CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	Unhelpful assumptions in software security research
Authors	Ryan, Ita;Roedig, Utz;Stol, Klaas-Jan
Publication date	2023-11-21
Original Citation	Ryan, I., Roedig, U and Stol, K.-J. (2023) 'Unhelpful assumptions in software security research', Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, Copenhagen, Denmark, 26-30 November, pp. 3460-3474. doi: 10.1145/3576915.3623122
Type of publication	Conference item
Link to publisher's version	10.1145/3576915.3623122
Rights	© 2023, the Authors. This work is licensed under a Creative Commons Attribution International 4.0 License. - https://creativecommons.org/licenses/by/4.0/
Download date	2026-11-06 08:02:22
Item downloaded from	https://hdl.handle.net/10468/15271

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Unhelpful Assumptions in Software Security Research

Ita Ryan
School of Computer Science and IT,
University College Cork, Cork
Ireland
ita.ryan@cs.ucc.ie

Utz Roedig
School of Computer Science and IT,
University College Cork, Cork
Ireland
u.roedig@cs.ucc.ie

Klaas-Jan Stol
School of Computer Science and IT,
University College Cork, Cork
Ireland
k.stol@ucc.ie

ABSTRACT

In the study of software security many factors must be considered. Once venturing beyond the simplest of laboratory experiments, the researcher is obliged to contend with exponentially complex conditions. Software security has been shown to be affected by priming, tool usability, library documentation, organisational security culture, the content and format of internet resources, IT team and developer interaction, Internet search engine ordering, developer personality, security warning placement, mentoring, developer experience and more. In a systematic review of software security papers published since 2016, we have identified a number of unhelpful assumptions that are commonly made by software security researchers. In this paper we list these assumptions, describe why they sometimes do not reflect reality, and suggest implications for researchers.

CCS CONCEPTS

• **Security and privacy → Software security engineering**; *Human and societal aspects of security and privacy*.

KEYWORDS

software security, secure software development

ACM Reference Format:
Ita Ryan, Utz Roedig, and Klaas-Jan Stol. 2023. Unhelpful Assumptions in Software Security Research. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623122

1 INTRODUCTION

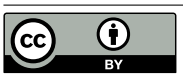
In an increasingly-connected world, online security is ever more important. Many hacking techniques depend on the existence of vulnerabilities in software. Their presence facilitates cybercrime, from ransomware to cyber espionage [90]. The field of Developer Centred Security (DCS) explores reasons why developers write vulnerable code [96], seeking to understand developers, how they can be helped, and what problems they face in practice. Seeking to gain greater understanding of DCS challenges, we conducted a systematic literature review focusing on papers published in known DCS venues since 2016. The objective was to identify reasons why, despite over two decades of software security research, the discovery rate of exploitable vulnerabilities continues to increase [80].

In the course of the review we observed a number of implicit assumptions within the field of software security research. Some are common, some less so. All are unhelpful when attempting to understand developer behaviour. Implicit assumptions may lead researchers to miss relevant information, or to jump to conclusions.

One example of an unhelpful assumption occurs in an excellent ethnographic study by Palombo et al. [72]. The researchers were embedded as coders in a software organisation selling network access software for a combined total of 18 months. Their observations indicated that the development team did not prioritise software security. Working alongside the team, they conducted penetration tests on the software. To their surprise, developers did not immediately fix revealed issues and did not even appear to take them all that seriously. This held true even for a defect involving remote code execution. Defects involving broken authentication looked like a potential systemic threat to the company's existence, yet they were not prioritised for fixing. The researchers discussed the developers balancing their primary aim to develop successful features for the software with their understanding that security is important.

The assumption made here was that the organisation prioritised software security, and it was the development team who were forced to balance that priority with other priorities. However, to send a genuine signal that security is important, management would need not only to state it explicitly, but also, for example, invest in testing including the commissioning of penetration tests, invest in security training and request a prioritisation of security issues in backlog meetings [40]. In this organisation, the researchers undertook penetration testing and found a defect on the first day, suggesting that penetration tests had not been done before. The researchers had to smuggle secure coding techniques into the code base, minimising the resulting impact on development and testing time. The length of time the researchers spent on site allowed them to get highly informative insights on the development process. The assumption that security was an organisational priority, however, meant that, although they observed a lack of security concern on the development team, the researchers did not focus on the secure-coding direction that the team received.

In this paper we list 25 such assumptions in 4 categories. For each assumption we outline the reality that is hidden by the assumption, and the implications for future work in the study of software security. The aim of this paper is not to point fingers or reveal flaws in work by others—much of which we admire and value greatly. Rather, we seek to initiate a discussion and reflection on the line of work that is done in the area of software security research. The goal of this paper is to draw the attention of the software security



This work is licensed under a Creative Commons Attribution International 4.0 License.

CCS '23, November 26–30, 2023, Copenhagen, Denmark
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0050-7/23/11.
https://doi.org/10.1145/3576915.3623122

research community to unhelpful research assumptions. We hope this will trigger discussion and broaden the search for the meaning of observed phenomena.

This paper is, clearly, not the first to review the challenges of developing secure software. Several papers are noteworthy in this context. Recently, Mokhberi and Beznosov identified '*Human, Organizational, and Technological*' dimensions of secure coding challenges, containing 17 challenge categories [56]. They viewed their work as the beginning of a systematic way of thinking about the challenges in software security. Some of the challenges have wide-ranging effects; for example, managers set and enforce priorities, contributing to culture, and therefore have a substantial effect on an organisation's security stance. To develop a security culture, both internal motivations such as valuing security and external motivations such as a belief that security will increase market share are needed. Software security is a non-functional requirement (NFR), and Mokhberi and Beznosov reflected on the additional challenges entailed by its lack of visibility and the constantly changing nature of the threat landscape. Wide-ranging recommendations from this study included using organisational strategies to improve security; for example, making software security visible, ensuring developers know that they have responsibility for it, and designing tool-friendly policies. Providing training and support to developers was also advocated.

Das Chowdhury et al. analysed developer-centred studies, identifying 72 papers for their analysis that were written over 13 years [23]. They sought to categorise challenges, behaviours and interventions in the literature, and how challenges and behaviours interact. They also analysed five '*tropes*,' or '*misplaced beliefs about how developers behave*.' They argued that these five misplaced beliefs strongly influence the design of libraries and tools, and the secure coding interventions that developers endure. These tropes are, that '*developers are experts*,' that '*everyone has the same security and privacy needs*,' that '*security is easy*,' '*security is a universal priority*,' and '*maintenance is fun*.' We too identified the common assumptions that all developers are security experts and that security is easy, although our naming is different.

This paper is organised as follows: Section 2 discusses our research method, Section 3 gives our results. We discuss and conclude the paper in Section 4.

2 METHODOLOGY

2.1 Venues and Search Terms

We selected a number of venues to search for related papers, based on conferences and journals that are known to publish DCS-related material. We also included workshop papers where available. We included papers published from 2016 on. We used the following core search term, allowing autostemming.

(software OR code) AND (developer OR development) AND security

Since no one search engine provided papers from all of our venues, we used the ACM Digital Library, IEEE Xplore, and Scopus. The exact syntax used depended on the search engine. The search resulted in an initial total of 852 papers. The full protocol including paper breakdown is available at https://osf.io/7a2dr/?view_only=e39e7822618f47f0a9bca7e5ec46d86c.

2.2 Selection procedure

The first author undertook the selection procedure, with consultation with the other authors where necessary. The titles and abstracts of all 852 papers were reviewed for first-round exclusions, which excluded duplicates and incomplete papers, such as abstracts, posters, and two-page outlines. In second-round exclusions, our rules were that we would include papers:

- that identify one or more factors that reduce the incidence of security vulnerabilities in code while it is being written;
- that identify one or more factors that increase the incidence of security vulnerabilities in code while it is being written.

The second-round exclusions therefore excluded papers that did not identify factors that reduce or increase the incidence of security issues in code. We included studies that described aids that help developers to code more securely. Some were existing or desired aids that were described in interviews with security experts. Others were new or proposed aids that were devised and assessed in a specific domain with a view to general application in the future. Papers describing tools that help to write secure code in specific problem domains with no application for the broader developer community were excluded. An example would be a paper on a tool to verify contract rules on the Ethereum blockchain. We also excluded papers describing specific implementations of tool types that are already well-known. An example would be a paper describing a new static analysis tool for a specific language. Finally, we excluded proposals for new processes or tools that have not been empirically assessed either in a laboratory setting or in industry. An example would be a proposed method for integrating security into Agile that has not been put into practice anywhere. When in doubt about a paper, it was reviewed by the research team. This process resulted in 90 papers selected for review.

2.3 Snowballing

We identified authors who had worked on three or more of the papers reviewed. Other papers by these authors were assessed for inclusion in the study, yielding 16 additional papers. The references sections of all our selected papers were evaluated and papers of interest with multiple references were selected for review. Since many of these papers were already included in the review, this resulted in only three additional papers.

To ensure that we had not missed important relevant venues, we analysed the venues where our 19 snowballed papers were published. We looked for venues that had published two or more snowballed papers and that we had not already searched. We did not find any such venues amongst the snowballed papers.

Our cutoff date for papers was September 30, 2022.

2.4 Data Extraction and Analysis

Data identification and extraction was done by the first author, with frequent consultation and discussion with the second and third authors. Each paper was analysed and factors affecting the security of code at coding time were extracted and categorised. Such factors could affect the code either positively or adversely. During this process, a number of apparent contradictions between papers came to light. In other cases, an obvious interpretation of the data collected by the researchers was not present. On analysing

these contradictions and missed opportunities, it became apparent that in some cases the root cause was an unconscious assumption made by researchers.

We identified these assumptions and categorised them based on the three dimensions identified by Mokhberi and Beznosov, which we found useful in our own work. Thus we have '*Technical*', '*Organisational*' and '*Developer*' (Mokhberi and Beznosov called these '*Human*') assumptions. Given that in this paper we are exploring and discussing the research process, we added a fourth dimension: '*Research assumptions*.'

2.5 Positionality statement

Given the nature of this study, it is worth noting the background of the authors of this work. The first author has a strong background in logic, having both a degree in Philosophy and a Masters in Computing. They also have 20 years of practical software development experience as a contractor, including a focus on secure software development. The second author has over 25 years of experience in security research. The third author has 15 years of experience in software engineering research, has a strong interest in (and has published about) research methodology, and has focused on 'assumptions' in other works.

