What Can Self-Admitted Technical Debt Tell Us About Security? A Mixed-Methods Study

Anonymous Author(s)

ABSTRACT

Self-Admitted Technical Debt (SATD) encompasses a wide array of sub-optimal design and implementation choices reported in software artefacts (e.g., code comments and commit messages) by developers themselves. Such reports have been central to the study of software maintenance and evolution over the last decades. However, they can also be deemed as dreadful sources of information on potentially exploitable vulnerabilities and security flaws. **Objective**: This work investigates the security implications of SATD from a technical and developer-centred perspective. On the one hand, it analyses whether security pointers disclosed inside SATD sources can be used to characterise vulnerabilities in Open-Source Software (OSS) projects and repositories. On the other hand, it delves into developers' perspectives regarding the motivations behind this practice, its prevalence, and its potential negative consequences. **Method**: We followed a mixed-methods approach consisting of (i) the analysis of a preexisting dataset containing 94,455 SATD instances and (ii) an online survey with 222 OSS practitioners. **Results**: We gathered 201 SATD instances through the dataset analysis and mapped them to different Common Weakness Enumeration (CWE) identifiers. Overall, 25 different types of CWEs were spotted across commit messages, pull requests, code comments, and issue sections, from which 8 appear among MITRE's Top-25 most dangerous ones. The survey shows that software practitioners often place security pointers across SATD artefacts to promote a security culture among their peers and help them spot flaky code sections, among other motives. However, they also consider such a practice risky as it may facilitate vulnerability exploits. **Implications**: Our findings suggest that preserving the contextual integrity of security pointers disseminated across SATD artefacts is critical to safeguard both commercial and OSS solutions against zero-day attacks.

CCS CONCEPTS

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

KEYWORDS

self-admitted technical debt, software security, software engineering, technical debt identification

MSR 2024, April 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Anonymous Author(s). 2024. What Can Self-Admitted Technical Debt Tell Us About Security? A Mixed-Methods Study. In *Proceedings of 21st International Conference on Mining Software Repositories (MSR 2024).* ACM, New York, NY, USA, 11 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Software developers often face complex technical challenges that, for diverse reasons, they circumvent with spurious design workarounds and sub-optimal implementations [10]. Such compromises are likely to result in a significant amount of Technical Debt (TD), namely low-quality artefacts that can impair the future maintenance and evolution of the system being developed. Different types of TD have been reported in the current literature, including code, design, requirements, and documentation. If not identified timely and properly managed, TD can incur extra work in the long-term, not to mention additional costs. Hence, it is deemed a critical issue for software development and considered ubiquitous to most software projects across all industry segments [10].

Motivation. Security issues are closely related to TD as they are often grounded in quality compromises in the software being built [29, 33]. Simple coding mistakes and design flaws can lead to exploitable vulnerabilities that are time- and effort-intensive to repair later on [1, 3]. Moreover, similar to TD, security incurs extra costs as it is frequently seen as an "after-thought" instead of an actual priority in the software development life cycle [26]. Therefore, recent studies have started looking at the intersection between TD and security with the aim of detecting and managing vulnerabilities in both code and the core architecture of information systems (e.g., [1, 29, 33, 34]). That is, to leverage TD as a security indicator [29] and for the automatic detection of code vulnerabilities [34]. Still, this research landscape is in its infancy and calls for further investigations at the interface of TD and software security.

A significant part of TD is explicitly reported by developers themselves in the form of "self-admissions", namely through software artefacts such as code comments [22]. Moreover, recent studies have regarded commit messages and pull requests as valuable sources of Self-Admitted Technical Debt (SATD) [39]. These artefacts have been central for the analysis of different TD types beyond code, including requirement debt and documentation debt [35]. However, SATD sources can also have major security implications when reporting flaws and vulnerabilities in the software. Particularly, an attacker could take advantage of *security pointers* disclosed inside these sources (e.g., a code comment like "`TODO: Make it thread-safe`") to spot exploitable weaknesses in the system threatening its availability, confidentiality and integrity. Nevertheless, and despite its importance, the security implications of SATD have not been investigated nor documented throughout the literature up to our knowledge.

