov/sbom.

[97] Jailton Coelho and Marco Tulio Valente. "*Why Modern Open Source Projects Fail*". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017.

[98] Gabriella Coleman. *The Anthropology of Hackers*. https://www.theatlantic.com/technology/archive/2010/09/the-anthropology-of-hackers/63308/. 2010.

[99] William Jay Conover. *Practical nonparametric statistics*. Vol. 350. John Wiley & Sons, 1998.

[100] Kattiana Constantino, Mauricio Souza, Shurui Zhou, Eduardo Figueiredo, and Christian Kästner. "*Perceptions of open-source software developers on collaborations: An interview and survey study*". In: *Journal of Software: Evolution and Process* (2021), e2393.

[101] Zara Cooper. *Getting started with contributing to open source*. https://stackoverflow.blog/2020/08/03/getting-started-with-contributing-to-open-source/. 2020.

[102] Ludovic Courtès. "*Code Staging in GNU Guix*". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17)*. 2017, pp. 41–48.

[103] Ludovic Courtès. "*Functional Package Management with Guix*". In: *European Lisp Symposium*. Madrid, Spain, June 2013. URL: https://hal.inria.fr/hal-00824004.

[104] Ludovic Courtès and Ricardo Wurmus. "*Reproducible and User-Controlled Software Environments in HPC with Guix*". In: *2nd International Workshop on Reproducibility in Parallel Computing (RepPar)*. Vienne, Austria, Aug. 2015. URL: https://hal.inria.fr/hal-01161771.

[105] Patrick Cousot and Radhia Cousot. "*Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. Los Angeles, California: Association for Computing Machinery, 1977, pp. 238–252. ISBN: 9781450373500. DOI: 10.1145/512950.512973. URL: https://doi.org/10.1145/512950.512973.

[106] Patrick Cousot and Radhia Cousot. "*Systematic Design of Program Analysis Frameworks*". In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '79. San Antonio, Texas: Association for Computing Machinery, 1979, pp. 269–282. ISBN: 9781450373579. DOI: 10.1145/567752.567778. URL: https://doi.org/10.1145/567752.567778.

[107] Russ Cox. *Our Software Dependency Problem*. https://research.swtch.com/deps. 2019.

[108] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. "*Free/Libre Open-Source Software Development: What We Know and What We Do Not Know*". In: *ACM Comput. Surv.* 44.2 (2008). ISSN: 0360-0300. DOI: 10.1145/2089125.2089127. URL: https://doi.org/10.1145/2089125.2089127.

[109] Pascal Cuoq. *tis-ct*. URL: http://web.archive.org/web/20200810074547/http://trust-in-soft.com/tis-ct/.

[110] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. *"Social Coding in GitHub: Transparency and Collaboration in an Open Software Repository"*. In: *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work (CSCW'12)*. 2012, pp. 1277–1286.

[111] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. *"Binsec/Rel: Efficient Relational Symbolic Execution for Constant-Time at Binary-Level"*. In: *2020 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2020, pp. 1021–1038. DOI: 10.1109/SP40000.2020.00074.

[112] Daniel De Almeida Braga, Pierre-Alain Fouque, and Mohamed Sabt. *"The Long and Winding Path to Secure Implementation of GlobalPlatform SCP10"*. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.3 (2020). https://tches.iacr.org/index.php/TCHES/article/view/8588, pp. 196–218. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i3.196-218.

[113] Heini Bergsson Debes, Thanassis Giannetsos, and Ioannis Krontiris. *BLINDTRUST: Oblivious Remote Attestation for Secure Service Function Chains*. 2021. arXiv: 2107.05054 [cs.CR].

[114] *Debian Developers' Corner / How to join Debian / Step 2: Identification*. https://www.debian.org/devel/join/nm-step2.

[115] Pantazis Deligiannis, Alastair F Donaldson, and Zvonimir Rakamaric. *"Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers"*. In: *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. 2015, pp. 166–177.

[116] Valgrind Developers. https://valgrind.org/docs/manual/mc-manual.html.

[117] devrandom. *Gitian: a secure software distribution method*. https://github.com/devrandom/gitian-builder. 2011.

[118] Whitfield Diffie and Martin E. Hellman. *"New Directions in Cryptography"*. In: *IEEE Transactions on Information Theory* 22.6 (1976), pp. 644–654. DOI: 10.1109/TIT.1976.1055638.

[119] *diffoscope In-depth comparison of files, archives, and directories.* https://diffoscope.org/. 2014. URL: https://diffoscope.org/.

[120] Trung T Dinh-Trong and James M Bieman. *"The FreeBSD project: A replication case study of open source development"*. In: *IEEE Transactions on Software Engineering* 31.6 (2005), pp. 481–494.

[121] Lisa Nguyen Quang Do, James R. Wright, and Karim Ali. *"Why Do Software Developers Use Static Analysis Tools? A User-Centered Study of Developer Needs and Motivations"*. In: *IEEE Trans. Software Eng.* 48.3 (2022), pp. 835–847. DOI: 10.1109/TSE.2020.3004525.

[122] Katherine Doherty, Liz Capo McCormick, and Alexandra Harris. *Cyber Attack Forces World's Biggest Bank to Trade via USB Stick*. https://time.com/6333716/china-icbc-bank-hack-usb-stick-trading/.

[123] Eelco Dolstra, Andres Löh, and Nicolas Pierron. *"NixOS: A purely functional Linux distribution"*. In: *Journal of Functional Programming* 20.5-6 (2010), pp. 577–615.

[124] James Dominic, Jada Houser, Igor Steinmacher, Charles Ritter, and Paige Rodeghero. *"Conversational Bot for Newcomers Onboarding to Open Source Projects"*. In: *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops (ICSEW'20)*. 2020, pp. 46–50.

[125] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. *"CacheAudit: A Tool for the Static Analysis of Cache Side Channels"*. In: *USENIX Security 2013: 22nd USENIX Security Symposium*. Ed. by Samuel T. King. Washington, DC, USA: USENIX Association, Aug. 2013, pp. 431–446.

[126] Goran Doychev and Boris Köpf. *"Rigorous analysis of software countermeasures against cache attacks"*. In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017*. Barcelona, Spain: ACM, June 2017, pp. 406–421. DOI: 10.1145/3062341.3062388.

[127] Léo Ducas, Eike Kiltz, Tancrède Lepoint, Vadim Lyubashevsky, Peter Schwabe, Gregor Seiler, and Damien Stehlé. "*CRYSTALS-Dilithium: A Lattice-Based Digital Signature Scheme*". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.1 (2018). https://tches.iacr.org/index.php/TCHES/article/view/839, pp. 238–268. ISSN: 2569-2925. DOI: 10.13154/tches.v2018.i1.238-268.

[128] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, and J. Alex Halderman. "*The Matter of Heartbleed*". In: *Proc. 2014 Internet Measurement Conference (IMC'14)*. ACM, 2014.

[129] *eBACS: ECRYPT Benchmarking of Cryptographic Systems*. accessed November 5, 2009. URL: https://bench.cr.yp.to.

[130] Nigel Edwards and Liqun Chen. "*An Historical Examination of Open Source Releases and Their Vulnerabilities*". In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS'12)*. 2012, pp. 183–194.

[131] William Enck and Laurie Williams. "*Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations*". In: *IEEE Security & Privacy* 20.2 (2022), pp. 96–100. DOI: 10.1109/MSEC.2022.3142338.

[132] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. "*Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises*". In: *2019 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2019, pp. 1202–1219. DOI: 10.1109/SP.2019.00005.

[133] *ESF Partners, NSA, and CISA Release Software Supply Chain Guidance for Suppliers*. https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3204427/esf-partners-nsa-and-cisa-release-software-supply-chain-guidance-for-suppliers/. 2022. URL: https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3204427/esf-partners-nsa-and-cisa-release-software-supply-chain-guidance-for-suppliers/.

[134] Douglas Everson, Long Cheng, and Zhenkai Zhang. "*Log4shell: Redefining the web attack surface*". In: *Workshop on Measurements, Attacks, and Defenses for the Web (MADWeb) 2022*. 2022.

[135] *Executive Order on America's Supply Chains*. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/02/24/executive-order-on-americas-supply-chains/. 2022. URL: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/02/24/executive-order-on-americas-supply-chains/.

[136] *Executive Order on Improving the Nation's Cybersecurity*. https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/. 2021. URL: https://www.whitehouse.gov/briefing-room/presidential-actions/2021/05/12/executive-order-on-improving-the-nations-cybersecurity/.

[137] Zixuan Feng, Mariam Guizani, and Anita Sarma. *The State of Diversity and Inclusion in the ASF Community: A Pulse Check*. https://news.apache.org/foundation/entry/the-state-of-diversity-and-inclusion-in-the-asf-community-a-pulse-check. 2023.

[138] Konstantin Fischer, Ivana Trummova, Phillip Gajland, Yasemin Acar, Sascha Fahl, and Angela Sasse. https://saschafahl.de/static/paper/cryptoadoption2024ext.pdf. 2024.

[139] Michael Flanders, Reshabh K Sharma, Alexandra E. Michael, Dan Grossman, and David Kohlbrenner. "*Avoiding Instruction-Centric Microarchitectural Timing Channels Via Binary-Code Transformations*". In: *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2024, to appear.

[140] Denae Ford, Mahnaz Behroozi, Alexander Serebrenik, and Chris Parnin. "*Beyond the Code Itself: How Programmers Really Look at Pull Requests*". In: *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Society (ICSE-SEIS'19)*. 2019, pp. 51–60.

[141] Marcel Fourné, Jan Jancar, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "*'These results must be false': A usability evaluation of constant-time analysis tools*". In: *33th USENIX Security Symposium (USENIX Security 2024)*. USENIX Association, 2024.