3 RESULTS

In the course of our research a number of assumptions emerged from the literature. They appeared to affect researchers' observations and what they paid attention to. In this section, we list these assumptions. For each one we discuss studies where it was made, and studies showing why it is unhelpful or untrue. Table 1 presents a summary of these assumptions. In the table, each assumption is immediately followed by references to the papers that made that assumption. An example of how the assumption affected a paper is included in the 'Implications and Examples' column.

3.1 Technical Assumptions

Assumption: code that obviously should be secure will be written securely.

Reality: security is seen as a cost, not a feature, and is not always prioritised.

Some code is so clearly security-related that it seems self-evident that it should be written securely. Cryptography is a good example. Haney et al. interviewed 21 individuals from organisations with cryptography-related products [40]. The dedication to security of the interview subjects seems impressive. The importance of security in their field is self-evident, and the observations made by Haney et al. appear to confirm that security features highly in the considerations of the organisations under study. Nevertheless, in this excellent study Haney et al. did not attempt to evaluate the actual security of the code produced by the organisations whose employees they interviewed, or to evaluate the secure-coding stance of each organisation. For example, there is no mention of questions regarding code breaches or successful hacks. The limitations acknowledged that the paper is based on perceptions and self-reporting.

Evidence that even cryptography developers do not always use the security tools at their disposal comes from Jancar et al., who interviewed 44 well-qualified developers of 27 open source cryptographic libraries to determine whether they used tools to assess their libraries' vulnerability to constant-time attacks, and if not, why not [45]. They found that, although all participants were aware of timing attacks, tool use was not universal. 38.6% of the developers interviewed used timing attack detection tools, while 25% did not even know about them.

While it would seem self-evident that building security into tools and frameworks is desirable, Pieczul et al. noted that tool vendors may feel that extra security inhibits adoption [81]. They gave the example of enforcing HTTPS for cloud-service endpoints, pointing out that this was not usually done because it would entail significant overhead on the developer side. Tool vendors worried that developers would reject this complexity, choosing a less secure but simpler service instead.

Even where security is a stated focus of a project, factors such as tight deadlines, poor organisation, bad management and running out of time can result in insecure implementations [33]. In their examination of a sub-optimal move to Agile and DevSecOps in a very large development program, Morales et al. found that security concerns were ignored or downplayed, even while the adoption of DevSecOps seemed to indicate that they would be prioritised [57]. Subcontractors faced tight deadlines and inadequate testing resources. The project leadership of a subcontractor explicitly stated that secure coding was not a priority, and speculated that the prime contractor '*probably does security testing*;' not caring enough to know for sure. The absence of secure coding leadership or attention was pervasive, present in the prime contractor and subcontractors, and was reflected in the absence of a security mindset amongst subcontractor developers. It was likely to result in vulnerabilities being released to production and remaining undiscovered for long periods of time [57].

Implications: researchers should carefully investigate the security posture of organisations and developers, even when the need for security appears self-evident.

Assumption: developers can introduce secure coding practices and tools to organisations easily.

Reality: developers are constrained by organisational rules, team norms, faulty tools and peer friction.

An example of this assumption arose in a study of security activities in Finland. Rindell et al. noted that although external penetration tests were perceived by survey respondents to have a high impact on security, they were not widely undertaken [89]. They speculated that perceived impact might not correlate with actual impact. A simpler explanation would be that the employees and engineers who responded to the survey were not in charge of the security budget. Penetration tests are expensive [109].

Developers may not have control over software security and how it is dealt with in their environment [49]. This causes difficulty with the introduction of security interventions and tools [94]. While security tools are essential to secure coding, they can be costly in a number of different ways, making them inaccessible to under-resourced developers. Expensive tools [50] will typically only be available to large or well-resourced organisations. Even ostensibly cheap or free tools will not be accessible to small teams or to isolated

developers if they entail invisible overheads. These may include setup time, training and learning time, and maintenance time [71]. Selecting the wrong tool is costly [55].

Another important factor when developers attempt to introduce new tools or practices is peer buy-in. Embedded in an organisation, Palombo et al. found that their attempts to introduce secure coding practices into their team were resisted until they found a way to make adoption by their peers easy and frictionless [72], a finding echoed in another ethnographic study by Tuladhar et al. [102]. Tools with a high false-positive ratio waste developer time [18, 26, 98]. Other developers may resist their introduction, and the false alerts will create alert fatigue [112]. Tools introduced to existing legacy code may flag enormous amounts of technical debt, causing developers to feel overwhelmed [71]. Security tools can be slow to run and difficult to configure. This causes friction within the team [51, 98]. Tool response times must be prompt. Distefano et al. found that static analysis issues detected overnight and assigned to developers had a response rate close to zero [26]. By contrast, issues detected at commit time and added via bot to code review notes had a fix rate of approximately 70%.

Exploring the reasons for poor use of timing-issue detection tools by cryptography developers, Jancar et al. identified barriers including a lack of developer awareness, a lack of tool usability, the need to annotate code, resource overheads and a fear that tools would not be available or maintained as the researchers who originally developed them moved on [45].

Implications: an absence or low take-up of software security tools and practices may not reflect a lack of developer interest. Environmental constraints should be investigated.

Assumption: if a tool exists, it will be used.

Reality: developers must find and evaluate the tool, and have faith that it will be maintained.

Useful and interesting security tools for developers are constantly under development [7, 32, 34, 67, 68]. There seems to be an implicit assumption that these security tools will be used; we could not find discussions of how the tools would be publicised and disseminated. Achieving tool use is a challenge [18]. Developers may not be aware of the most useful tool for a task, even if it is available in the IDE [94]. Information on program analysis tools is not widely disseminated [6]. Tool efficacy varies [15, 71], and tool evaluation is '*cumbersome and demotivating*' for developers [14]. The pace of development is swift, as is the pace of adoption for new languages and frameworks. Perfect, but top-heavy, languages and IDEs, and the painstakingly-learned lessons they exemplify, can be quickly abandoned [74]. The relatively recent move to a microservice architecture is an example of a new technology causing new pain points, with limited support for microservice-based security found in a search of academic and grey literature [79].

There is also a problem with tool abandonment by academics. We encountered several frameworks [59] and tools [24, 27, 66] that have not been updated by their creators for several years. This lends substance to the problem mentioned in Jancar et al., that the fear that researchers may abandon tools inhibits tool adoption by developers [45].

Implications: researchers should consider whether a proposed tool will be used.

Assumption: secure coding is an individual undertaking.

Reality: most developers work in complex environments with multiple stakeholders.

Research on software security has converged around a number of themes. One is an analogy with usable security proposed by Acar et al. [4]. Just as ordinary users should have usable authentication and other security tools, developers should have usable tools in their environment to facilitate secure coding . There has been considerable research into how to optimise developer security tools for usability. Researchers have focused on issues such as speed and scope of IDE warnings [22], and optimising library error messages [35, 36]. Implicit in the usability analogy is the concept of the developer as a tool-using individual, separate from the social environment in which they write code. Absent pair programming, only one person at a time can write a line of insecure code or fail to include an automated security test.

Issues with the usable security analogy include the fact that the developer's environment is complex, with multiple stakeholders all having potential conflicts of interest [39], and is difficult to replicate in the laboratory in a realistic way [81].

Laboratory studies that focus on the results of a single developer's work have validity, but by their nature [95] they ignore the complexity with which developers must usually contend. This should be borne in mind by researchers. We did not encounter any proposals for or evaluations of tools with a team focus; for example, tools to enhance team secure coding communication.

Implications: researchers should evaluate developers' work environment.

Assumption: Agile is unsuited to secure development.

Reality: issues attributed to use of Agile methods may have causes not related to Agile, for example, they may be due to not prioritising security.

Some studies show a tendency to attribute software security characteristics to the use of the Agile method [70, 83]. For example, Oyetoyan et al. concluded, amongst other things, that effective software security in an Agile setting needs a driver [70]. This implies that software security drivers are not needed in non-Agile settings. Other papers in the review suggest that drivers are needed in all settings. Rajba gave an account of strategies used in a large automotive organisation to improve security in both new and legacy systems [87], vividly illustrating the advantages of having an enthusiastic software security driver. The advantage of availing of software security '*champions*' to spread security enthusiasm in the organisation is discussed throughout the literature [40, 98, 102], with no emphasis on Agile.

This assumption was also made by Poller et al., who observed several development teams during different stages of reaction to an external penetration test [83]. They assessed whether the organisation made changes to prevent the same types of vulnerabilities from reappearing. They were disappointed to see no long-term changes, follow-up, or extra security tasks in the Agile pipeline.

Table 1: Unhelpful Assumptions

Assumption	Reality	Implications and Examples
Technical Assumptions		
Code that obviously should be secure will be written securely [40].	Security is seen as a cost, not a feature, and is not always prioritised.	Researchers should carefully investigate the security posture of organisations and developers, even when the need for security appears self-evident. For Haney et al. [40], this could have revealed a disconnect between stated and actual security.
Developers can introduce secure coding practices and tools to organisations easily [89].	Developers are constrained by organisational rules, team norms, faulty tools and peer friction.	An absence or low take-up of software security tools and practices may not reflect a lack of developer interest. Environmental constraints should be investigated. For Rindell et al. [89], not considering this resulted in a missed opportunity for deeper understanding of the data.
If a tool exists, it will be used [7, 32, 67, 68].	Developers must find and evaluate the tool, and have faith that it will be maintained.	Researchers should consider whether a proposed tool will be used. Ohm et al. devised a framework to detect malicious dependency updates [68]. Their contribution would be greatly enhanced by a plan to promulgate the tool.
Secure coding is an individual undertaking [4].	Most developers work in complex environments with multiple stakeholders.	Researchers should evaluate developers' work environment. Acar et al. [4] suggested that researchers should measure actual developer behaviour, but did not propose gaining a holistic understanding of the developer's context.
Agile is unsuited to secure development [70, 83].	Issues attributed to use of Agile methods may have causes not related to Agile, for example, they may be due to not prioritising security.	Researchers working in an Agile environment should carefully consider whether the use of Agile methods is relevant to uncovered issues. Although attention to good design is an Agile principle, Oyetoyan et al. [70] attributed a desire for secure design training to the Agile nature of the work environment they were investigating, missing that this training may be a universal need.
Keeping software updated is always good [25, 42].	Patches can cause new issues or failures in software.	Researchers should bear in mind that developers may rationally delay patching their code. Derr et al. suggested automated library updates as a solution to terrible update rates, noting that if the updates turn out to cause defects they can be rolled back [25]. If the defect is a vulnerability, the rollback may not occur on time to prevent exploitation; this concern is not discussed.
Safe defaults are always good [38, 54, 76, 93, 104].	Defaults presumed safe can cause unexpected consequences.	Before advocating for safe defaults, the relevant software's usage characteristics should be assessed. Advice such as Patnaik et al.'s based on 'deny access' [76] may not be appropriate for cyberphysical systems.
Secure coding regulations and compliance constraints improve software security [29].	Compliance requirements can have unintended consequences.	Careful consideration should be given to the wording and structure of legislation and regulations. Gasiba et al. expressed surprise that developers ignore secure coding instructions they consider unreasonable; that could be due to their making this assumption [29].
Developer Assumptions		
All developers want to code securely [86, 88, 109].	Some developers are uninterested in secure coding or feel that it is not relevant to them.	This assumption can cause researchers to misdiagnose reasons for insecure coding behaviour. Rahman et al. did not consider laziness or indifference as reasons for bypassing security tool warnings, although 85% of bypasses originated from 20% of developers [86]. This may have affected their analysis of the phenomena they observed.
All developers know how to code securely [108–110].	There is a very broad spectrum of code security knowledge and experience amongst software developers.	Researchers should not assume that developers know how to code securely. Weir et al. [109] did not investigate the secure coding stance or training needs of workshop participants, and may have missed important nuances.
Developers' perceptions on whether security is needed in their work environment are accurate [48].	Developers who are not well informed about software security may be poor judges of when it is required.	Researchers should not rely exclusively on developers' own assessments when judging whether software is vulnerable to attack. Lopez et al. appeared to take developers' assessments of security needs at face value [48]. They did not explore further as to whether the software produced was actually secure, or consider how that could be measured.