Contribution and Research Questions. This work elaborates on the implications of security pointers inside SATD instances across multiple sources. Particularly, it investigates whether explicit references to security vulnerabilities inside code comments, commits, issue reports, and pull requests can help characterise exploitable weaknesses inside Open-Source Software (OSS). We focused our study on OSS projects and repositories as they become easy targets of security attacks by making their artefacts visible and available to the public in general [18, 23]. We conducted a mixed-methods analysis of data extracted from OSS repositories and a complementary online survey to address the following Research Questions (RQs):

- ***RQ1: Can SATD sources contain pointers to security weaknesses and vulnerabilities inside OSS repositories?*** To answer this RQ, we conducted an analysis on a preexisting dataset containing 94,455 SATD instances from multiple sources [22]. Using a keyword-based search, we extracted 546 security-relevant SATD candidates and inspected them manually for security pointers. As a result, we gathered 201 Security Self-Admitted Technical Debt (SSATD) observations that we considered for further analysis.
- ***RQ2: Which particular weakness and vulnerabilities are frequently exposed through different SATD sources?*** For this RQ, we manually mapped each SSATD instance in our resulting dataset to Common Weakness Enumeration (CWE) identifiers. Overall, we identified 25 different CWE types across commit messages, code comments, pull requests, and issue reports. Moreover, 8 of these 25 CWEs appear listed among MITRE's Top-25 most dangerous software weaknesses in 2023.
- ***RQ3: What is developers' perspective on security-related SATD in OSS repositories?*** We addressed this RQ through an online survey with OSS practitioners. We recruited a total of 222 participants and asked them about their tendencies to disclose security pointers inside SATD artefacts, their underlying motivations, and whether they considered this to be a risky practice. Our results show that many practitioners engage with this practice, although they recognise it as potentially risky. In total, we identified nine motives that drive developers to disclose security pointers inside SATD artefacts (e.g., 'promote a security culture') and three potential risks stemming from them (e.g., 'security misconceptions').

The remainder of this paper is organised as follows. Section 2 provides the background and summarises the related work. We explain our research methodology in Section 3, followed by reporting the findings in Section 4. Section 5 reflects on the findings and provides implications for research and practice. We discuss the possible threats of our study and adopted mitigation strategies in Section 6. Section 7 concludes the paper and discusses some future research directions.

2 BACKGROUND AND RELATED WORK

Security and Technical Debt. As mentioned, only a few publications in the current literature explore the interface between software security and TD. Rindell and Holvitie [30] conducted a seminal work addressing the role and importance of TD for managing emerging security risks in software projects. They propose extending traditional TD management frameworks with (i) means for assessing the risks of TD items and (ii) security-based tools for unveiling unintentional TD instances (i.e., those caused by unawareness or sheer recklessness). Overall, they suggest that tools/methods such as code reviews and threat modelling are suitable to spot internal quality issues in software requirements, coding, architecture, and testing [29]. Alongside, Martinez et al. [25] conducted a literature review to understand the main sociotechnical characteristics of "security debt" and its impact on the software development life cycle. They acknowledge that TD can entail security implications at different levels (e.g., code and architecture) and, given its potential damage, should be highly prioritised.

Prior work has also elaborated on the interplay between TD and security flaws, namely on their role for the (automatic) identification of exploitable weaknesses. For instance, Izurieta et al. [14] proposed a method to assess and prioritise TD instances linked to security flaws using the Common Weakness Enumeration (CWE) and its scoring mechanism. This approach was further investigated and extended in [13] by mapping CWEs to malicious attack tactics. Similarly, Siavvas et al. [33] explored TD as a software security indicator and used it later on as a feature for predicting vulnerabilities at both project and class levels [34]. They conclude that TD pointers can help improve the performance of Machine Learning (ML) models for vulnerability prediction. Recent work by Aldrich Edbert et al. [1] has investigated security-related TD in Question-Answer (Q&A) platforms like Stack Overflow (SO). Their findings show that around 38% of security questions in SO are related to TD, being *encryption* along with *cryptography* among the most popular discussion topics.

Security and Self-Admissions. Research on self-admissions, on the other hand, seeks to draw a more complete picture of the TD landscape by examining additional sources of actionable information. It is worth mentioning that, despite that SATD literature is sparse, most of it has focused on the analysis of TD through code comments. For example, Maldonado and Shihab [24] investigated the different types of TD (e.g., design or documentation debt) developers disclose inside their code comments and provided a set of heuristics for their identification. Likewise, Huang et al. [11] and de Freitas Farias et al. [7] applied text mining techniques for the automatic extraction of SATD inside comments. Russo et al. [31] conducted one of the few investigations addressing security in SATD and observed that more than 55% of self-admissions in the form of code comments correspond to insecure implementations/snippets.

Recent work has stressed the importance of scrutinising alternative SATD sources such as commit messages and pull requests to better characterise TD in software projects [22, 39]. Under this premise, Li et al. [22] ran a large-scale study on SATD addressing different sources, including code comments, commit messages, pull requests, and issue reports across 103 open-source projects. Their results show that SATD is evenly spread among all these sources with a high prevalence of code/design debt. Still, up to the extent of our knowledge, multi-source studies at the interface between SATD and security are missing in the current literature. Particularly, the impact of explicit *security pointers* across multiple SATD instances (e.g., comments or commit messages suggesting the presence of

vulnerabilities) has not been actively explored and calls for further investigations.