[142] Marcel Fourné, Dominique Petersen, and Norbert Pohlmann. "*Attack-test and verification systems, steps towards verifiable anomaly detection*". In: *INFORMATIK 2013 – Informatik angepasst an Mensch, Organisation und Umwelt*. Gesellschaft für Informatik eV, 2013, pp. 2213–2224. ISBN: 978-3-88579-614-5.

[143] Marcel Fourné, Kevin Stegemann, Dominique Petersen, and Norbert Pohlmann. "*Aggregation of Network Protocol Data Near Its Source*". In: *Information and Communication Technology: Second IFIP TC5/8 International Conference, ICT-EurAsia 2014, Bali, Indonesia, April 14-17, 2014. Proceedings 2*. Springer. 2014, pp. 482–491. ISBN: 978-3-642-55031-7. DOI: 10.1007/978-3-642-55032-4_49.

[144] Marcel Fourné, Dominik Wermke, William Enck, Sascha Fahl, and Yasemin Acar. "*It's like flossing your teeth: On the Importance and Challenges of Reproducible Builds for Software Supply Chain Security*". In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023, pp. 1527–1544. DOI: 10.1109/SP46215.2023.10179320.

[145] Marcel Fourné, Dominik Wermke, Sascha Fahl, and Yasemin Acar. "*A Viewpoint on Human Factors in Software Supply Chain Security: A Research Agenda*". In: *IEEE Security & Privacy* 21.6 (2023), pp. 59–63. DOI: 10.1109/MSEC.2023.3316569.

[146] Kelsey R. Fulton, Anna Chan, Daniel Votipka, Michael Hicks, and Michelle L. Mazurek. "*Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study*". In: *Proceedings of the 17th Symposium on Usable Privacy and Security (SOUPS 2021)*. Aug. 2021, pp. 597–616.

[147] Patricia I Fusch and Lawrence R Ness. "*Are we there yet? Data saturation in qualitative research*". In: (2015).

[148] Kevin Gallagher, Sameer Patil, and Nasir Memon. "*New me: Understanding expert and non-expert perceptions and usage of the Tor anonymity network*". In: *Proceedings of the 13th Symposium on Usable Privacy and Security (SOUPS'17)*. 2017, pp. 385–398.

[149] Cesar Pereida García and Billy Bob Brumley. "*Constant-Time Callees with Variable-Time Callers*". In: *USENIX Security 2017: 26th USENIX Security Symposium*. Ed. by Engin Kirda and Thomas Ristenpart. Vancouver, BC, Canada: USENIX Association, Aug. 2017, pp. 83–98.

[150] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "*"Make Sure DSA Signing Exponentiations Really are Constant-Time"*". In: *ACM CCS 2016: 23rd Conference on Computer and Communications Security*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. Vienna, Austria: ACM Press, Oct. 2016, pp. 1639–1650. DOI: 10.1145/2976749.2978420.

[151] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. "*Certified Side Channels*". In: *USENIX Security 2020: 29th USENIX Security Symposium*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, Aug. 2020, pp. 2021–2038.

[152] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. "*A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries*". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2023, pp. 1690–1704. DOI: 10.1145/3576915.3623112. URL: https://doi.org/10.1145/3576915.3623112.

[153] Daniel Genkin, Adi Shamir, and Eran Tromer. "*Acoustic Cryptanalysis*". In: *Journal of Cryptology* 30.2 (Apr. 2017), pp. 392–443. DOI: 10.1007/s00145-015-9224-2.

[154] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. "*VulinOSS: A Dataset of Security Vulnerabilities in Open-Source Systems*". In: *Proceedings of the 15th International Conference on Mining Software Repositories*. MSR '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 18–21. ISBN: 9781450357166.

[155] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "*DART: Directed automated random testing*". In: *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. 2005, pp. 213–223.

[156] Kurt Gödel. *Über die Vollständigkeit des Logikkalküls*. ger. 1929.

[157] Kurt Gödel. "*Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I*". In: *Monatshefte für Mathematik und Physik* 38.1 (Dec. 1931), pp. 173–198. ISSN: 1436-5081. DOI: 10.1007/BF01700692. URL: https://doi.org/10.1007/BF01700692.

[158] 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: *CHI '20: Conference on Human Factors in Computing Systems*. ACM, 2020, pp. 1–13. DOI: 10.1145/3313831.3376142.

[159] Georgios Gousios, Martin Pinzger, and Arie van Deursen. "*An Exploratory Study of the Pull-Based Software Development Model*". In: *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. 2014, pp. 345–355.

[160] Georgios Gousios and Diomidis Spinellis. "*GHTorrent: GitHub's Data from a Firehose*". In: *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories (MSR'12)*. 2012, pp. 12–21.

[161] Georgios Gousios, Bogdan Vasilescu, Alexander Serebrenik, and Andy Zaidman. "*Lean GHTorrent: GitHub Data on Demand*". In: *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. 2014, pp. 384–387.

[162] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. "*ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures*". In: *ISOC Network and Distributed System Security Symposium – NDSS 2020*. San Diego, CA, USA: The Internet Society, Feb. 2020.

[163] Iaroslav Gridin, Cesar Pereida García, Nicola Tuveri, and Billy Bob Brumley. "*Triggerflow: Regression Testing by Advanced Execution Path Inspection*". In: *Detection of Intrusions and Malware, and Vulnerability Assessment - 16th International Conference, DIMVA 2019*. Ed. by Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren. Vol. 11543. LNCS. Springer, 2019, pp. 330–350. DOI: 10.1007/978-3-030-22038-9_16.

[164] Arne-Kristian Groven, Kirsten Haaland, Ruediger Glott, and Anna Tannenberg. "*Security Measurements within the Framework of Quality Assessment Models for Free/Libre Open Source Software*". In: *Proceedings of the 4th European Conference on Software Architecture (ECSA'10): Companion Volume*. 2010, pp. 229–235.

[165] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "*Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches*". In: *USENIX Security 2015: 24th USENIX Security Symposium*. Ed. by Jaeyeon Jung and Thorsten Holz. Washington, DC, USA: USENIX Association, Aug. 2015, pp. 897–912.

[166] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. "*Spectector: Principled Detection of Speculative Information Flows*". In: *2020 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2020, pp. 1–19. DOI: 10.1109/SP40000.2020.00011.

[167] Qian Guo, Thomas Johansson, and Alexander Nilsson. "*A Key-Recovery Timing Attack on Post-quantum Primitives Using the Fujisaki-Okamoto Transformation and Its Application on FrodoKEM*". In: *Advances in Cryptology – CRYPTO 2020*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12171. LNCS. Springer, 2020, pp. 359–386. URL: https://eprint.iacr.org/2020/743.

[168] Wenbo Guo, Zhengzi Xu, Chengwei Liu, Cheng Huang, Yong Fang, and Yang Liu. *"An Empirical Study of Malicious Code In PyPI Ecosystem"*. In: *arXiv preprint arXiv:2309.11021* (2023).

[169] Marco Gutfleisch, Jan H Klemmer, Niklas Busch, Yasemin Acar, M Angela Sasse, and Sascha Fahl. *"How does usable security (not) end up in software products? results from a qualitative interview study"*. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 893–910.

[170] Marco Gutfleisch, Jan H. Klemmer, Niklas Busch, Yasemin Acar, M. Angela Sasse, and Sascha Fahl. *"How Does Usable Security (Not) End Up in Software Products? Results From a Qualitative Interview Study"*. In: *Proc. 43rd IEEE Symposium on Security and Privacy (SP'22)*. IEEE, 2022. DOI: 10.1109/SP46214.2022.9833756. URL: https://publications.teamusec.de/2022-oakland-usec-in-sdps/.

[171] 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, 2018.

[172] 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: *Fourteenth Symposium on Usable Privacy and Security, SOUPS 2018*. USENIX Association, 2018, pp. 357–373. URL: https://www.usenix.org/conference/soups2018/presentation/haney-mindsets.

[173] Christoph Hannebauer and Volker Gruhn. *"Motivation of Newcomers to FLOSS Projects"*. In: *Proceedings of the 12th International Symposium on Open Collaboration (OpenSym'16)*. 2016.

[174] Alexander Hars and Shaosong Ou. *"Working for Free? Motivations for Participating in Open-Source Projects"*. In: *Int. J. Electron. Commerce* 6.3 (Apr. 2002), pp. 25–39. ISSN: 1086-4415.

[175] Greg Hartrell. *Where Did Hacker Culture Come From?* https://www.forbes.com/sites/quora/2017/09/07/where-did-hacker-culture-come-from/. 2017.

[176] Sohaib ul Hassan, Iaroslav Gridin, Ignacio M. Delgado-Lozano, Cesar Pereida García, Jesús-Javier Chi-Domínguez, Alejandro Cabrera Aldaya, and Billy Bob Brumley. *"Déjà Vu: Side-Channel Analysis of Mozilla's NSS"*. In: *ACM CCS 2020: 27th Conference on Computer and Communications Security*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. Virtual Event, USA: ACM Press, Nov. 2020, pp. 1887–1902. DOI: 10.1145/3372297.3421761.

[177] Hideaki Hata, Raula Gaikovina Kula, Takashi Ishio, and Christoph Treude. *"Research Artifact: The Potential of Meta-Maintenance on GitHub"*. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2021, pp. 192–193. DOI: 10.1109/ICSE-Companion52605.2021.00084.

[178] Lile P. Hattori and Michele Lanza. *"On the nature of commits"*. In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering - Workshops*. 2008, pp. 63–71.

[179] Øyvind Hauge, Claudia Ayala, and Reidar Conradi. *"Adoption of open source software in software-intensive organizations – A systematic literature review"*. In: *Information and Software Technology* 52.11 (2010). Special Section on Best Papers PROMISE 2009, pp. 1133–1154. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2010.05.008. URL: https://www.sciencedirect.com/science/article/pii/S0950584910000972.