Table 1: Unhelpful Assumptions (continued)

Developers are enabled decision-makers not subject to psychological pressures like fear of downsizing [12].	Experience of downsizing events may impact developers' confidence and motivation.	Researchers should pay attention to developers' job security and the turbulence of their work environment when assessing software security motivation and practice. Assal and Chiasson's survey study might have detected disillusionment and fear as security blockers if relevant questions had been asked [12].
API and library developers are security experts [17, 25].	API and library developers have the same range of security expertise as other developers.	Researchers should not assume that API developers have any special security expertise. Derr et al. studied the difficulties inherent in keeping applications updated; their discussion on API developers' use of semantic versioning and bundling security updates did not consider that API developers might not understand the security implications of update policies [25].
All coders have access to the same resources [13, 89].	Constraints include background, training, isolation, the organisational secure-coding environment, and budget.	When analysing developers', teams' and organisations' security decisions, researchers should consider their budget and other resource constraints. Bai et al. did not assess budget or language skills when studying propagation of Stack Overflow security errors, which may have affected their analysis of issues [13].
Organisational Assumptions		
Managers are software security champions [88, 108–110].	Many managers are ill-informed about software security, fail to grasp the risk involved, and/or perceive it as unimportant.	When evaluating software security within an organisation, researchers should explicitly assess the stance of upper, middle and line management. For the software security interventions devised by Weir et al., upper management permission for the interventions is assumed [109]. This work would benefit from discussion of what would motivate upper management to permit security interventions in organisations that are inattentive to code security.
Organisations prioritise software security [72, 109].	Not all organisations prioritise software security.	Implications: researchers should pay attention to the priorities developers receive from their managers, both stated and unstated. Palombo et al., although noting that security fixes were sometimes resisted by developers and were often not implemented at all, did not explore the secure coding priorities of the organisation they were embedded in, or whether software security was in fact an organisational priority [72].
The organisational environment is cohesive and therefore can control the complexity of secure software development [9, 88].	In large organisations, different departments and teams may have different priorities, causing conflict over software security goals. Subcontractors may complicate the issue.	Researchers should pay attention to inter-departmental dynamics when studying large organisations. Arizon-Peretz et al.'s insightful paper on software security climate theory assumes that the organisation is cohesive [9]. It does not investigate issues of inter-departmental dynamics and power-plays, outsourcing, or fragmentation caused by frequent acquisitions, which could affect attempts to build a positive software security climate.
The presence of secure coding activities implies that security is a high priority within an organisation [48, 59].	Secure coding activities may be compliance driven, with little concern for actual security.	Researchers should assess the secure coding culture of an organisation as well as its secure coding activities. Morrison et al. observed secure coding practice in an IBM team but did not ask whether practices were ever skipped for deadline reasons [59].
Research Assumptions		
Developers and/or organisations will admit to a lack of understanding of, or enthusiasm for, security [82, 111].	Security is seen as a positive quality, and therefore both developers and organisations are unlikely to admit to a lack of understanding of it or interest in it.	Individuals' and organisations' assertions on prioritising software security should not be taken at face value. Poller et al. reported survey results that were positive or neutral about penetration testing [82]. Had they considered their own close relationship with management, they might have been more sceptical about these results.
Activities that can lead to insecure code cannot also lead to secure code [3, 13, 31].	Some common coding activities are widely used, and do not reflect the security of the resulting code.	Researchers should assess context of research findings, including whether control groups were used. Bai et al. found reuse of insecure code snippets from Stack Overflow in GitHub, resulting in insecure code [13]. However, they did not check for reuse of *secure* Stack Overflow code snippets, which could have revealed a positive effect.

Table 1: Unhelpful Assumptions (continued)

Findings in a single paper are final and can be cited confidently, with no need for replication [11, 14, 23, 32, 96, 107].	Scientific evidence requires confirmation, preferably from different types of study.	Researchers should not perceive research questions as definitively answered unless there is a cumulative body of research. Das Chowdhury et al. [23] uncritically cited a Van der Linden et al. study [103] finding that developers go by answer length when choosing a Stack Overflow answer, even though participants for that study were not vetted.
Measuring secure coding activities is equivalent to measuring code security [11, 70, 89].	Code security for non-trivial software cannot be definitively measured.	Researchers should bear in mind that the level of secure coding activities may not reflect actual code security. Oyetoyan et al. used a list of secure coding practices to assess secure coding in organisations [70]. They did not note that these practices may not reflect code security, and did not attempt to assess issues of culture such as deadline constraints, which may impact on whether the activities are thoroughly carried out [91].
If secure coding activities take place, they will be reflected in artefacts [43, 78, 99].	Artefacts do not always reflect work done.	Definitive conclusions should not be reached based on artefacts alone, although they can be used to triangulate. Imtiaz et al. studied use of the Coverity tool in open source projects [43]. One finding was that developers are slow to fix alerts and do not fix them based on tool-assigned priority level. It is possible that a coherent strategy, such as matching team member expertise to defect assignment, exists. It would not be available in Coverity history if so.
Artefact analysis done at scale is error-free [31, 43, 99].	Artefact analysis when done at scale may be flawed and results should be viewed with caution.	Researchers should carefully evaluate artefact analysis steps when considering results. Fischer et al. found a large percentage of insecure snippets in Android applications, using a complex pipeline to compare snippets to bytecode [31]. The paper did not discuss manual verification of the comparison process, and it is difficult to determine how this could be done, so the results may be open to question.
Coders, product managers, and testers can all be studied as a homogeneous group [47, 109].	Different disciplines need different types of support.	Researchers should consider the differing support needs of the different roles that they are studying. Lopez et al. assessed how developers responded to security episodes [47]. The paper veered between using 'developer' for all roles and coding roles, and may have missed an opportunity to distinguish responses and behaviour by role.

They concluded that this was partly due to the Agile development methodology, and that, in order to give developers some autonomy, managers had avoided giving precise security direction. However, where management genuinely prioritises security, it is made an overt rather than a non-functional requirement. For example, Whitmore and Tobin described work in IBM that was inspired by the need to facilitate developers' move towards rapid deployment via Agile and DevOps, demonstrating that where there is software security consciousness there can be a corresponding software security drive [113]. Tuladhar et al. showed that in an organisation that desires security, security stories may be added to the scrum [102]. Extra time may be allocated for security concerns [47].

This assumption should be avoided, unless it is possible to compare software security results to other teams in the same organisation that are not using Agile.

Implications: researchers working in an Agile environment should carefully consider whether the use of Agile methods is relevant to uncovered issues.

Assumption: keeping software updated is always good.

Reality: patches can cause new issues or failures in software.

Researchers have observed and criticised a tendency to delay updating and patching third-party libraries [25, 42]. The assumption is that keeping patches up-to-date maximises code security [42]. This is usually true, but sometimes new code or patches contain vulnerabilities or even malicious code [28]. Das Chowdhury et al. noted that the '*maintenance is fun*' trope they identified can lead researchers to expect unrealistically rigorous update behaviour. However, even security-aware professionals may prefer to wait a period after patches have been released before upgrading, to ensure that they are not exposed by vulnerabilities introduced in the patch [28]. This is rational behaviour and can only really be averted by ensuring that all upgrades are vulnerability free; not easy to do. One partial solution to this is the correct use of semantic versioning [17] to isolate security patches, so that developers can be sure that the patches will fix vulnerabilities and have the minimum possible additional effect on their code.

Implications: researchers should bear in mind that developers may rationally delay patching their code.

Assumption: safe defaults are always good.

Reality: defaults presumed safe can cause unexpected consequences.

Several studies in the review advocated provision of secure defaults [38, 54, 104]. While it seems safe to assume that safe, secure defaults are always beneficial to security, there can be exceptions. It is not possible to anticipate every use of a piece of software in

advance of releasing it [81]. Meng [54] advocated deprecating outdated encryption algorithms, but Gorski [35] noted that deprecation can entail locking API users to the version of the library that still contains the deprecated function, preventing them from receiving new updates. Naiakshina et al. [64] suggested that insecure options could be eradicated entirely, for example by removing deprecated hashing functions from encryption libraries. However, they noted that this could raise backward compatibility issues, potentially making applications even more insecure.

One of the recommendations often seen for secure development, dating at least from Saltzer and Schroeder's 1975 design principles, is that access decisions should be based '*on permission rather than exclusion*' [93], defaulting to denial of access. Massacci raised the issue of '*deny access*' being a potentially dangerous default response to authentication failure in cyberphysical systems [53]. He described a DDOS attack on the heating system of an apartment complex in Finland which caused freezing temperatures in the block for a week. A more deadly failure was the refusal of the software on a Boeing 737 Max plane to allow the pilots to override a dive triggered by a faulty sensor [53].

Implications: before advocating for safe defaults, the relevant software's usage characteristics should be assessed.

Assumption: secure coding regulations and compliance constraints improve software security.

Reality: compliance requirements can have unintended consequences.