[180] Mohammadreza Hazhirpasand, Oscar Nierstrasz, Mohammadhossein Shabani, and Mohammad Ghafari. *"Hurdles for Developers in Cryptography"*. In: *CoRR* abs/2108.07141 (2021). arXiv: 2108.07141. URL: https://arxiv.org/abs/2108.07141.

[181] Shaobo He, Michael Emmi, and Gabriela F. Ciocarlie. *"ct-fuzz: Fuzzing for Timing Leaks"*. In: *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 2020, pp. 466–471. DOI: 10.1109/ICST46399.2020.00063. URL: https://doi.org/10.1109/ICST46399.2020.00063.

[182] Cormac Herley and Paul C. van Oorschot. "*SoK: Science, Security and the Elusive Goal of Security as a Scientific Pursuit*". In: *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 2017, pp. 99–120. DOI: 10.1109/SP.2017.38.

[183] Guido Hertel, Sven Niedner, and Stefanie Herrmann. "*Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel*". In: *Research Policy* 32.7 (2003). Open Source Software Development, pp. 1159–1177. ISSN: 0048-7333. DOI: https://doi.org/10.1016/S0048-7333(03)00047-7. URL: https://www.sciencedirect.com/science/article/pii/S0048733303000477.

[184] Robert Heumüller, Sebastian Nielebock, Jacob Krüger, and Frank Ortmeier. "*Publish or perish, but do not forget your software artifacts*". In: *Empir. Softw. Eng.* 25.6 (2020), pp. 4585–4616. DOI: 10.1007/s10664-020-09851-6.

[185] Mar Hicks. *Programmed Inequality: How Britain Discarded Women Technologists and Lost Its Edge in Computing*. MIT Press, 2017. ISBN: 9780262535182.

[186] Jonas Hielscher, Uta Menges, Simon Parkin, Annette Kluge, and M Angela Sasse. ""*Employees Who Don't Accept the Time Security Takes Are Not Aware Enough": The CISO View of Human-Centred Security*". In: *32st USENIX Security Symposium (USENIX Security 23), Boston, MA*. 2023.

[187] Jonas Hielscher, Markus Schöps, Uta Menges, Marco Gutfleisch, Mirko Helbling, and M Angela Sasse. "*Lacking the tools and support to fix friction: results from an interview study with security managers*". In: *Nineteenth Symposium on Usable Privacy and Security (SOUPS 2023)*. 2023, pp. 131–150.

[188] Viet Tung Hoang, David Miller, and Ni Trieu. "*Attacks only Get Better: How to Break FF3 on Large Domains*". In: *Advances in Cryptology – EUROCRYPT 2019, Part II*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11477. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Heidelberg, Germany, May 2019, pp. 85–116. DOI: 10.1007/978-3-030-17656-3_4.

[189] Jana Hofmann, Emanuele Vannacci, Cédric Fournet, Boris Köpf, and Oleksii Oleksenko. "*Speculation at fault: modeling and testing microarchitectural leakage of CPU exceptions*". In: *Proceedings of the 32nd USENIX Conference on Security Symposium*. SEC '23. Anaheim, CA, USA: USENIX Association, 2023. ISBN: 978-1-939133-37-3.

[190] Allen D Householder, Jeff Chrabaszcz, Trent Novelly, David Warren, and Jonathan M Spring. "*Historical analysis of exploit availability timelines*". In: *Proceedings of the 13th USENIX Workshop on Cyber Security Experimentation and Test (CSET 20)*. 2020.

[191] *2017 IEEE Symposium on Security and Privacy*. San Jose, CA, USA: IEEE Computer Society Press, May 2017.

[192] *2019 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2019.

[193] *2020 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2020.

[194] *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2022.

[195] Nasif Imtiaz and Laurie Williams. "*Phantom Artifacts & Code Review Coverage in Dependency Updates*". In: *arXiv preprint arXiv:2206.09422* (2022).

[196] Intel. *pin-based-cec*. https://github.com/intel/pin-based-cec.

[197] Marko Ivanković, Goran Petrović, René Just, and Gordon Fraser. "*Code Coverage at Google*". In: *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'19)*. 2019, pp. 955–963.

[198] Jan Jancar. *The state of tooling for verifying constant-timeness of cryptographic implementations*. 2021. URL: https://neuromancer.sk/article/26.

[199] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. *""They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks*". In: *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2022, pp. 632–649. DOI: 10.1109/SP46214.2022.9833713.

[200] Jan Jancar, Vladimir Sedlacek, Petr Svenda, and Marek Sys. *"Minerva: The curse of ECDSA nonces*". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2020.4 (2020). https://tches.iacr.org/index.php/TCHES/article/view/8684, pp. 281–308. ISSN: 2569-2925. DOI: 10.13154/tches.v2020.i4.281-308.

[201] Ke Jiang, Yuyan Bao, Shuai Wang, Zhibo Liu, and Tianwei Zhang. *"Cache Refinement Type for Side-Channel Detection of Cryptographic Software*". In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 1583–1597. DOI: 10.1145/3548606.3560672.

[202] Wenxin Jiang, Nicholas Synovic, Rohan Sethi, Aryan Indarapu, Matt Hyatt, Taylor R Schorlemmer, George K Thiruvathukal, and James C Davis. *"An empirical study of artifacts and security risks in the pre-trained model supply chain*". In: *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. 2022, pp. 105–114.

[203] Ørjan Johansen. https://esolangs.org/wiki/Malbolge_Unshackled. 2007.

[204] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. *"Why don't software developers use static analysis tools to find bugs?"* In: *35th International Conference on Software Engineering, ICSE 2013*. IEEE, 2013, pp. 672–681. DOI: 10.1109/ICSE.2013.6606613.

[205] Hubert Kario. *"Everlasting ROBOT: The Marvin Attack*". In: *ESORICS 2023 - 28th European Symposium on Research in Computer Security*. Vol. 14346. LNCS. The Hague, The Netherlands: Springer, Sept. 2023, pp. 243–262. DOI: 10.1007/978-3-031-51479-1_13.

[206] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. *"When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015*". In: *CANS 16: 15th International Conference on Cryptology and Network Security*. Ed. by Sara Foresti and Giuseppe Persiano. Vol. 10052. Lecture Notes in Computer Science. Milan, Italy: Springer, Heidelberg, Germany, Nov. 2016, pp. 573–582. DOI: 10.1007/978-3-319-48965-0_36.

[207] Taehun Kim and Youngjoo Shin. *"ThermalBleed: A Practical Thermal Side-Channel Attack*". In: *IEEE Access* 10 (2022), pp. 25718–25731. DOI: 10.1109/ACCESS.2022.3156596.

[208] James C King. *"Symbolic execution and program testing*". In: *Communications of the ACM* 19.7 (1976), pp. 385–394.

[209] Jan H Klemmer, Marco Gutfleisch, Christian Stransky, Yasemin Acar, M Angela Sasse, and Sascha Fahl. *"" Make Them Change it Every Week!": A Qualitative Exploration of Online Developer Advice on Usable and Secure Authentication*". In: *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. 2023, pp. 2740–2754.

[210] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. *"Spectre Attacks: Exploiting Speculative Execution*". In: *2019 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[211] Paul C. Kocher. *"Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems*". In: *Advances in Cryptology – CRYPTO'96*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 1996, pp. 104–113. DOI: 10.1007/3-540-68697-5_9.

[212] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "*Differential Power Analysis*". In: *Advances in Cryptology – CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Heidelberg, Germany, Aug. 1999, pp. 388–397. DOI: 10.1007/3-540-48405-1_25.

[213] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. "*Automatic Quantification of Cache Side-Channels*". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. LNCS. Springer, 2012, pp. 564–580. DOI: 10.1007/978-3-642-31424-7_40. URL: https://doi.org/10.1007/978-3-642-31424-7_40.

[214] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. "*Spectre Returns! Speculation Attacks using the Return Stack Buffer*". In: *12th USENIX Workshop on Offensive Technologies, WOOT 2018*. Baltimore, MD, USA: USENIX Association, Aug. 2018.

[215] Frens Kroeger. "*Trusting organizations: The institutionalization of trust in interorganizational relationships*". In: *Organization* 19.6 (2012), pp. 743–763. DOI: 10.1177/1350508411420900. URL: https://doi.org/10.1177/1350508411420900.

[216] Katharina Krombholz, Wilfried Mayer, Martin Schmiedecker, and Edgar R. Weippl. "*"I Have No Idea What I'm Doing" - On the Usability of Deploying HTTPS*". In: *USENIX Security 2017: 26th USENIX Security Symposium*. Ed. by Engin Kirda and Thomas Ristenpart. Vancouver, BC, Canada: USENIX Association, Aug. 2017, pp. 1339–1356.

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

[218] Stefan Krüger, Michael Reif, Anna-Katharina Wickert, Sarah Nadi, Karim Ali, Eric Bodden, Yasemin Acar, Mira Mezini, and Sascha Fahl. "*Securing Your Crypto-API Usage Through Tool Support - A Usability Study*". In: *IEEE Secure Development Conference, SecDev 2023*. Atlanta, GA, USA: IEEE, Oct. 2023, pp. 14–25. DOI: 10.1109/SECDEV56634.2023.00015.

[219] Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. "*Crysl: An extensible approach to validating the correct usage of cryptographic apis*". In: *IEEE Transactions on Software Engineering* (2019).

[220] Markus G Kuhn and Ross J Anderson. "*Soft tempest: Hidden data transmission using electromagnetic emanations*". In: *Information Hiding: Second International Workshop, IH'98 Portland, Oregon, USA, April 14–17, 1998 Proceedings 2*. Springer. 1998, pp. 124–142.

[221] P. Ladisa, H. Plate, M. Martinez, and O. Barais. "*SoK: Taxonomy of Attacks on Open-Source Software Supply Chains*". In: *Proceedings of the 44th IEEE Symposium on Security and Privacy (S&P'23)*. May 2023, pp. 167–184.

[222] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. "*Risk Explorer for Software Supply Chains: Understanding the Attack Surface of Open-Source Based Software Development*". In: *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. SCORED'22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 35–36. ISBN: 9781450398855. DOI: 10.1145/3560835.3564546. URL: https://doi.org/10.1145/3560835.3564546.

[223] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, Olivier Barais, and Serena Elisa Ponta. "*Towards the Detection of Malicious Java Packages*". In: *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. SCORED'22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 63–72. ISBN: 9781450398855. DOI: 10.1145/3560835.3564548. URL: https://doi.org/10.1145/3560835.3564548.

[224] Piergiorgio Ladisa, Serena Elisa Ponta, Nicola Ronzoni, Matias Martinez, and Olivier Barais. "*On the Feasibility of Cross-Language Detection of Malicious Packages in Npm and PyPI*". In: *Proceedings of the 39th Annual Computer Security Applications Conference*. ACSAC '23. , Austin, TX, USA, Association for Computing Machinery, 2023, pp. 71–82. ISBN: 9798400708862. DOI: 10.1145/3627106.3627138. URL: https://doi.org/10.1145/3627106.3627138.

[225] Piergiorgio Ladisa, Merve Sahin, Serena Elisa Ponta, Marco Rosa, Matias Martinez, and Olivier Barais. "*The Hitchhiker's Guide to Malicious Third-Party Dependencies*". In: *Proceedings of the 2023 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. SCORED '23. , Copenhagen, Denmark, Association for Computing Machinery, 2023, pp. 65–74. ISBN: 9798400702631. DOI: 10.1145/3605770.3625212. URL: https://doi.org/10.1145/3605770.3625212.

[226] Chris Lamb and Ximin Luo. https://reproducible-builds.org/specs/source-date-epoch/. 2015-2017.

[227] Chris Lamb and Stefano Zacchiroli. "*Reproducible Builds: Increasing the Integrity of Software Supply Chains*". In: *IEEE Software* 39.2 (2022), pp. 62–70.

[228] Chris Lamb and Stefano Zacchiroli. "*Reproducible Builds: Increasing the Integrity of Software Supply Chains*". In: *IEEE Software* 39.2 (Mar. 2022), pp. 62–70.

[229] Adam Langley. *ctgrind*. 2010.

[230] Adam Langley. *curve25519-donna*. https://github.com/agl/curve25519-donna. 2008.

[231] Jonathan Lazar, Jinjuan Heidi Feng, and Harry Hochheiser. *Research methods in human-computer interaction*. Morgan Kaufmann, 2017.

[232] Barry M. Leiner, Vinton G. Cerf, David D. Clark, Robert E. Kahn, Leonard Kleinrock, Daniel C. Lynch, Jon Postel, Larry G. Roberts, and Stephen Wolff. *A Brief History of the Internet*. https://www.internetsociety.org/internet/history-internet/brief-history-internet/. 1997.

[233] E. Levy. "*Poisoning the software supply chain*". In: *IEEE Security & Privacy* 1.3 (2003), pp. 70–73. DOI: 10.1109/MSECP.2003.1203227.

[234] Frank Li and Vern Paxson. "*A large-scale empirical study of security patches*". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS'17)*. 2017, pp. 2201–2215.

[235] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. "*CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel*". In: *USENIX Security 2021: 30th USENIX Security Symposium*. Ed. by Michael Bailey and Rachel Greenstadt. USENIX Association, Aug. 2021, pp. 717–732.

[236] Renee Li, Pavitthra Pandurangan, Hana Frluckaj, and Laura Dabbish. "*Code of Conduct Conversations in Open Source Software Projects on Github*". In: *Proc. ACM Hum.-Comput. Interact.* 5.CSCW1 (Apr. 2021).

[237] Wen Li, Na Meng, Li Li, and Haipeng Cai. "*Understanding Language Selection in Multi-Language Software Projects on GitHub*". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2021, pp. 256–257.

[238] Morten Linderud. "*Reproducible Builds: Break a log, good things come in trees*". In: (2019). Master Thesis.

[239] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "*Meltdown: Reading Kernel Memory from User Space*". In: *USENIX Security 2018: 27th USENIX Security Symposium*. Ed. by William Enck and Adrienne Porter Felt. Baltimore, MD, USA: USENIX Association, Aug. 2018, pp. 973–990.

[240] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. *"A Survey of Microarchitectural Side-channel Vulnerabilities, Attacks, and Defenses in Cryptography"*. In: *ACM Comput. Surv.* 54.6 (2022), 122:1–122:37. DOI: 10.1145/3456629. URL: https://doi.org/10.1145/3456629.

[241] Matthias Lutter. https://lutter.cc/unshackled/brainfuck.html.

[242] Paolo Mainardi. *The Rising Threat of Software Supply Chain Attacks: Managing Dependencies of Open Source projects*. https://linuxfoundation.eu/newsroom/the-rising-threat-of-software-supply-chain-attacks-managing-dependencies-of-open-source-projects. 2023.

[243] Daniel Mayer and Joel Sandin. *Time Trial: Racing Towards Practical Remote Timing Attacks*. Tech. rep. Available at https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/TimeTrial.pdf. NCC Group, 2014.

[244] Peter Mayer, Damian Poddebniak, Konstantin Fischer, Marcus Brinkmann, Juraj Somorovsky, Angela Sasse, Sebastian Schinzel, and Melanie Volkamer. *"" I {don't} know why I check this..."- Investigating Expert Users' Strategies to Detect Email Signature Spoofing Attacks"*. In: *Eighteenth Symposium on Usable Privacy and Security (SOUPS 2022)*. 2022, pp. 77–96.

[245] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. *"Reliability and Inter-Rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice"*. In: *ACM on Human-Computer Interaction* 3.CSCW, 72 (2019), pp. 1–23.

[246] Nora McDonald, Sarita Schoenebeck, and Andrea Forte. *"Reliability and Inter-rater Reliability in Qualitative Research: Norms and Guidelines for CSCW and HCI Practice"*. In: *Proc. ACM Hum. Comput. Interact.* 3.CSCW (2019), 72:1–72:23. DOI: 10.1145/3359174. URL: https://doi.org/10.1145/3359174.

[247] Susan E McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. *"Investigating the computer security practices and needs of journalists"*. In: *Proceedings of the 24th USENIX Security Symposium (Sec'15)*. 2015, pp. 399–414.

[248] Susan E McGregor, Elizabeth Anne Watkins, Mahdi Nasrullah Al-Ameen, Kelly Caine, and Franziska Roesner. *"When the weakest link is strong: Secure collaboration in the case of the Panama Papers"*. In: *Proceedings of the 26th USENIX Security Symposium (Sec'17)*. 2017, pp. 505–522.

[249] Marcela S. Melara and Santiago Torres-Arias. *"A Viewpoint on Software Supply Chain Security: Are We Getting Lost in Translation?"* In: *IEEE Security & Privacy* 21.6 (2023), pp. 55–58. DOI: 10.1109/MSEC.2023.3316568.

[250] Uta Menges, Jonas Hielscher, Annalina Buckmann, Annette Kluge, M. Angela Sasse, and Imogen Verret. *"Why IT Security Needs Therapy"*. In: *Computer Security. ESORICS 2021 International Workshops*. Ed. by Sokratis Katsikas, Costas Lambrinoudakis, Nora Cuppens, John Mylopoulos, Christos Kalloniatis, Weizhi Meng, Steven Furnell, Frank Pallas, Jörg Pohle, M. Angela Sasse, Habtamu Abie, Silvio Ranise, Luca Verderame, Enrico Cambiaso, Jorge Maestre Vidal, and Marco Antonio Sotelo Monge. Cham: Springer International Publishing, 2022, pp. 335–356. ISBN: 978-3-030-95484-0.

[251] Kelsey Merrill, Zachary Newman, Santiago Torres-Arias, and Karen Sollins. *"Speranza: Usable, privacy-friendly software signing"*. In: *arXiv preprint arXiv:2305.06463* (2023).

[252] Vishal Midha and Prashant Palvia. *"Factors affecting the success of Open Source Software"*. In: *Journal of Systems and Software* 85.4 (2012), pp. 895–905.

[253] Courtney Miller, Sophie Cohen, Bogdan Vasilescu, and Christian Kästner. *"'Did You Miss My Comment or What?' Understanding Toxicity in Open Source Discussions"*. In: *Proceedings of the 44th International Conference on Software Engineering (ICSE'22)*. 2022.

[254] Courtney Miller, Christian Kästner, and Bogdan Vasilescu. *""We Feel Like We're Winging It:" A Study on Navigating Open-Source Dependency Abandonment*". In: *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2023. , San Francisco, CA, USA, Association for Computing Machinery, 2023, pp. 1281–1293. ISBN: 9798400703270. DOI: 10.1145/3611643.3616293. URL: https://doi.org/10.1145/3611643.3616293.

[255] Courtney Miller, David Gray Widder, Christian Kästner, and Bogdan Vasilescu. *"Why Do People Give Up FLOSSing? A Study of Contributor Disengagement in Open Source*". In: *Open Source Systems*. Springer International Publishing, 2019, pp. 116–129. ISBN: 978-3-030-20883-7.

[256] Jaron Mink, Harjot Kaur, Juliane Schmüser, Sascha Fahl, and Yasemin Acar. *""Security is not my field, I'm a stats guy": A Qualitative Root Cause Analysis of Barriers to Adversarial Machine Learning Defenses in Industry*". In: *Proceedings of the 32nd USENIX Security Symposium*. 2023.