One approach to improving software security is to introduce legislation and regulations to impose security requirements. While this might improve overall software security, especially for those organisations that currently do not consider software security at all [11], it can have unintended consequences. Vendors may in some cases choose to misclassify issues to avoid notification requirements, reducing the likelihood that the issues will be properly addressed [81]. In other situations, certification may be lost if a major change is made. For example, with FIPS 140-2 certification, fixing a security vulnerability may be classed as a major change and entail loss of certification. Organisations may sometimes be forced to choose between the secure option of fixing the vulnerability and the insecure option of retaining the certification [40].

Implications: careful consideration should be given to the wording and structure of legislation and regulations.

3.2 Developer Assumptions

Assumption: all developers want to code securely.

Reality: some developers are uninterested in secure coding or feel that it is not relevant to them.

Rauf et al. [88] explicitly made '*the assumption that developers have internalised these security goals, i.e. they have become their own goals, regardless of their origin*'. However, some developers in the studies they analysed explicitly expressed a lack of interest in security goals [11]. In their literature review, the authors catalogued the reasons why developers, despite having this assumed interest in secure coding, failed to code securely. A general potential for security enthusiasm was also assumed by Weir et al. in their secure coding interventions [109].

This assumption is one of the '*tropes*' identified by Das Chowdhury et al. [23]; they called it '*security is a universal priority*' and cited it as an indicator of '*a mismatch between what security experts think developers value and what they do in practice.*' Many of the studies in this review indicate that this assumption is false. Ashenden and Lawrence reported a developer saying '*that security practitioners "are like a rock in a stream, we just flow around you"*' [10]. In a study of the use of a tool for detecting checked-in secrets, Rahman et al. found that 51.4% of the time the commit proceeded with a one-time or a permanent bypass of the warning [86]. They found that 20% of developers generated 85% of bypasses, and speculated that these developers were unusually busy or security unaware. However, it is equally possible that the developers were lazy, or indifferent to security. Discussing software security training for engineers, Crowther and Rust said that '*our first experiments...failed because the training was boring and optional*' [20]. When embedded with a software team, Palombo et al. were taken aback to find that some vulnerabilities were considered by their fellow developers to be features, because they reduced support demands [72]. Patnaik et al. identified a Stack Overflow behaviour they called '*passing-the-buck*', where developers '*don't care about learning and just want to be told what to do*' [77].

Implications: this assumption can cause researchers to misdiagnose reasons for insecure coding behaviour.

Assumption: all developers know how to code securely.

Reality: there is a very broad spectrum of code security knowledge and experience amongst software developers.

Developers vary widely in their level of basic programming ability [1] and also in their degree of software security knowledge and training [30, 38]. Nevertheless, some studies implicitly perceive all developers as having an ability to understand code security, and to code securely, and do not delve into this question. This may be because developers can be reluctant to admit ignorance on any topic, particularly in group environments [46]. One example is the series of studies on security interventions in organisations by Weir et al. [108–110]. These studies included workshops where code security was discussed in a group setting. Within the workshops, there did not appear to be space to admit to ignorance of secure coding practice, or a desire for secure coding training. Given that other studies have revealed a high degree of ignorance amongst developers on software security [5, 63], this seems surprising. It is possible that developers are reluctant to reveal a feeling of ignorance when surrounded by their peers; this possibility was not explored, and exit interviews did not appear to ask about developers' feelings of competence. The '*blockers and motivators*' section in a paper on the long-term effects of the interventions [109] does not include any mention of developer ability, knowledge or need for training. This may be due to these omissions.

Contrasting with this apparent interchangeability of developers in a secure coding context is the widespread perception that secure coding in the work environment needs reinforcement from '*champions*:' people who are enthusiastic and knowledgeable about code security [33, 44, 98].

Implications: researchers should not assume that developers know how to code securely.

Assumption: developers' perceptions on whether security is needed in their work environment are accurate.

Reality: developers who are not well informed about software security may be poor judges of when it is required.

Developers are expert workers and are usually respected for their expertise. This can lead researchers to assume the correctness of developers' assessments of when security is needed. This appeared to be the case for Lopez et al. [48]. In an ethnographic study, they largely took developers' secure coding self-assessment at face value, missing an opportunity to dig deeper.

This assumption is related to the *'security is easy'* trope identified by Das Chowdhury et al. Developers can often suffer from the '*optimistic bias*,' which encourages them to believe that their software is too isolated, insignificant or obscure to be attacked [11]. The reality is that all software running on an online device can be targeted and exploited. Software that itself does not store significant data or perform significant activities may allow hostile actors to gain a foothold in a target's network or system [86]. Even software that does not run online may be leveraged by attackers if the host device is connected, using a technique known as '*living off the land*' [19]. Many developers are not familiar with hacking techniques and are poor judges of whether the software they are working on may be vulnerable to attack [86, 92]. Studies have found that developers can be over-confident in their secure coding abilities [64]. For example, Naiakshina et al. found that several of their participants encoded passwords with Base64, thinking that this made them secure [63]. Base64 changes text's appearance, but it is designed to aid binary data transfer, has no security properties, and is trivial to decode.

Implications: researchers should not rely exclusively on developers' own assessments when judging whether software is vulnerable to attack.

Assumption: developers are enabled decision-makers not subject to psychological pressures like fear of downsizing.

Reality: experience of downsizing events may impact developers' confidence and motivation.

The Agile principles assume that developers are enabled decision-makers, and it has been argued that true Agile depends on psychological safety [16]. Job insecurity can affect organisational citizenship behaviours [52]. Current studies of software security behaviour do not explicitly take developer vulnerability or organisational turbulence into account. Haney et al.'s study of organisations with good security culture found such indicators of job security and employer respect as mentoring, training, and long-term, committed staff [40]. By contrast, when Weir et al. introduced security interventions into organisations, one of the three teams they worked with ceased to exist during the three-month lifetime of their initial interventions [109]. One year later, only one of the initial interviewees still worked for the company. Institutional memory cannot survive the disappearance of an entire team.

As discussed, numerous studies cite the value of software security '*champions*'; individuals who embrace the software security message and spread it to their peers [40, 98, 102]. Most papers advise that there is no need to specifically recompense champions for their software security role, and recommend sending them to security conferences and organising activities for them as motivators. It is difficult to see how such champions can sustain organisational commitment in an adverse environment. Security culture is widely recognised as an important ingredient of a successful secure coding environment [40, 57, 91, 102]; such a culture depends on mutual trust and is unlikely to thrive in a hostile corporate environment.

Yet current studies of software security behaviour do not explicitly take developer vulnerability or organisational turbulence into account. Assal and Chiasson's excellent survey on secure coding motivation [12] asked numerous questions on motivators and blockers, but none concerned the effect of organisational turbulence or rounds of 'downsizing' on motivation. Anecdotally, and in the first author's extensive professional experience, such events affect employees' enthusiasm, loyalty and commitment to an organisation.

At time of writing in early 2023, the tech industry is in the throes of widespread layoffs of technical staff, including security staff. In this environment, the potential for good institutional secure-coding memory and sustained security practice and culture is not good. If developers work in institutions with a history of large-scale downsizing, or have experienced a mass-downsizing event, this may impact their belief that they can act autonomously. It may also affect their motivation to prioritise the organisation's best interests.

Implications: researchers should pay attention to developers' job security and the turbulence of their work environment when assessing software security motivation and practice.

Assumption: API and library developers are security experts.

Reality: API and library developers have the same range of security expertise as other developers.

Although API and library developers are, by definition, developers [23], some researchers seem to assume that they have more security expertise than developers. This may be because secure coding has been found to be associated with development experience in some studies [5, 70] (though not in others [63, 65, 69]). Many of the papers we reviewed dealt with API and library concerns such as documentation, abstraction level, level of support for developer testing and updates [36, 54, 61]. The implicit belief was that library developers understood, or should understand, the implications of their design decisions, and their impact on API users. Yet many libraries are developed by the open source community, where security guidance and policies may be more ad hoc [112].

Multiple studies show that library developers are not always better informed than other developers. Acar et al. noted that system developers' understanding of permissions on the Android ecosystem is not significantly better than that of other developers [1]. The issue of package updates in third-party libraries illustrates a similar failure by library developers to understand important security-related concepts; in this case, semantic versioning. Semantic versioning indicates whether a fix is major, minor, or a patch, and can be used by library developers to indicate the likelihood of an update breaking backwards compatibility [42]. Since some updates are security updates, it is important to encourage developers to

update libraries as frequently as possible. Developers are reluctant to update libraries because dependency updates can break code [75]. They would appreciate well-indicated security fixes that avoid breaking changes [75], but these are often not available. Chinthanet et al. found that over 90% of npm fixes studied were bundled with unrelated commits [17]. Library developers often fail to use semantic versioning correctly; this is a likely contributor to the reluctance of developers to update libraries [25].

Implications: researchers should not assume that API developers have any special security expertise.

Assumption: all coders have access to the same resources.

Reality: constraints include background, training, isolation, the organisational secure-coding environment, and budget.

Some studies implicitly assume that all developers have access to the same resources, and could code securely if they wanted to [13]. In fact, developers' environments vary widely and they may be subject to constraints that are not immediately apparent. The relatively isolated security environment of app developers was examined by van der Linden et al. [103] in a study of security decision-making. Working for an organisation is not necessarily better, as some organisations lack the positive security climate that facilitates developers who wish to code securely [8]. Developers constrained to use languages with characteristics that tend to cause high levels of vulnerabilities, such as C and C++ [73], are at a secure-coding disadvantage.

When it comes to issues of budgetary constraint, there is not much work available in the literature. Indeed, researchers themselves may be constrained; a study analysing and comparing static analysis tools only included free tools [97]. At a time when premium charges for basic security features like cloud-hosted logging is in the news [41], the effect of budgetary constraints should be considered by researchers.

Language comprehension may be another issue that affects some researchers. Votipka et al. found that 78% of the coding mistakes in their '*Build it, break it, fix it*' series of secure coding challenges were due to participants misunderstanding requirements [104]. Gorski et al., studying the embedding of security information in documentation for non-security APIs, noted that their recruits were German speakers, who might struggle with English documentation [37]. This has implications for language use and simplicity, and would be an interesting topic for further research.

Implications: when analysing developers', teams' and organisations' security decisions, researchers should consider their budget and other resource constraints.

3.3 Organisational Assumptions

Assumption: managers are software security champions.

Reality: many managers are ill-informed about software security, fail to grasp the risk involved, and/or perceive it as unimportant.

Implicit in discussions of how to introduce interventions into organisations is the assumption that management in the organisation will be in favour of increased software security [88, 108–110]. In reality, managers are not always aware of the importance of software security [82]. Senior managers, though paying lip service to software security, may perceive a diversion of resources to software security as impinging on more important goals [11, 47]. Project managers and product owners sometimes have little interest in security, and this can be exacerbated if they perceive that security is taken care of by someone else such as a security officer [100].

Implications: when evaluating software security within an organisation, researchers should explicitly assess the stance of upper, middle and line management.

Assumption: organisations prioritise software security.

Reality: not all organisations prioritise software security.

Throughout the literature, we observe an unstated assumption that organisations prioritise software security [72, 109]. Like maintainability and efficiency, software security is an NFR. While managers assume that NFRs will be present in software, they do not always explicitly require them [96]. This has led to an assumption that managers would require and prioritise software security if they thought of doing so. This assumption is not borne out by the evidence. It is rare, though not unknown [11], for managers to explicitly state that software security is not a priority. Actions signal priorities more clearly than words though, and many studies have found that security is not prioritised in practice [11, 57, 72]. Functionality usually takes top priority. Tight deadlines and inadequate staffing and training may result in decisions to release software that is likely to be insecure [101].

As discussed in Section 1, Palombo et al.'s ethnographic study found numerous secure development issues [72]. The researchers seemed to assume that the organisation prioritised security, but that it slipped through the cracks due to the development team being subject to conflicting priorities [72]. However, a close reading of the paper reveals a wholesale absence of management concern about software security. An absence of management support and instruction affects secure development emphasis [30]. Teams receive direction on priorities from the organisation they work in, and are unlikely to prioritise security if not asked to [9, 62–65].

Implications: researchers should pay attention to the priorities developers receive from their managers, both stated and unstated.

Assumption: the organisational environment is cohesive and therefore can control the complexity of secure software development.

Reality: in large organisations, different departments and teams may have different priorities, causing conflict over software security goals. Subcontractors may complicate the issue.

It is easy to assume that within an organisational environment, there is a coherent and cohesive software security policy. All aspects of the organisation move in concert. An example of such an environment appears in a study by Wang et al. describing an API hardening approach to XSS elimination at Google [106]. This was carefully designed and engineered to run on the enormous Google codebase, which is largely contained in a single repository.

However, in many large organisations software security policy may be fragmented. Security checks required by one department

may conflict with the goals of another department or unit [98]. Subcontractors complicate the issue [101]. Their goals are unlikely to be aligned with the long-term success of the organisation. They will generally focus on meeting tight deadlines within budget [57].

Implications: researchers should pay attention to inter-departmental dynamics when studying large organisations.

Assumption: the presence of secure coding activities implies that security is a high priority within an organisation.

Reality: secure coding activities may be compliance driven, with little concern for actual security.

Assal and Chiasson [11] evaluated 13 teams on whether they undertook security activities at different stages of the software development life cycle. From their results, they identified '*security inattentive*' and '*security adopter*' organisations. All of the organisations they studied fell firmly into either one group or the other. However, the existence of the '*checkbox*' phenomenon is well documented [10, 40, 84, 100]. This is a tendency of an organisation to undertake software security activities for compliance reasons, without an inherent security concern. Studies have shown that some organisations will do the minimum work necessary to pass known checks; for example, one study found that organisations evaluating PCI compliance did not test widely and did not vary their tests, allowing client organisations to limit security work [84]. This mindset is not universal. Enck and Williams identified considerable security energy amongst organisations debating supply chain security, despite compliance weariness [28].

The absence of secure coding activities is an indicator that an organisation is security inattentive, but the inverse is not necessarily true. In the absence of a security culture, organisations may undertake a minimal amount of code assurance. This can lead to trouble as the software security problem will not have the drivers it needs [91]. One suggested solution to this issue is to use organisational climate theory to improve the climate for software security within the work environment [9].

Implications: researchers should assess the secure coding culture of an organisation as well as its secure coding activities.

3.4 Research Assumptions

Assumption: developers and/or organisations will admit to a lack of understanding of, or enthusiasm for, security.

Reality: security is seen as a positive quality, and therefore both developers and organisations are unlikely to admit to a lack of understanding of it or interest in it.

Developers may be reluctant, particularly in group interviews that include colleagues, to admit to a lack of knowledge about software security [46]. There may be power dynamics and group history at play. The same is true of admitting to a lack of interest. Most organisations are reluctant to admit to corporate failings [105]. The chances of their acknowledging that they are ignoring software security are slim. Thus it is important to probe and explore further, rather than taking statements on software security at face value.

In one study where the researchers had close relationships with management, negative feelings towards security may have been suppressed by the development team in survey entries for fear of management finding out [82]. Several people chose '*neutral*' on security matters, which seemed equivocal. Weir et al. found a poor correlation between survey respondents' claim to use assurance techniques and the actual security of apps they had written [111]. They did not appear to consider that the self-reported use of assurance techniques could have been exaggerated.

Implications: individuals' and organisations' assertions on prioritising software security should not be taken at face value.

Assumption: activities that can lead to insecure code cannot also lead to secure code.

Reality: some common coding activities are widely used, and do not reflect the security of the resulting code.

This assumption is made in a study of the use of copy-and-paste from Stack Overflow and other online forums [13]. Acar et al., in a study comparing different information sources during coding, found that use of copy-and-paste from Stack Overflow resulted in insecure code [3]. However, Naiakshina et al. did a qualitative evaluation of participants' development process during their studies on secure password storage [65]. They found that all participants who produced secure solutions used copy-and-paste from an online location. In a study on the usability of five different cryptographic APIs, Acar et al. found that there was a statistically significant correlation between the use of copy-and-paste and the functional correctness of a task, but there was no such correlation between copy-and-paste and the security of the solution [2]. This shows that use of copy-and-paste helps developers to achieve functional correctness. If they are sufficiently competent to produce functionally correct code they are likely to be using copy-and-paste. They will continue to use it while trying to secure the code; this does not affect the security of the code.

Bai et al. searched for instances of known cryptographic errors on Stack Overflow and matched them to code in GitHub projects [13]. For projects where they found a match, they invited the developers for a survey and interview. One of the limitations listed in the paper was that they did not include a control group. Such a group would be difficult to identify; it is impossible to prove that software is secure; a new defect could be found tomorrow. But a group could have been approximated by identifying secure code snippets in Stack Overflow, matching them to GitHub projects in the same way as was done for insecure snippets. This might have revealed that a good deal of secure code in GitHub originated with Stack Overflow.

Implications: researchers should assess context of research findings, including whether control groups were used.

Assumption: findings in a single paper are final and can be cited confidently, with no need for replication.

Reality: scientific evidence requires confirmation, preferably from different types of study.

The findings of a single study are not conclusive. Findings should be replicated, or even better, reached in multiple ways by triangulation [60]. However, several well-designed studies in the DCS field are routinely referenced with no reservations as to whether their results have been reproduced elsewhere. For example, the several

papers based on a 2014 study of security tool adoption [114, 115] appear to be considered definitive. While not wishing to detract from that original work, we suggest that further investigation into this phenomenon could shed new light on security tool adoption and use. Similarly, work by Oliveira et al. on API blindspots is frequently cited [69]. This study found a correlation between the personality trait of openness and an ability to fix security issues in the presence of blindspots, suggesting novel recruitment strategies could be developed based on this finding. Given the potential impact on candidates of filtering recruits by their level of openness, an attempt at confirming the finding appears to be indicated.

In a good example of confirming findings, Naiakshina et al. and Danilova et al. conducted a series of studies on secure password storage with different developer populations including students, freelancers and professional developers [21, 62–65]. The studies consistently and markedly found that explicitly asking for security has a strong impact on whether developers attempt to code securely.

Implications: researchers should not perceive research questions as definitively answered unless there is a cumulative body of research.

Assumption: measuring secure coding activities is equivalent to measuring code security.

Reality: code security for non-trivial software cannot be definitively measured.

It is relatively easy to confirm that a piece of software is **in**secure; one need only find a vulnerability. By contrast, it is impossible to confirm that complex software is secure. The best that can be done is to confirm that to date, it has no known vulnerabilities. There was only one study in the review that attempted to assess the actual security of software [111]; its results were inconclusive. Absent a widely-embraced standard way to assess secure coding in an organisation, many researchers have devised measures of their own [11, 58, 70, 89]. These are all based on the performance of widely-recognised software security assurance activities. Such measures are usually recreated by each group of researchers and tend not be reused across multiple studies. The Ryan metric, a lightweight measure based on empirical data, was proposed by Ryan et al. [91]. Whatever measure is used, researchers should avoid conflating use of security assurance techniques with code security.

Implications: researchers should bear in mind that the level of secure coding activities may not reflect actual code security.

Assumption: if secure coding activities take place, they will be reflected in artefacts.

Reality: artefacts do not always reflect work done.

Several studies that examined artefacts such as source control comments for evidence of whether activities were performed [43, 78, 99] were included in the review. This approach will not always find evidence of activities even if they are in fact performed; there is not necessarily a correspondence between how frequently something is done and how frequently it appears in artefacts. Morrison et al. [59] noted that their artefact analysis did not reliably reflect the use of techniques that they had confirmed via a survey and in qualitative analysis. Palombo et al. found that developers rarely used security-related terms in defect ticket descriptions. They had to manually correlate tickets with implementation code [72].

Implications: definitive conclusions should not be reached based on artefacts alone, although they can be used to triangulate.

Assumption: artefact analysis done at scale is error-free.

Reality: artefact analysis when done at scale may be flawed and results should be viewed with caution.

Large-scale artefact analysis was used by a number of interesting studies in the review. Since it uses automated analysis tools, usually created by the researchers, this method can be subject to large-scale analysis failure that remains undetected. Good research in this area [85] will include manual checking of the results of automated tools, and will acknowledge threats to validity [42].

Thompson and Wagner attempted to assess the correlation between code review and security vulnerabilities in 3,126 GitHub projects using 143 languages [99]. They used metadata analysis techniques to determine code review status and machine learning to identify security vulnerabilities. They found a weak negative correlation between code review and numbers of both '*all issues*' and '*security issues*.' The techniques used to identify issues did not ascertain issue severity, limiting the usefulness of the results for helping to decide on code review strategies. The authors were forced to make a number of approximations and assumptions to determine what is a code review and what is a security defect when analysing source code at large scale. They made a good attempt at both objectives and gave detailed explanations of their approach. This makes the paper a valuable learning tool, but a degree of scepticism should be applied to the results.

Fischer et al. developed a pipeline to match security-related code snippets in Stack Overflow (identified using machine learning) to applications from Google Play [31]. They stated that 15.4% of the Android applications contained security-related code snippets from Stack Overflow. 97.9% of these applications contained insecure snippets. However, from reading the paper, it was difficult to determine how much verification of the Android snippet detection they had done or been able to do.