[257] Audris Mockus, Roy T Fielding, and James D Herbsleb. *"Two case studies of open source software development: Apache and Mozilla*". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 11.3 (2002), pp. 309–346.

[258] Daniel Moghimi. *"Downfall: Exploiting Speculative Data Gathering*". In: *32nd USENIX Security Symposium, USENIX Security 2023*. USENIX Association, 2023, pp. 7179–7193.

[259] Gideon Mohr, Marco Guarnieri, and Jan Reineke. *"Synthesizing Hardware-Software Leakage Contracts for RISC-V Open-Source Processors*". In: *Proceedings of the 27th Design, Automation and Test in Europe Conference and Exhibition*. ACM/IEEE, 2024, to appear.

[260] Lukas Moldon, Markus Strohmaier, and Johannes Wachs. *"How Gamification Affects Software Developers: Cautionary Evidence from a Natural Experiment on GitHub*". In: *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. 2021, pp. 549–561.

[261] Marina Moore, Trishank Karthik Kuppusamy, and Justin Cappos. *"Artemis: Defanging Software Supply Chain Attacks in Multi-repository Update Systems*". In: *Proceedings of the 39th Annual Computer Security Applications Conference*. 2023, pp. 83–97.

[262] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. *"Axiomatic hardware-software contracts for security*". In: *ISCA '22: Proceedings of the 49th Annual International Symposium on Computer Architecture*. ACM, 2022, pp. 72–86.

[263] Omar S. Navarro Leija, Kelly Shiptoski, Ryan G. Scott, Baojun Wang, Nicholas Renner, Ryan R. Newton, and Joseph Devietti. *"Reproducible Containers*". In: *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20)*. 2020, pp. 167–182.

[264] Moritz Neikes. *TIMECOP*.

[265] Matus Nemec, Dusan Klinec, Petr Svenda, Peter Sekan, and Vashek Matyas. *"Measuring Popularity of Cryptographic Libraries in Internet-Wide Scans*". In: *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC 2017)*. ACM, 2017. DOI: 10.1145/3134600.3134612.

[266] Shradha Neupane, Grant Holmes, Elizabeth Wyss, Drew Davidson, and Lorenzo De Carli. *"Beyond typosquatting: an in-depth look at package confusion*". In: *32nd USENIX Security Symposium (USENIX Security 23)*. 2023, pp. 3439–3456.

[267] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. *"Sigstore: Software Signing for Everybody*". In: *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. CCS '22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, pp. 2353–2367. ISBN: 9781450394505. DOI: 10.1145/3548606.3560596. URL: https://doi.org/10.1145/3548606.3560596.

[268] Zachary Newman, John Speed Meyers, and Santiago Torres-Arias. "*Sigstore: Software Signing for Everybody*". In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 2353–2367. DOI: 10.1145/3548606.3560596.

[269] 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.

[270] Thanh-Dat Nguyen, Yang Zhou, Xuan Bach D Le, David Lo, et al. "*Adversarial Attacks on Code Models with Discriminative Graph Patterns*". In: *arXiv preprint arXiv:2308.11161* (2023).

[271] Flemming Nielson, Hanne R Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2015.

[272] Janneke Nieuwenhuizen and Ludovic Courtès. https://guix.gnu.org/en/blog/2023/the-full-source-bootstrap-building-from-source-all-the-way-down/. 2023.

[273] Kirill Nikitin, Eleftherios Kokoris-Kogias, Philipp Jovanovic, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Justin Cappos, and Bryan Ford. "*CHAINIAC: Proactive Software-Update Transparency via Collectively Signed Skipchains and Verified Builds*". In: *USENIX Security 2017: 26th USENIX Security Symposium*. Ed. by Engin Kirda and Thomas Ristenpart. Vancouver, BC, Canada: USENIX Association, Aug. 2017, pp. 1271–1287.

[274] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. "*DifFuzz: differential fuzzing for side-channel analysis*". In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE '19. Montreal, Quebec, Canada: IEEE, 2019, pp. 176–187. DOI: 10.1109/ICSE.2019.00034.

[275] NIST. *Submission Requirements and Evaluation Criteria for the Post-Quantum Cryptography Standardization Process*. 2016. URL: https://csrc.nist.gov/CSRC/media/Projects/Post-Quantum-Cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[276] *NSA, CISA, ODNI Release Software Supply Chain Guidance for Developers*. https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3146465/nsa-cisa-odni-release-software-supply-chain-guidance-for-developers/. 2022. URL: https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3146465/nsa-cisa-odni-release-software-supply-chain-guidance-for-developers/.

[277] Anna Offenwanger, Alan John Milligan, Minsuk Chang, Julia Bullard, and Dongwook Yoon. "*Diagnosing Bias in the Gender Representation of HCI Research Participants: How It Happens and Where We Are*". In: *Proceedings of the 2021 ACM Conference on Human Factors in Computing Systems (CHI'21)*. 2021.

[278] Chinenye Okafor, Taylor R Schorlemmer, Santiago Torres-Arias, and James C Davis. "*Sok: Analysis of software supply chain security by establishing secure design properties*". In: *Proceedings of the 2022 ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*. 2022, pp. 15–24.

[279] Ben Olmsted. https://esolangs.org/wiki/Malbolge. 1998.

[280] Aleph One. *Smashing The Stack For Fun And Profit*. http://phrack.org/issues/49/14.html. 1996.

[281] OpenSSF. *OpenSSF Scorecards*. https://securityscorecards.dev/.

[282] Tavis Ormandy. *Zenbleed*. https://lock.cmpxchg8b.com/zenbleed.html. 2023.

[283] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "*Cache Attacks and Countermeasures: The Case of AES*". In: *Topics in Cryptology – CT-RSA 2006*. Ed. by David Pointcheval. Vol. 3860. Lecture Notes in Computer Science. San Jose, CA, USA: Springer, Heidelberg, Germany, Feb. 2006, pp. 1–20. DOI: 10.1007/11605805_1.

[284] Thales Bandiera Paiva and Routo Terada. *"A Timing Attack on the HQC Encryption Scheme"*. In: *SAC 2019: 26th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Kenneth G. Paterson and Douglas Stebila. Vol. 11959. Lecture Notes in Computer Science. Waterloo, ON, Canada: Springer, Heidelberg, Germany, Aug. 2019, pp. 551–573. DOI: 10.1007/978-3-030-38471-5_22.

[285] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. *"VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits"*. In: *ACM CCS 2015: 22nd Conference on Computer and Communications Security*. Ed. by Indrajit Ray, Ninghui Li, and Christopher Kruegel. Denver, CO, USA: ACM Press, Oct. 2015, pp. 426–437. DOI: 10.1145/2810103.2813604.

[286] Alan J. Perlis. *"Special Feature: Epigrams on Programming"*. In: *SIGPLAN Not.* 17.9 (Sept. 1982), pp. 7–13. ISSN: 0362-1340. DOI: 10.1145/947955.1083808. URL: https://doi.org/10.1145/947955.1083808.

[287] Jason Perlow. *A Summary of Census II: Open Source Software Application Libraries the World Depends On*. https://www.linuxfoundation.org/blog/blog/a-summary-of-census-ii-open-source-software-application-libraries-the-world-depends-on. 2022.

[288] Mike Perry, Seth Schoen, and Hans Steiner. *Reproducible Builds Moving Beyond Single Points of Failure for Software Distribution*. https://media.ccc.de/v/31c3_-_6240_-_en_-_saal_g_-_201412271400_-_reproducible_builds_-_mike_perry_-_seth_schoen_-_hans_steiner. 2014.

[289] Peter Pessl, Leon Groot Bruinderink, and Yuval Yarom. *"To BLISS-B or not to be: Attacking strongSwan's Implementation of Post-Quantum Signatures"*. In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 1843–1855. DOI: 10.1145/3133956.3134023.

[290] Karl Pettis and Robert C. Hansen. *"Profile Guided Code Positioning"*. In: *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*. PLDI '90. White Plains, New York, USA: Association for Computing Machinery, 1990, pp. 16–27. ISBN: 0897913647. DOI: 10.1145/93542.93550. URL: https://doi.org/10.1145/93542.93550.

[291] Valentina Piantadosi, Simone Scalabrino, and Rocco Oliveto. *"Fixing of Security Vulnerabilities in Open Source Projects: A Case Study of Apache HTTP Server and Apache Tomcat"*. In: *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST'19)*. 2019, pp. 68–78.

[292] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. *"The Software Heritage Graph Dataset: Large-Scale Analysis of Public Software Development History"*. In: *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*. 2020, pp. 1–5.

[293] Antoine Pietri, Diomidis Spinellis, and Stefano Zacchiroli. *"The Software Heritage Graph Dataset: Public Software Development under One Roof"*. In: *Proceedings of the 16th International Conference on Mining Software Repositories (MSR'19)*. 2019, pp. 138–142.

[294] Gustavo Pinto, Igor Steinmacher, and Marco Aurélio Gerosa. *"More Common Than You Think: An In-depth Study of Casual Contributors"*. In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER'16)*. 2016, pp. 112–123.

[295] Daniel Pletea, Bogdan Vasilescu, and Alexander Serebrenik. *"Security and Emotion: Sentiment Analysis of Security Discussions on GitHub"*. In: *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. 2014, pp. 348–351.

[296] Rachel Potvin and Josh Levenberg. *"Why Google Stores Billions of Lines of Code in a Single Repository"*. In: *Commun. ACM* 59.7 (June 2016), pp. 78–87.

[297] LLVM project. https://llvm.org/docs/LangRef.html.

[298] The Bootstrappable Builds Project. https://bootstrappable.org/. 2016-2023.

[299] The Reproducible Builds Project. https://reproducible-builds.org/. 2013-2023.