Paul et al. looked for factors that impacted on the success of code reviews in finding security vulnerabilities in the Chromium project [78]. They compared attributes of code reviews that successfully detected vulnerabilities to attributes of code reviews that failed to detect vulnerabilities that were subsequently discovered. The study made use of automation to find relevant commits, with extensive checking by the authors to verify results. The result was a convincing paper finding that the time taken to complete a review, whether it was a bug fix review, and the number of mutual reviews between two developers, had a positive impact on the probability that a vulnerability would be found.

Imtiaz et al. studied five open source projects that had used static analysis tool Coverity for at least five years, attempting to answer six research questions such as how developers use the tool and what their fix rate is [43]. They found that the developers fixed between 27.4% and 49.5% of alerts. They also found that developers may take a long time to fix the alerts. Since all their analysis was done via automated processing of commits and commit messages, they could not provide any insights into their findings beyond conjecture.

Implications: researchers should carefully evaluate artefact analysis steps when considering results.

Assumption: coders, product managers, and testers can all be studied as a homogeneous group.

Reality: different disciplines need different types of support.

This assumption was made by Weir et al. [109] when introducing security interventions. Interventions were introduced to teams, with team members not differentiated by function. Similarly, in an ethnographic study examining how organisations respond to security events, Lopez et al. included programmers, testers, designers and product managers in their opening definition of '*software developers*' [47]. While conflating all these roles may make sense in some contexts, in others this assumption is unhelpful.

Project managers and product owners may have little interest in security, and may need training or support to engage with software security [100]. Studies that overtly look at all of these roles but implicitly focus on coding without distinguishing other functions are likely to miss important nuances. Coders and their managers do not have the same challenges and demands [44]. They need different training and support [82].

Implications: researchers should consider the differing support needs of the different roles that they are studying.

4 DISCUSSION AND CONCLUSION

This paper revealed a considerable number of assumptions in prior literature focusing on developer centred security, and identified by the first author who has extensive practical software security experience. We note that such assumptions can only be identified using the reader's knowledge, experience, grasp of the field and familiarity with other research, as described in the positionality statement in Section 2.5, and that additional assumptions could be present which we did not identify. We now discuss the implications of our findings organised by category of assumptions. Researchers that study secure coding could use the list of assumptions as a '*checklist*' during study design, to ensure that the study does not inadvertently rely on any of these assumptions.

Technical Assumptions. Technical assumptions are those that make generalising claims regarding the nature of code, processes, and tools. These assumed generalised truisms are problematic because they oversimplify reality, which means that studies may be based on assumptions that do not generally hold, and that findings cannot be generalised to other settings. In other words, context matters, and 'it depends.' What factors study findings may depend on is something to consider in designing future studies.

Developer Assumptions. Developer assumptions are those that make generalising claims regarding the developers who are expected to produce secure software. For example, studies assume that developers *want* to produce secure software (i.e., they care about it), or that they *know how to*. Other assumptions include those that developers have considerable decision power in organisations, or that they are well resourced to develop secure code. Clearly, these assumptions do not hold across organisations, and so future studies should consider these aspects, for example, by investigating developers' training, context and resources. It may very well be that a major obstacle to secure code is uninterested or under-resourced developers, rather than a missing process.

Organisational Assumptions. Organisational assumptions are those that make generalising claims regarding the nature of organisations and their management. From the secure software researcher's perspective, secure code is a key topic of interest. It is clear from our findings that this is not always the case in organisations. However, no organisation will ever acknowledge that secure code is not a top priority, therefore we must scratch below the surface. Having secure coding policies in place is not an indicator of an organisation's true commitment to delivering secure code. Future studies, for example ethnographic studies, may be able to observe the contrast between what organisations and managers say, and what they do on a day-to-day basis.

Research Assumptions. The fourth category, Research Assumptions, are assumptions that tend to result in generalising statements regarding organisations, study findings, observations, artefacts, and stakeholders. Some of the assumptions are not specific per se to secure coding research. For example, the assumption that a study's findings are '*final*,' in that they do not need to be replicated or put into context, is a general issue within software development research. Other assumptions relate to how researchers operationalise concepts under study. For example, studies that seek to measure code security might consider secure coding activities, but clearly they are not the same; the proof is in the pudding, not the recipe.

Conclusion In this paper we discussed a number of unhelpful implicit assumptions that can adversely affect software security research. We outlined cases where the assumptions appeared to hold and discussed the reality that they obscured. We considered the implications for researchers when considering the production of secure software.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their time and insights. This work is supported by Science Foundation Ireland grants 18/CRT/6222, 13/RC/2077_P2, 13/RC/2094_P2, 15/SIRG/3293.

REFERENCES

[1] Y. Acar, M. Backes, S. Bugiel, S. Fahl, P. McDaniel, and M. Smith. 2016. SoK: Lessons Learned from Android Security Research for Appified Software Platforms. In *IEEE Symp. on Security and Privacy*. 433–451.
[2] Y. Acar, M. Backes, S. Fahl, S. Garfinkel, D. Kim, M. L. Mazurek, and C. Stransky. 2017. Comparing the Usability of Cryptographic APIs. In *IEEE Symp. on Security and Privacy*. 154–171.
[3] Y. Acar, M. Backes, S. Fahl, D. Kim, M. L. Mazurek, and C. Stransky. 2016. You Get Where You're Looking for: The Impact of Information Sources on Code Security. In *IEEE Symp. on Security and Privacy*. 289–305.
[4] Y. Acar, S. Fahl, and M. L. Mazurek. 2016. You are Not Your Developer, Either: A Research Agenda for Usable Security and Privacy Research Beyond End Users. In *IEEE Cybersecur. Dev.* 3–8.
[5] Y. Acar, C. Stransky, D. Wermke, M. L Mazurek, and S. Fahl. 2017. Security Developer Studies with GitHub Users: Exploring a Convenience Sample. In *13th Symp. Usable Priv. Secur.* 81–95.
[6] Y. Acar, C. Stransky, D. Wermke, C. Weir, M. L. Mazurek, and S. Fahl. 2017. Developers Need Support, Too: A Survey of Security Advice for Software Developers. In *IEEE Cybersecur. Dev.* 22–26.
[7] A. AlJarrah and M. Shehab. 2017. The Demon is in the Configuration: Revisiting Hybrid Mobile Apps Configuration Model. In *12th Int. Conf. on Availability, Reliability and Security*. ACM, New York, NY, USA.
[8] R. Arizon-Peretz, I. Hadar, and G. Luria. 2022. The Importance of Security Is in the Eye of the Beholder: Cultural, Organizational, and Personal Factors Affecting the Implementation of Security by Design. *IEEE Trans. Softw. Eng.* 48, 11 (2022), 4433–4446.

[9] R. Arizon-Peretz, I. Hadar, G. Luria, and S. Sherman. 2021. Understanding developers privacy and security mindsets via climate theory. *Empirical Software Engineering* 26, 6 (2021).

[10] D. Ashenden and D. Lawrence. 2016. Security Dialogues: Building Better Relationships between Security and Business. *IEEE Secur Priv* 14, 3 (2016), 82–87.

[11] H. Assal and S. Chiasson. 2018. Security in the Software Development Lifecycle. In *14th USENIX Conf. Usable Priv. Secur.* USA, 281–296.

[12] H. Assal and S. Chiasson. 2019. "Think secure from the beginning": A Survey with Software Developers. In *Conf. Hum. Factors Comput. Syst.*

[13] W. Bai, O. Akgul, and M. L. Mazurek. 2019. A Qualitative Investigation of Insecure Code Propagation from Online Forums. In *IEEE Cybersecur. Dev.* 34–48.

[14] A. Z. Baset and T. Denning. 2017. IDE Plugins for Detecting Input-Validation Vulnerabilities. In *IEEE Security and Privacy Workshops*. 143–146.

[15] S. Berkovich, J. Kam, and G. Wurster. 2020. UBCIS: Ultimate Benchmark for Container Image Scanning. In *13th USENIX Workshop on Cyber Security Experimentation and Test*.

[16] M. P. Buvik and A. Tkalich. 2022. Psychological Safety in Agile Software Development Teams: Work Design Antecedents and Performance Consequences. In *Proc. of the 55th Hawaii Int. Conf. on System Sciences*.

[17] B. Chinthanet, R.G. Kula, S. McIntosh, T. Ishio, A. Ihara, and K. Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* 26, 3 (2021).

[18] M. Christakis and C. Bird. 2016. What developers want and need from program analysis: An empirical study. In *ASE 2016 - 31st IEEE/ACM Int. Conf. on Automated Software Engineering*. 332–343.

[19] Crowdstrike. 2023. What Are Living off the Land (LOTL) Attacks? https://www.crowdstrike.com/cybersecurity-101/living-off-the-land-attacks-lotl/.

[20] K. G. Crowther and B. Rust. 2020. Built-in cybersecurity: Insights into product security for cyberphysical systems at a large company. *IEEE Secur Priv* 18, 5 (2020), 74–79.

[21] A. Danilova, A. Naiakshina, J. Deuter, and M. Smith. 2020. Replication: On the ecological validity of online security developer studies: Exploring deception in a password-storage study with freelancers. In *16th Symp. Usable Priv. Secur.* 165–184.

[22] A. Danilova, A. Naiakshina, and M. Smith. 2020. One size does not fit all: a grounded theory and online survey study of developer preferences for security warning types. In *ACM/IEEE 42nd Int. Conf. Softw. Eng.* 136–148.

[23] P. Das Chowdhury, J. Hallett, N. Patnaik, M. Tahaei, and A. Rashid. 2021. Developers are Neither Enemies Nor Users: They are Collaborators. In *IEEE Secure Development Conf.* 47–55.

[24] N. Davidsson, A. Pawlowski, and T. Holz. 2019. Towards Automated Application-Specific Software Stacks. In *Lecture Notes in Computer Science*. Springer, 88–109.

[25] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes. 2017. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *ACM SIGSAC Conf. on Computer and Communications Security*. 2187–2200.

[26] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn. 2019. Scaling static analyses at Facebook. *Commun. ACM* 62, 8 (2019), 62–70.

[27] W. El-Hajj, G. Ben Brahim, H. Hajj, H. Safa, and R. Adaimy. 2016. Security-by-construction in web applications development via database annotations. *Computers & Security* 59 (2016), 151–165.

[28] W. Enck and L. Williams. 2022. Top Five Challenges in Software Supply Chain Security: Observations From 30 Industry and Government Organizations. *IEEE Secur Priv* 20, 2 (2022), 96–100.

[29] T. Espinha Gasiba, I. Andrei-Cristian, U. Lechner, and M. Pinto-Albuquerque. 2021. Raising Security Awareness of Cloud Deployments using Infrastructure as Code through CyberSecurity Challenges. In *The 16th Int. Conf. on Availability, Reliability and Security*. ACM, 1–8.

[30] T. Espinha Gasiba, U. Lechner, M. Pinto-Albuquerque, and D. Méndez. 2021. Is Secure Coding Education in the Industry Needed? An Investigation Through a Large Scale Survey. In *IEEE/ACM 43rd Int. Conf. Software Engineering*. 241–252.

[31] F. Fischer, K. Böttinger, H. Xiao, C. Stransky, Y. Acar, M. Backes, and S. Fahl. 2017. Stack Overflow Considered Harmful? The Impact of Copy Paste on Android Application Security. In *IEEE Symp. on Security and Privacy*. 121–136.

[32] F. Fischer, Y. Stachelscheid, and J. Grossklags. 2021. The Effect of Google Search on Software Security: Unobtrusive Security Interventions via Content Re-ranking. In *ACM SIGSAC Conf. on Computer and Communications Security*. New York, NY, USA, 3070–3084.

[33] K. R. Fulton, D. Votipka, D. Abrokwa, M. L. Mazurek, M. Hicks, and J. Parker. 2022. Understanding the How and the Why: Exploring Secure Development Practices through a Course Competition. In *ACM SIGSAC Conf. on Computer and Communications Security*. 1141–1155.

[34] L. Geierhaas, A.-M. Ortloff, M. Smith, and A. Naiakshina. 2022. Let's Hash: Helping Developers with Password Security. In *18th Symp. Usable Priv. Secur.* 503–522.

[35] P. L. Gorski, Y. Acar, L. Lo Iacono, and S. Fahl. 2020. Listen to Developers! A Participatory Design Study on Security Warnings for Cryptographic APIs. In *Conf. Hum. Factors Comput. Syst.* ACM, New York, NY, USA.

[36] P. L. Gorski, L. Lo Iacono, D. Wermke, C. Stransky, S. Moeller, Y. Acar, and S. Fahl. 2018. Developers Deserve Security Warnings, Too: On the Effect of Integrated Security Advice on Cryptographic API Misuse. In *14th Symp. Usable Priv. Secur.* 265–281.

[37] P. L. Gorski, S. Moller, S. Wiefling, and L. Lo Iacono. 2021. "I just looked for the solution!" - on integrating security-relevant information in non-security API documentation to support secure coding practices. *IEEE Trans. Softw. Eng.* 48, 9 (2021).

[38] M. Green and M. Smith. 2016. Developers are Not the Enemy!: The Need for Usable Security APIs. *IEEE Secur Priv* 14, 5 (2016), 40–46.

[39] T. Hærem, B. T. Pentland, and K. D. Miller. 2015. Task Complexity: Extending a Core Concept. *Acad Manage Rev* 40, 3 (2015), 446–460.

[40] J. Haney, M. Theofanos, Y. Acar, and S. Spickard Prettyman. 2018. "We make it a big deal in the company": Security Mindsets in Organizations that Develop Cryptographic Products. In *14th Symp. Usable Priv. Secur.* USENIX Assoc., 357–373.

[41] S. Hendery. 2023. Email hack prompts call for Microsoft to make security logs free. https://www.scmagazine.com/analysis/email-hack-prompts-call-for-microsoft-to-make-security-logs-free.

[42] K. Huang, B. Chen, C. Xu, Y. Wang, B. Shi, X. Peng, Y. Wu, and Y. Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empirical Software Engineering* 27, 4 (2022).

[43] N. Imtiaz, B. Murphy, and L. Williams. 2019. How Do Developers Act on Static Analysis Alerts? An Empirical Study of Coverity Usage. In *IEEE 30th Int. Symp. on Software Reliability Engineering*. 323–333.

[44] M. G. Jaatun and D. Soares Cruzes. 2021. Care and Feeding of Your Security Champion. In *Int. Conf. on Cyber Situational Awareness, Data Analytics and Assessment*. IEEE, 1–7.

[45] J. Jancar, M. Fourné, D. De Almeida Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar. 2022. "They're not that hard to mitigate": What Cryptographic Library Developers Think About Timing Attacks. In *IEEE Symp. on Security and Privacy*. 632–649.

[46] R. A. Krueger and M. A. Casey. 2000. *Focus Groups, A Practical Guide for Applied Research* (5 ed.). SAGE Publications.

[47] T. Lopez, H. Sharp, T. Tun, A. Bandara, M. Levine, and B. Nuseibeh. 2022. Security Responses in Software Development. *ACM Trans Soft Eng Methodol* (2022).

[48] T. Lopez, H. Sharp, T. Tun, A. K. Bandara, M. Levine, and B. Nuseibeh. 2019. "Hopefully We Are Mostly Secure": Views on Secure Code in Professional Practice. In *12th Int. Workshop on Cooperative and Human Aspects of Software Engineering*. IEEE, 61–68.

[49] T. Lopez, T. T. Tun, A. K. Bandara, M. Levine, B. Nuseibeh, and H. Sharp. 2020. Taking the Middle Path: Learning About Security Through Online Social Interaction. *IEEE Software* 37, 1 (2020), 25–30.

[50] T. Loruenser, H. C. Pöhls, L. Sell, and T. Laenger. 2018. CryptSDLC: Embedding Cryptographic Engineering into Secure Software Development Lifecycle. In *13th Int. Conf. on Availability, Reliability and Security*. ACM.

[51] L. Luo, M. Schäf, D. Sanchez, and E. Bodden. 2021. IDE support for cloud-based static analyses. In *29th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*. 1178–1189.

[52] A. B. Mahmoud, W. D. Reisel, L. Fuxman, and I. Mohr. 2021. A motivational standpoint of job insecurity effects on organizational citizenship behaviors: A generational study. *Scandinavian Journal of Psychology* 62, 2 (2021), 267–275.

[53] F. Massacci. 2019. Is "Deny Access" a Valid "Fail-Safe Default" Principle for Building Security in Cyberphysical Systems? *IEEE Secur Priv* 17, 5 (2019), 90–93.

[54] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. A. Argoty. 2018. Secure coding practices in Java: challenges and vulnerabilities. In *ACM/IEEE 40th Int. Conf. Softw. Eng.* 372–383.

[55] V. Mohan and L. B. Othmane. 2016. SecDevOps: Is It a Marketing Buzzword? - Mapping Research on Security in DevOps. In *11th Int. Conf. on Availability, Reliability and Security*. 542–547.

[56] A. Mokhberi and K. Beznosov. 2021. SoK: Human, Organizational, and Technological Dimensions of Developers' Challenges in Engineering Secure Software. In *European Symp. on Usable Security*. ACM, 59–75.

[57] J. A. Morales, T. P. Scanlon, A. Volkmann, J. Yankel, and H. Yasar. 2020. Security Impacts of Sub-Optimal DevSecOps Implementations in a Highly Regulated Environment. In *15th Int. Conf. on Availability, Reliability and Security*. ACM.

[58] P. Morrison. 2015. A security practices evaluation framework. In *37th Int. Conf. Softw. Eng.* IEEE, 935–938.

[59] P. Morrison, B. H. Smith, and L. Williams. 2017. Measuring Security Practice Use: A Case Study at IBM. In *IEEE/ACM 5th Int. Workshop on Conducting Empirical Studies in Industry*. 16–22.

[60] M. R. Munafò and G. Davey Smith. 2018. Robust research needs many lines of evidence. *Nature* 553, 7689 (2018), 399–401.

[61] S. Nadi, S. Krüger, M. Mezini, and E. Bodden. 2016. Jumping through hoops: why do Java developers struggle with cryptography APIs?. In *38th Int. Conf. Softw. Eng.* ACM, 935–946.

[62] A. Naiakshina, A. Danilova, E. Gerlitz, and M. Smith. 2020. On Conducting Security Developer Studies with CS Students: Examining a Password-Storage