[300] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Béguelin. "*EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider*". In: *2020 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2020, pp. 983–1002. DOI: 10.1109/SP40000.2020.00114.

[301] Naveen Raman, Minxuan Cao, Yulia Tsvetkov, Christian Kästner, and Bogdan Vasilescu. "*Stress and Burnout in Open Source: Toward Finding, Understanding, and Mitigating Unhealthy Interactions*". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER'20)*. 2020, pp. 57–60.

[302] Ralf Ramsauer, Lukas Bulwahn, Daniel Lohmann, and Wolfgang Mauerer. "*The Sound of Silence: Mining Security Vulnerabilities from Secret Integration Channels in Open-Source Projects*". In: *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*. CCSW'20. Virtual Event, USA: Association for Computing Machinery, 2020. URL: https://doi.org/10.1145/3411495.3421360.

[303] Alastair Reid, Luke Church, Shaked Flur, Sarah de Haas, Maritza Johnson, and Ben Laurie. "*Towards making formal methods normal: meeting developers where they are*". In: *arXiv preprint arXiv:2010.16345* (2020).

[304] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. *Automated Localization for Unreproducible Builds*. 2018. DOI: 10.48550/ARXIV.1803.06766. URL: https://arxiv.org/abs/1803.06766.

[305] Zhilei Ren, Changlin Liu, Xusheng Xiao, He Jiang, and Tao Xie. "*Root Cause Localization for Unreproducible Builds via Causality Analysis Over System Call Tracing*". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 527–538. DOI: 10.1109/ASE.2019.00056.

[306] Zhilei Ren, Shiwei Sun, Jifeng Xuan, Xiaochen Li, Zhide Zhou, and He Jiang. "*Automated Patching for Unreproducible Builds*". In: *Proceedings of the 44th ACM International Conference on Software Engineering (ICSE'22)*. 2022, pp. 200–211.

[307] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. "*Dude, is my code constant time?*" In: *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*. Ed. by David Atienza and Giorgio Di Natale. IEEE, 2017, pp. 1697–1702. DOI: 10.23919/DATE.2017.7927267. URL: https://doi.org/10.23919/DATE.2017.7927267.

[308] *Reproducible Arch Linux?!* https://tests.reproducible-builds.org/archlinux/archlinux.html. 2023.

[309] Reproducible Builds project. https://reproducible-builds.org/docs/definition/. URL: https://reproducible-builds.org/docs/definition/.

[310] *Reproducible Debian Overview*. https://tests.reproducible-builds.org/debian/reproducible.html. 2023.

[311] *reprotest*. https://salsa.debian.org/reproducible-builds/reprotest. 2016. URL: https://salsa.debian.org/reproducible-builds/reprotest.

[312] Joshua Reynolds, Trevor Smith, Ken Reese, Luke Dickinson, Scott Ruoti, and Kent E. Seamons. "*A Tale of Two Studies: The Best and Worst of YubiKey Usability*". In: *2018 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2018, pp. 872–888. DOI: 10.1109/SP.2018.00067.

[313] Henry Gordon Rice. "*Classes of recursively enumerable sets and their decision problems*". In: *Transactions of the American Mathematical society* 74.2 (1953), pp. 358–366.

[314] Alan JA Robinson and Andrei Voronkov. *Handbook of automated reasoning*. Vol. 1. Elsevier, 2001.

[315] Gregorio Robles, Laura Arjona Reina, Alexander Serebrenik, Bogdan Vasilescu, and Jesús M. González-Barahona. "*FLOSS 2013: A Survey Dataset about Free Software Contributors: Challenges for Curating, Sharing, and Combining*". In: *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. 2014, pp. 396–399.

[316] Bruno Rodrigues, Fernando Magno Quintão Pereira, and Diego F. Aranha. "*Sparse representation of implicit flows with applications to side-channel detection*". In: *Proceedings of the 25th International Conference on Compiler Construction, CC 2016, Barcelona, Spain, March 12-18, 2016*. Ed. by Ayal Zaks and Manuel V. Hermenegildo. ACM, 2016, pp. 110–120. DOI: 10.1145/2892208.2892230. URL: https://doi.org/10.1145/2892208.2892230.

[317] Phillip Rogaway. *The Moral Character of Cryptographic Work*. Cryptology ePrint Archive, Report 2015/1162. https://eprint.iacr.org/2015/1162. 2015.

[318] Sebastian Roth, Lea Gröber, Michael Backes, Katharina Krombholz, and Ben Stock. "*12 Angry Developers - A Qualitative Study on Developers' Struggles with CSP*". In: *ACM CCS 2021: 28th Conference on Computer and Communications Security*. Ed. by Giovanni Vigna and Elaine Shi. Virtual Event, Republic of Korea: ACM Press, Nov. 2021, pp. 3085–3103. DOI: 10.1145/3460120.3484780.

[319] Jungwoo Ryoo, Bryan Malone, Phillip A. Laplante, and Priya Anand. "*The Use of Security Tactics in Open Source Software Projects*". In: *IEEE Transactions on Reliability* 65.3 (2016), pp. 1195–1204. DOI: 10.1109/TR.2015.2500367.

[320] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. "*Lessons from building static analysis tools at Google*". In: *Commun. ACM* 61.4 (2018), pp. 58–66. DOI: 10.1145/3188720.

[321] Joanna C. S. Santos, Anthony Peruma, Mehdi Mirakhorli, Matthias Galstery, Jairo Veloz Vidal, and Adriana Sejfia. "*Understanding Software Vulnerabilities Related to Architectural Security Tactics: An Empirical Investigation of Chromium, PHP and Thunderbird*". In: *Proceedings of the 2017 IEEE International Conference on Software Architecture (ICSA'17)*. 2017, pp. 69–78.

[322] *sbuild*. https://salsa.debian.org/debian/sbuild.

[323] Walt Scacchi, Joseph Feller, Brian Fitzgerald, Scott Hissam, and Karim Lakhani. *Understanding free/open source software development processes*. 2006.

[324] Simone Scalco, Ranindya Paramitha, Duc-Ly Vu, and Fabio Massacci. "*On the feasibility of detecting injections in malicious npm packages*". In: *Proceedings of the 17th International Conference on Availability, Reliability and Security*. 2022, pp. 1–8.

[325] Stephen R. Schach. *Practical Software Engineering*. 1992.

[326] Alexander Schaub. "*Formal methods for the analysis of cache-timing leaks and key generation in cryptographic implementations*". Theses. Institut Polytechnique de Paris, Dec. 2020.

[327] Lou Scheffer. *Programming in Malbolge*. http://www.lscheffer.com/malbolge.shtml. 2015.

[328] Bruce Schneier. https://www.schneier.com/crypto-gram/archives/1998/1015.html#cipherdesign. 1998.

[329] Peter Schwabe. *Eliminating Timing Side-Channels. A Tutorial*. https://cryptojedi.org/peter/data/shmoocon-20150118.pdf. Jan. 18, 2015.

[330] *Secure development and deployment guidance*. https://www.ncsc.gov.uk/collection/developers-collection/principles/secure-your-development-environment.

[331] Alessandro Segala. *How (and why) to sign Git commits*. https://withblue.ink/2020/05/17/how-and-why-to-sign-git-commits.html. 2020.

[332] Muhammad Shahzad, Muhammad Zubair Shafiq, and Alex X. Liu. *"A Large Scale Exploratory Analysis of Software Vulnerability Life Cycles"*. In: *Proceedings of the 34th International Conference on Software Engineering (ICSE'12)*. 2012, pp. 771–781.

[333] Andrew Shallue and Christiaan E. van de Woestijne. *"Construction of Rational Points on Elliptic Curves over Finite Fields"*. In: *Algorithmic Number Theory, 7th International Symposium, ANTS-VII*. Ed. by Florian Hess, Sebastian Pauli, and Michael E. Pohst. Vol. 4076. LNCS. SV, 2006, pp. 510–524. URL: https://doi.org/10.1007/11792086_36.

[334] Ayushi Sharma, Shashank Sharma, Santiago Torres-Arias, and Aravind Machiry. *Rust for Embedded Systems: Current State, Challenges and Open Problems*. 2023. arXiv: 2311.05063 [cs.CR].

[335] Young-joo Shin, Hyung Chan Kim, Dokeun Kwon, Ji-Hoon Jeong, and Junbeom Hur. *"Unveiling Hardware-based Data Prefetcher, a Hidden Source of Information Leakage"*. In: *ACM CCS 2018: 25th Conference on Computer and Communications Security*. Ed. by David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang. Toronto, ON, Canada: ACM Press, Oct. 2018, pp. 131–145. DOI: 10.1145/3243734.3243736.

[336] Kelly Shortridge and Josiah Dykstra. *"Opportunity Cost and Missed Chances in Optimizing Cybersecurity: The Loss of Potential Gain from Other Alternatives When One Alternative is Chosen"*. In: *Queue* 21.1 (Apr. 2023), pp. 30–56. ISSN: 1542-7730. DOI: 10.1145/3588041. URL: https://doi.org/10.1145/3588041.

[337] Mario Silic and Andrea Back. *"Information Security and Open Source Dual Use Security Software: Trust Paradox"*. In: *Open Source Software: Quality Verification*. Ed. by Etiel Petrinja, Giancarlo Succi, Nabil El Ioini, and Alberto Sillitti. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 194–206. ISBN: 978-3-642-38928-3.

[338] Laurent Simon, David Chisnall, and Ross J. Anderson. *"What You Get is What You C: Controlling Side Effects in Mainstream C Compilers"*. In: *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018*. London, United Kingdom: IEEE, Apr. 2018, pp. 1–15. DOI: 10.1109/EUROSP.2018.00009. URL: https://doi.org/10.1109/EuroSP.2018.00009.

[339] Vibha Singhal Sinha, Senthil Mani, and Saurabh Sinha. *"Entering the Circle of Trust: Developer Initiation as Committers in Open-Source Projects"*. In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. MSR '11. Waikiki, Honolulu, HI, USA: Association for Computing Machinery, 2011, pp. 133–142. ISBN: 9781450305747. DOI: 10.1145/1985441.1985462. URL: https://doi.org/10.1145/1985441.1985462.

[340] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. *"When Do Changes Induce Fixes?"* In: *SIGSOFT Softw. Eng. Notes* 30.4 (May 2005), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/1082983.1083147. URL: https://doi.org/10.1145/1082983.1083147.

[341] Inna Smirnova, Markus Reitzig, and Oliver Alexy. *"What makes the right OSS contributor tick? Treatments to motivate high-skilled developers"*. In: *Research Policy* 51.1 (2022), p. 104368. ISSN: 0048-7333. DOI: https://doi.org/10.1016/j.respol.2021.104368. URL: https://www.sciencedirect.com/science/article/pii/S0048733321001657.

[342] Devika Sondhi, Avyakt Gupta, Salil Purandare, Ankit Rana, Deepanshu Kaushal, and Rahul Purandare. *"Dataset to Study Indirectly Dependent Documentation in GitHub Repositories"*. In: *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2021, pp. 215–216. DOI: 10.1109/ICSE-Companion52605.2021.00096.

[343] Igor Steinmacher, Tayana Conte, Marco Aurélio Gerosa, and David Redmiles. *"Social Barriers Faced by Newcomers Placing Their First Contribution in Open Source Software Projects"*. In: *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing (SCW'15)*. 2015, pp. 1379–1392.

[344] Igor Steinmacher, Tayana Uchoa Conte, Christoph Treude, and Marco Aurélio Gerosa. *"Overcoming Open Source Project Entry Barriers with a Portal for Newcomers"*. In: *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 2016, pp. 273–284.

[345] Igor Steinmacher, Christoph Treude, and Marco Aurelio Gerosa. *"Let Me In: Guidelines for the Successful Onboarding of Newcomers to Open Source Projects"*. In: *IEEE Software* 36.4 (2019), pp. 41–49. DOI: 10.1109/MS.2018.110162131.

[346] Evgeniy Stepanov and Konstantin Serebryany. *"MemorySanitizer: fast detector of uninitialized memory use in C++"*. In: *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2015, pp. 46–55.

[347] Vikram N Subramanian, Ifraz Rehman, Meiyappan Nagappan, and Raula Gaikovina Kula. *"Analyzing first contributions on GitHub: what do newcomers do"*. In: *IEEE Software* (2020).

[348] Chungha Sung, Brandon Paulsen, and Chao Wang. *"CANAL: A Cache Timing Analysis Framework via LLVM Transformation"*. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE '18. Montpellier, France: Association for Computing Machinery, 2018, pp. 904–907. ISBN: 9781450359375. DOI: 10.1145/3238147.3240485.

[349] Mahbubul Syeed, Juho Lindman, and Imed Hammouda. *"Measuring Perceived Trust in Open Source Software Communities"*. In: *Open Source Systems: Towards Robust Practices*. Ed. by Federico Balaguer, Roberto Di Cosmo, Alejandra Garrido, Fabio Kon, Gregorio Robles, and Stefano Zacchiroli. Cham: Springer International Publishing, 2017.

[350] Mohammad Tahaei and Kami Vaniea. *"Recruiting Participants With Programming Skills: A Comparison of Four Crowdsourcing Platforms and a CS Student Mailing List"*. In: *CHI '22: Conference on Human Factors in Computing Systems*. ACM, 2022, 590:1–590:15. DOI: 10.1145/3491102.3501957.

[351] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. *"Bug characteristics in open source software"*. In: *Empirical software engineering* 19.6 (2014), pp. 1665–1705.

[352] Technical Committee ISO/TC 159. Subcommittee SC 4. *ISO 9241-11:2018 Ergonomics of Human-system Interaction. Usability : definitions and concepts. Part 11*. International standard. ISO, 2018.

[353] Christopher Thompson and David Wagner. *"A Large-Scale Study of Modern Code Review and Security in Open Source Projects"*. In: *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*. Toronto, Canada: Association for Computing Machinery, 2017. URL: https://doi.org/10.1145/3127005.3127014.

[354] Ken Thompson. *"Reflections on Trusting Trust"*. In: *Commun. ACM* 27.8 (Aug. 1984), pp. 761–763. ISSN: 0001-0782. DOI: 10.1145/358198.358210. URL: https://doi.org/10.1145/358198.358210.

[355] T. J. Thompson. *"Designer's workbench: Providing a production environment"*. In: *The Bell System Technical Journal* 59.9 (1980), pp. 1811–1825. DOI: 10.1002/j.1538-7305.1980.tb03063.x.

[356] Mehdi Tibouchi and Alexandre Wallet. *"One Bit is All It Takes: A Devastating Timing Attack on BLISS's Non-Constant Time Sign Flips"*. In: *J. Math. Cryptol.* 15.1 (2021), pp. 131–142. DOI: 10.1515/jmc-2020-0079. URL: https://doi.org/10.1515/jmc-2020-0079.

[357] S. Torres-Arias, D. Geer, and J. Meyers. *"A Viewpoint on Knowing Software: Bill of Materials Quality When You See It"*. In: *IEEE Security & Privacy* 21.06 (Nov. 2023), pp. 50–54. ISSN: 1558-4046. DOI: 10.1109/MSEC.2023.3315887.

[358] Santiago Torres-Arias, Hammad Afzali, Trishank Karthik Kuppusamy, Reza Curtmola, and Justin Cappos. *"in-toto: Providing farm-to-table guarantees for bits and bytes"*. In: *USENIX Security 2019: 28th USENIX Security Symposium*. Ed. by Nadia Heninger and Patrick Traynor. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 1393–1410.

[359] Jason Tsay, Laura Dabbish, and James Herbsleb. "*Influence of Social and Technical Factors for Evaluating Contribution in GitHub*". In: *Proceedings of the 36th International Conference on Software Engineering (ICSE'14)*. 2014, pp. 356–366.

[360] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. "*Cryptanalysis of DES implemented on computers with cache*". In: *Cryptographic Hardware and Embedded Systems – CHES 2003*. Vol. 2779. LNCS. Springer, 2003, pp. 62–76.

[361] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Miyauchi. "*Cryptanalysis of Block Ciphers Implemented on Computers with Cache*". In: *Proceedings of the International Symposium on Information Theory and Its Applications, ISITA 2002*. 2002, pp. 803–806.

[362] Qiang Tu et al. "*Evolution in open source software: A case study*". In: *Proceedings of the 2000 International Conference on Software Maintenance*. IEEE. 2000, pp. 131–142.

[363] UCSD PLSysSec. *haybale-pitchfork*. URL: https://github.com/PLSysSec/haybale-pitchfork.

[364] UCSD PLSysSec. *pitchfork-angr*. URL: https://github.com/PLSysSec/pitchfork-angr.

[365] Alexander Ulrich, Ralph Holz, Peter Hauck, and Georg Carle. "*Investigating the OpenPGP Web of Trust*". In: *Proc. 16th European Symposium on Research in Computer Security (ESORICS'11)*. Springer, 2011.

[366] Engin Kirda and Thomas Ristenpart, eds. *USENIX Security 2017: 26th USENIX Security Symposium*. Vancouver, BC, Canada: USENIX Association, Aug. 2017.

[367] William Enck and Adrienne Porter Felt, eds. *USENIX Security 2018: 27th USENIX Security Symposium*. Baltimore, MD, USA: USENIX Association, Aug. 2018.

[368] Nadia Heninger and Patrick Traynor, eds. *USENIX Security 2019: 28th USENIX Security Symposium*. Santa Clara, CA, USA: USENIX Association, Aug. 2019.

[369] Srdjan Capkun and Franziska Roesner, eds. *USENIX Security 2020: 29th USENIX Security Symposium*. USENIX Association, Aug. 2020.

[370] Kevin R. B. Butler and Kurt Thomas, eds. *USENIX Security 2022: 31st USENIX Security Symposium*. Boston, MA, USA: USENIX Association, Aug. 2022.

[371] Sangat Vaidya, Santiago Torres-Arias, Justin Cappos, and Reza Curtmola. "*Bootstrapping Trust in Community Repository Projects*". In: *International Conference on Security and Privacy in Communication Systems*. Springer. 2022, pp. 450–469.

[372] Mathy Vanhoef and Eyal Ronen. "*Dragonblood: Analyzing the Dragonfly Handshake of WPA3 and EAP-pwd*". In: *2020 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2020, pp. 517–533. DOI: 10.1109/SP40000.2020.00031.

[373] Bogdan Vasilescu, Kelly Blincoe, Qi Xuan, Casey Casalnuovo, Daniela Damian, Premkumar Devanbu, and Vladimir Filkov. "*The Sky is Not the Limit: Multitasking across GitHub Projects*". In: *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 2016, pp. 994–1005.

[374] Jose Rodrigo Sanchez Vicarte, Michael Flanders, Riccardo Paccagnella, Grant Garrett-Grossman, Adam Morrison, Christopher W. Fletcher, and David Kohlbrenner. "*Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest*". In: *2022 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE Computer Society Press, May 2022, pp. 1491–1505. DOI: 10.1109/SP46214.2022.9833570.

[375] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. "*Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data*". In: *42nd IEEE Symposium on Security and Privacy, SP 2022*. IEEE, 2021, pp. 347–360.

[376] 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: *USENIX Security 2020: 29th USENIX Security Symposium*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, Aug. 2020, pp. 109–126.