Study with CS Students, Freelancers, and Company Developers. In *Conf. Hum. Factors Comput. Syst.* ACM.
[63] A. Naiakshina, A. Danilova, E. Gerlitz, E. von Zezschwitz, and M. Smith. 2019. "If you want, I can store the encrypted password": A Password-Storage Field Study with Freelance Developers. In *Conf. Hum. Factors Comput. Syst.* ACM.
[64] A. Naiakshina, A. Danilova, C. Tiefenau, M. Herzog, S. Dechand, and M. Smith. 2017. Why Do Developers Get Password Storage Wrong? A Qualitative Usability Study. In *ACM Conf. on Computer and Communications Security*. 311–328.
[65] A. Naiakshina, A. Danilova, C. Tiefenau, and M. Smith. 2018. Deception Task Design in Developer Password Studies: Exploring a Student Sample. In *14th Symp. Usable Priv. Secur.* USENIX Assoc., 297–313.
[66] D.C. Nguyen, D. Wermke, Y. Acar, M. Backes, C. Weir, and S. Fahl. 2017. A Stitch in Time: Supporting Android Developers in Writing Secure Code. In *ACM SIGSAC Conf. on Computer and Communications Security*. 1065–1077.
[67] L. Nguyen Quang Do and E. Bodden. 2018. Gamifying static analysis. In *26th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering*. ACM, 714–718.
[68] M. Ohm, A. Sykosch, and M. Meier. 2020. Towards Detection of Software Supply Chain Attacks by Forensic Artifacts. In *15th Int. Conf. on Availability, Reliability and Security*. ACM.
[69] D. S. Oliveira, T. Lin, M. S. Rahman, R. Akefirad, D. Ellis, E. Perez, R. Bobhate, L. A. DeLong, J. Cappos, and Y. Brun. 2018. API Blindspots: Why Experienced Developers Write Vulnerable Code. In *14th Symp. Usable Priv. Secur.* 315–328.
[70] T. D. Oyetoyan, D. S. Cruzes, and M. G. Jaatun. 2016. An Empirical Study on the Relationship between Software Security Skills, Usage and Training Needs in Agile Settings. In *11th Int. Conf. on Availability, Reliability and Security*. 548–555.
[71] T. D. Oyetoyan, B. Milosheska, M. Grini, and D. Soares Cruzes. 2018. Myths and facts about static application security testing tools: an action research at Telenor digital. In *Agile Processes in Software Engineering and Extreme Programming*. Springer, 86–103.
[72] H. Palombo, A.Z. Tabari, D. Lende, J. Ligatti, and X. Ou. 2020. An Ethnographic Understanding of Software (In)Security and a Co-Creation Model to Improve Secure Software Development. In *16th Symp. Usable Priv. Secur.* 205–220.
[73] J. Parker, M. Hicks, A. Ruef, M. L. Mazurek, D. Levin, D. Votipka, P. Mardziel, and K. R. Fulton. 2020. Build It, Break It, Fix It: Contesting Secure Development. *ACM Transactions on Privacy and Security* 23, 2 (2020), 10:1–10:36.
[74] D. Parsons, T. Susnjak, and A. Mathrani. 2015. The Software Developer Cycle: Career demographics and the market clock: or, is SQL the new COBOL?. In *24th Australasian Software Engineering Conference*. ACM, 86–90.
[75] I. Pashchenko, D.-L. Vu, and F. Massacci. 2020. A Qualitative Study of Dependency Management and its Security Implications. In *ACM SIGSAC Conf. on Computer and Communications Security*. 1513–1531.
[76] N. Patnaik, A. C. Dwyer, J. Hallett, and A. Rashid. 2022. SLR: From Saltzer & Schroeder to 2021...47 Years of Research on the Development and Validation of Security API Recommendations. *ACM Trans Soft Eng Methodol* (2022).
[77] N. Patnaik, J. Hallett, and A. Rashid. 2019. Usability Smells: An Analysis of Developers' Struggle With Crypto Libraries. In *15th Symp. Usable Priv. Secur.* 245–257.
[78] R. Paul, A. K. Turzo, and A. Bosu. 2021. Why Security Defects Go Unnoticed During Code Reviews? A Case-Control Study of the Chromium OS Project. In *IEEE/ACM 43rd Int. Conf. Softw. Eng.* 1373–1385.
[79] A. Pereira-Vale, E. B. Fernandez, R. Monge, H. Astudillo, and G. Marquez. 2021. Security in microservice-based systems: A Multivocal literature review. *Comput Secur* 103 (2021).
[80] A. Petrosyan. 2023. Number of common IT security vulnerabilities and exposures (CVEs) worldwide from 2009 to 2023 YTD. https://www.statista.com/statistics/500755/worldwide-common-vulnerabilities-and-exposures/.
[81] O. Pieczul, S. Foley, and M. E. Zurko. 2017. Developer-centered security and the symmetry of ignorance. In *New Security Paradigms Workshop*. ACM, 46–56.
[82] A. Poller, L. Kocksch, K. Kinder-Kurlanda, and F. A. Epp. 2016. First-time Security Audits as a Turning Point?: Challenges for Security Practices in an Industry Software Development Team. In *CHI Conf. Extended Abstracts on Hum Factors Comput. Syst.* ACM, 1288–1294.
[83] A. Poller, L. Kocksch, S. Türpe, F. A. Epp, and K. Kinder-Kurlanda. 2017. Can Security Become a Routine? A Study of Organizational Change in an Agile Software Development Group. In *ACM Conf. on Computer Supported Cooperative Work and Social Computing*. ACM, New York, NY, USA, 2489–2503.
[84] S. Rahaman, G. Wang, and D. Yao. 2019. Security Certification in Payment Card Industry: Testbeds, Measurements, and Recommendations. In *ACM SIGSAC Conf. on Computer and Communications Security*. New York, NY, USA, 481–498.
[85] A. Rahman, C. Parnin, and L. Williams. 2019. The Seven Sins: Security Smells in Infrastructure as Code Scripts. In *IEEE/ACM 41st Int. Conf. Softw. Eng.* 164–175.
[86] M.R. Rahman, N. Imtiaz, M.-A. Storey, and L. Williams. 2022. Why secret detection tools are not enough: It's not just about false positives - An industrial case study. *Empirical Software Engineering* 27, 3 (2022).
[87] P. Rajba. 2018. Challenges and mitigation approaches for getting secured applications in an enterprise company. In *13th Int. Conf. on Availability, Reliability and Security*. ACM.
[88] I. Rauf, M. Petre, T. Tun, T. Lopez, P. Lunn, D. Van der Linden, J. Towse, H. Sharp, M. Levine, A. Rashid, and B. Nuseibeh. 2021. The Case for Adaptive Security Interventions. *ACM Trans Soft Eng Methodol* 31, 1 (2021).
[89] K. Rindell, J. Ruohonen, and S. Hyrynsalmi. 2018. Surveying Secure Software Development Practices in Finland. In *13th Int. Conf. on Availability, Reliability and Security*. ACM.
[90] I. Ryan, U. Roedig, and K.-J. Stol. 2022. Insecure Software on a Fragmenting Internet. In *Cyber Research Conf. - Ireland (Cyber-RCI)*. 1–9.
[91] I. Ryan, U. Roedig, and K.-J. Stol. 2023. Measuring Secure Coding Practice and Culture: A Finger Pointing at the Moon is not the Moon. In *IEEE/ACM 45th Int. Conf. Softw. Eng.*
[92] M. Sahin, T. Ünlü, C. Hébert, L. A. Shepherd, N. Coull, and C. Mc Lean. 2022. Measuring Developers' Web Security Awareness from Attack and Defense Perspectives. In *IEEE Security and Privacy Workshops*. 31–43.
[93] J.H. Saltzer and M.D. Schroeder. 1975. The protection of information in computer systems. *IEEE* 63, 9 (1975), 1278–1308.
[94] J. Smith, B. Johnson, E. Murphy-Hill, B. Chu, and H. R. Lipford. 2019. How Developers Diagnose Potential Security Vulnerabilities with a Static Analysis Tool. *IEEE Trans. Softw. Eng.* 45, 9 (2019), 877–897.
[95] K.-J. Stol and B. Fitzgerald. 2018. The ABC of Software Engineering Research. *ACM Trans Soft Eng Methodol* 27, 3 (2018).
[96] M. Tahaei and K. Vaniea. 2019. A Survey on Developer-Centred Security. In *IEEE European Symp. on Security and Privacy Workshops (EuroS&PW)*. 129–138.
[97] M. Tahaei, K. Vaniea, K. Beznosov, and M. K. Wolters. 2021. Security Notifications in Static Analysis Tools: Developers' Attitudes, Comprehension, and Ability to Act on Them. In *Conf. Hum. Factors Comput. Syst.*
[98] T. W. Thomas, M. Tabassum, B. Chu, and H. Lipford. 2018. Security During Application Development: An Application Security Expert Perspective. In *Conf. Hum. Factors Comput. Syst.* ACM, New York, NY, USA.
[99] C. Thompson and D. Wagner. 2017. A Large-Scale Study of Modern Code Review and Security in Open Source Projects. In *13th Int. Conf. on Predictive Models and Data Analytics in Software Engineering*. ACM, 83–92.
[100] I.A. Tøndel, D.S. Cruzes, M.G. Jaatun, and G. Sindre. 2022. Influencing the security prioritisation of an agile software development project. *Comput Secur* 118 (2022).
[101] I. A. Tondel, M. G. Jaatun, and D. Soares Cruzes. 2020. IT Security is From Mars, Software Security is From Venus. *IEEE Secur Priv* 18, 4 (2020), 48–54.
[102] A. Tuladhar, D. Lende, J. Ligatti, and X. Ou. 2021. An analysis of the role of situated learning in starting a security culture in a software company. In *17th Symp. Usable Priv. Secur.* 617–631.
[103] D. van der Linden, P. Anthonysamy, B. Nuseibeh, T. T. Tun, M. Petre, M. Levine, J. Towse, and A. Rashid. 2020. Schrödinger's Security: Opening the Box on App Developers' Security Rationale. In *42nd Int. Conf. Softw. Eng.*
[104] D. Votipka, K. R Fulton, J. Parker, M. Hou, M. L Mazurek, and M. Hicks. 2020. Understanding security mistakes developers make: Qualitative analysis from Build It, Break It, Fix It. In *29th USENIX Security Symp.* USENIX Assoc., 109–126.
[105] T. Wagner, D. Korschun, and C. Troebs. 2020. Deconstructing corporate hypocrisy: A delineation of its behavioral, moral, and attributional facets. *Journal of Business Research* 114 (2020), 385–394.
[106] P. Wang, J. Bangert, and C. Kern. 2021. If It's Not Secure, It Should Not Compile: Preventing DOM-Based XSS in Large-Scale Web Development with API Hardening. In *IEEE/ACM 43rd Int. Conf. Softw. Eng.* 1360–1372.
[107] C. Weir, I. Becker, and L. Blair. 2021. A Passion for Security: Intervening to Help Software Developers. In *IEEE/ACM 43rd Int. Conf. Softw. Eng.* 21–30.
[108] C. Weir, I. Becker, J. Noble, L. Blair, A. Sasse, and A. Rashid. 2019. Interventions for software security: creating a lightweight program of assurance techniques for developers. In *IEEE/ACM 41st Int. Conf. Softw. Eng.* 41–50.
[109] C. Weir, I. Becker, J. Noble, L. Blair, M. A. Sasse, and A. Rashid. 2020. Interventions for long-term software security: Creating a lightweight program of assurance techniques for developers. *Software Pract Exper* 50, 3 (2020), 275–298.
[110] C. Weir, L. Blair, I. Becker, A. Sasse, and J. Noble. 2018. Light-Touch Interventions to Improve Software Development Security. In *IEEE Cybersecur. Dev.* 85–93.
[111] C. Weir and B. Hermann. 2020. From Needs to Actions to Secure Apps? The Effect of Requirements and Developer Practices on App Security. In *29th USENIX Security Symp.* USENIX Assoc.
[112] D. Wermke, N. Wöhler, J. H. Klemmer, M. Fourné, Y. Acar, and S. Fahl. 2022. Committed to Trust: A Qualitative Study on Security & Trust in Open Source Software Projects. In *IEEE Symp. on Security and Privacy*. 1880–1896.
[113] J. Whitmore and W. Tobin. 2017. Improving Attention to Security in Software Design with Analytics and Cognitive Techniques. In *IEEE Cybersecur. Dev.* 16–21.
[114] J. Witschey, S. Xiao, and E. Murphy-Hill. 2014. Technical and Personal Factors Influencing Developers' Adoption of Security Tools. In *ACM Workshop on Security Information Workers - SIW '14*. ACM, Scottsdale, AZ, USA, 23–26.
[115] S. Xiao, J. Witschey, and E. Murphy-Hill. 2014. Social influences on secure development tool adoption: Why security tools spread. In *17th ACM Conf. on Computer Supported Cooperative Work and Social Computing*. 1095–1106.