[377] Duc-Ly Vu, Zachary Newman, and John Speed Meyers. *"Bad Snakes: Understanding and Improving Python Package Index Malware Scanning"*. In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE. 2023, pp. 499–511.

[378] Guillaume Wafo-Tapa, Slim Bettaieb, Loïc Bidoux, Philippe Gaborit, and Etienne Marcatel. *"A practicable timing attack against HQC and its countermeasure"*. In: *Advances in Mathematics of Computation* (2020). URL: http://dx.doi.org/10.3934/amc.2020126.

[379] James Walden. *"The impact of a major security event on an open source project: The case of OpenSSL"*. In: *Proceedings of the 17th International Conference on Mining Software Repositories (MSR'20)*. 2020, pp. 409–419.

[380] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. *"Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation"*. In: *USENIX Security 2019: 28th USENIX Security Symposium*. Ed. by Nadia Heninger and Patrick Traynor. Santa Clara, CA, USA: USENIX Association, Aug. 2019, pp. 657–674.

[381] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. *"CacheD: Identifying Cache-Based Timing Channels in Production Software"*. In: *USENIX Security 2017: 26th USENIX Security Symposium*. Ed. by Engin Kirda and Thomas Ristenpart. Vancouver, BC, Canada: USENIX Association, Aug. 2017, pp. 235–252.

[382] Wubing Wang, Yinqian Zhang, and Zhiqiang Lin. *"Time and Order: Towards Automatically Identifying Side-Channel Vulnerabilities in Enclave Binaries"*. In: *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. Chaoyang District, Beijing: USENIX Association, Sept. 2019, pp. 443–457. ISBN: 978-1-939133-07-6.

[383] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. *"Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86"*. In: *USENIX Security 2022: 31st USENIX Security Symposium*. Ed. by Kevin R. B. Butler and Kurt Thomas. Boston, MA, USA: USENIX Association, Aug. 2022, pp. 679–697.

[384] Conrad Watt, John Renner, Natalie Popescu, Sunjay Cauligi, and Deian Stefan. *"CT-wasm: type-driven secure cryptography for the web ecosystem"*. In: *Proc. ACM Program. Lang.* 3.POPL (2019), 77:1–77:29. DOI: 10.1145/3290390. URL: https://doi.org/10.1145/3290390.

[385] Charles Weir, Ingolf Becker, James Noble, Lynne Blair, M Angela Sasse, and Awais Rashid. *"Interventions for long-term software security: Creating a lightweight program of assurance techniques for developers"*. In: *Software: Practice and Experience* 50.3 (2020), pp. 275–298.

[386] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. *"Big Numbers - Big Troubles: Systematically Analyzing Nonce Leakage in (EC)DSA Implementations"*. In: *USENIX Security 2020: 29th USENIX Security Symposium*. Ed. by Srdjan Capkun and Franziska Roesner. USENIX Association, Aug. 2020, pp. 1767–1784.

[387] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. *"DATA - Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries"*. In: *USENIX Security 2018: 27th USENIX Security Symposium*. Ed. by William Enck and Adrienne Porter Felt. Baltimore, MD, USA: USENIX Association, Aug. 2018, pp. 603–620.

[388] Shao-Fang Wen. *"Learning Secure Programming in Open Source Software Communities: A Socio-Technical View"*. In: *Proceedings of the 6th International Conference on Information and Education Technology*. ICIET '18. Osaka, Japan: Association for Computing Machinery, 2018.

[389] Shao-Fang Wen. "*Software security in open source development: A systematic literature review*". In: *2017 21st Conference of Open Innovations Association (FRUCT)*. 2017, pp. 364–373. DOI: 10.23919/FRUCT.2017.8250205.

[390] Shao-Fang Wen, Mazaher Kianpour, and Stewart Kowalski. "*An Empirical Study of Security Culture in Open Source Software Communities*". In: *Proceedings of the 2019 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining*. ASONAM '19. Vancouver, British Columbia, Canada: Association for Computing Machinery, 2020, pp. 863–870. ISBN: 9781450368681. DOI: 10.1145/3341161.3343520. URL: https://doi.org/10.1145/3341161.3343520.

[391] Dominik Wermke, Jan H. Klemmer, Noah Wöhler, Juliane Schmüser, Harshini Sri Ramulu, Yasemin Acar, and Sascha Fahl. "*'Always Contribute Back': A Qualitative Study on Security Challenges of the Open Source Supply Chain*". In: *In Proceedings of the 44th IEEE Symposium on Security and Privacy (IEEE S&P '23)*. IEEE Computer Society, May 2023. URL: https://www.ieee-security.org/TC/SP2023/program-papers.html.

[392] Dominik Wermke, Noah Wöhler, Jan H. Klemmer, Marcel Fourné, Yasemin Acar, and Sascha Fahl. "*Committed to Trust: A Qualitative Study on Security & Trust in Open Source Software Projects*". In: *Proceedings of the 43rd IEEE Symposium on Security and Privacy (S&P'22)*. May 2022.

[393] D. A. Wheeler. "*Countering Trusting Trust through Diverse Double-Compiling*". In: *Proceedings of the 21st IEEE Annual Computer Security Applications Conference (ACSAC'05)*. IEEE, 2005. DOI: 10.1109/csac.2005.17. URL: https://doi.org/10.1109/csac.2005.17.

[394] David A. Wheeler. "*Fully Countering Trusting Trust through Diverse Double-Compiling*". In: *CoRR* abs/1004.5534 (2010). arXiv: 1004.5534. URL: http://arxiv.org/abs/1004.5534.

[395] *Why Constant-Time Crypto?* https://www.bearssl.org/constanttime.html. 2018.

[396] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. "*MicroWalk: A Framework for Finding Side Channels in Binaries*". In: *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 2018, pp. 161–173. DOI: 10.1145/3274694.3274741. URL: https://doi.org/10.1145/3274694.3274741.

[397] Jan Wichelmann, Florian Sieck, Anna Pätschke, and Thomas Eisenbarth. "*Microwalk-CI: Practical Side-Channel Analysis for JavaScript Applications*". In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 2915–2929. DOI: 10.1145/3548606.3560654.

[398] Dan Willemsen. https://android.googlesource.com/platform/build/soong/+/master/docs/best_practices.md. 2016.

[399] Gordon B Willis. *Cognitive interviewing: A tool for improving questionnaire design*. sage publications, 2004.

[400] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. "*Eliminating timing side-channel leaks using program repair*". In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. Ed. by Frank Tip and Eric Bodden. ACM, 2018, pp. 15–26. DOI: 10.1145/3213846.3213851. URL: https://doi.org/10.1145/3213846.3213851.

[401] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. "*STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves*". In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 859–874. DOI: 10.1145/3133956.3134016.

[402] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "*CacheBleed: a timing attack on OpenSSL constant-time RSA*". In: *Journal of Cryptographic Engineering* 7.2 (June 2017), pp. 99–112. DOI: 10.1007/s13389-017-0152-y.

[403] Tuba Yavuz, Farhaan Fowze, Grant Hernandez, Ken Yihang Bai, Kevin R. B. Butler, and Dave Jing Tian. *"ENCIDER: Detecting Timing and Cache Side Channels in SGX Enclaves and Cryptographic APIs"*. In: *IEEE Transactions on Dependable and Secure Computing* 20.2 (2023), pp. 1577–1595. DOI: 10.1109/TDSC.2022.3160346.

[404] Yuanyuan Yuan, Zhibo Liu, and Shuai Wang. *"CacheQL: Quantifying and Localizing Cache Side-Channel Vulnerabilities in Production Software"*. In: *32nd USENIX Security Symposium, USENIX Security 2023*. Anaheim, CA, USA: USENIX Association, Aug. 2023, pp. 2009–2026. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/yuan-yuanyuan-cacheql.

[405] Yuanyuan Yuan, Qi Pang, and Shuai Wang. *"Automated Side Channel Analysis of Media Software with Manifold Learning"*. In: *USENIX Security 2022: 31st USENIX Security Symposium*. Ed. by Kevin R. B. Butler and Kurt Thomas. Boston, MA, USA: USENIX Association, Aug. 2022, pp. 4419–4436.

[406] Mansooreh Zahedi, Muhammad Ali Babar, and Christoph Treude. *"An empirical study of security issues posted in open source projects"*. In: *Proceedings of the 51st Hawaii International Conference on System Sciences (HICSS'18)*. 2018, pp. 5504–5513.

[407] Fiorella Zampetti, Simone Scalabrino, Rocco Oliveto, Gerardo Canfora, and Massimiliano Di Penta. *"How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines"*. In: *Proceedings of the 14th IEEE International Conference on Mining Software Repositories (MSR'17)*. 2017, pp. 334–344.

[408] Jamie Zawinski. *LOL Github*. https://www.jwz.org/blog/2018/06/lol-github/. 2018.

[409] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesus Gonzalez-Barahona. *On The Relation Between Outdated Docker Containers, Severity Vulnerabilities and Bugs*. 2018. DOI: 10.48550/ARXIV.1811.12874. URL: https://arxiv.org/abs/1811.12874.

[410] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. *"Cross-VM side channels and their use to extract private keys"*. In: *the ACM Conference on Computer and Communications Security, CCS'12*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM, 2012, pp. 305–316. URL: https://doi.org/10.1145/2382196.2382230.

[411] Yaqin Zhou and Asankhaya Sharma. *"Automated Identification of Security Issues from Commit Messages and Bug Reports"*. In: *Proceedings of the 11th ACM Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*. 2017, pp. 914–919.

[412] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. *"HACL*: A Verified Modern Cryptographic Library"*. In: *ACM CCS 2017: 24th Conference on Computer and Communications Security*. Ed. by Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu. Dallas, TX, USA: ACM Press, Oct. 2017, pp. 1789–1806. DOI: 10.1145/3133956.3134043.