Documentation Practices. Research has also addressed developers' documentation and repayment practices of SATD. For instance, Shmerlin et al. [32] studied the drivers behind code documentation and observed that *comprehension*, *order* and *quality* are among developers' primary motivations. Conversely, they observed that the perceived *time* and *effort* required to document code often act as deterrents for SATD. Code reviews were also shown to play a key role in the emergence of SATD. Particularly, a study by Kashiwa et al. [15] revealed that around 28-48% of self-admissions inside comments extracted from Qt and Open Stack systems were introduced during reviewing activities. Alongside, a survey by Xavier et al. [38] investigated the motivations that drive developers to (i) introduce and (ii) repay SATD reported in issues. Results show that *timely delivery* is often a cause for the former, whereas *reducing the costs* and interests associated with TD may influence the latter.

Although the reasons why developers take and later repay TD are well documented in the current literature, little is known about their motivations to report it in the form of self-admissions. That is, which are the underlying factors that either encourage or discourage developers from reporting technical shortcuts across different software artefacts. An exploratory study on SATD in R packages by Vidoni [37] provides some insights, suggesting that self-admissions are often introduced as "self-reminders", "warnings", or "to facilitate future planning". Still, other intrinsic and extrinsic motivations, such as perceived rewards, risks, and social norms, remain unexplored and demand a thorough investigation at both software engineering and security levels.

3 METHODOLOGY

To answer the RQs presented in Section 1, we followed a mixed-methods approach encompassing (i) the analysis of OSS repositories and (ii) an online survey with practitioners. The following sections describe the steps we followed at each stage along with the instruments employed during the different data acquisition and processing activities.

3.1 Repositories Study

This study consisted of identifying and analysing security pointers disclosed inside SATD artefacts. For this, we used a preexisting SATD dataset as a reference and conducted a keyword-based search enhanced with a series of manual validations and annotations.

3.1.1 Reference Dataset. In a recent publication, Li et al. [22] presented the results of a large-scale study on SATD in OSS projects and released a dataset for the automatic identification of self-admissions. The dataset contains **94,455 software artefacts** from which 5,000 correspond to commit messages and another 5,000 to pull requests collected from 103 Apache OSS projects. It also includes 23,180 issue sections collected during a prior study [21] and 62,275 code comments from a dataset curated by da Silva Maldonado et al. [6]. All instances in the final dataset are labelled either as SATD or non-SATD, making it suitable for the purpose of our research.

3.1.2 Generation of SSATD Candidates. Figure 1 illustrates the steps we followed for curating a dataset of Security Self-Admitted Technical Debt (SSATD) instances. First, we took the *software artefacts* dataset (SWAD) of Li et al. [22] and filtered the SATD instances by checking the corresponding label value (i.e., SATD or non-SATD). As shown in Table 1, we gathered **8,812 self-admissions** from which 4,071 correspond to code comments, 747 to commit messages, 3,276 to issue sections, and 718 to pull requests. Next, we conducted a keyword-based search to detect SSATD candidates using a list of security terms elaborated by Croft et al. [5]. This list is an updated version of the one proposed by Le et al. [20] and contains 288 security keywords (e.g., 'leak', 'thread-safe'), making it one of the most extensive ones documented in the literature. Although this type of filtering has known limitations (e.g., it can produce false positives), we considered it appropriate at this stage given the size of our dataset and the relatively short text length of the self-admissions. We applied the keyword search after pre-processing each SATD instance (i.e., by removing all punctuation characters) and obtained a total of **546 SSATD candidates**.

Table 1: Number and type of SATD instances in the *reference*, *candidates*, and *final* datasets.

Source	SATD	SSATD Candidates	Final SSATD
Comments	4,071	180	29
Commits	747	45	33
Issues	3,276	284	123
Pull requests	718	37	16
Total	8,812	546	201

3.1.3 Manual Validation and Labelling. Two of the authors manually validated the dataset of SSATD candidates by checking for false positives. Both of them had prior experience in software security (one slightly senior to the other), so they conducted this task based on their own knowledge and technical expertise. As a general rule, they had to determine whether the information disclosed in the artefact they assessed could indicate the presence of a security flaw. Each author applied this criterion independently and labelled each of the 546 SSTD candidates as a *true positive* (TP) or *false positive* (FP), accordingly. We measured the level of agreement between the classifications of both authors using Cohen's Kappa. The values obtained indicate a 'substantial' agreement with a coefficient of +0.65 [19]. The authors then discussed, negotiated, and solved the discrepancies in their annotations on a one-by-one basis during a joint session. This led to a final dataset of **201 SSATD instances**, which they mapped jointly to Common Weakness Enumeration (CWE) identifiers in a follow-up session. Supported by MITRE, the CWE initiative consists of a public, community-developed catalogue with information about more than 900 types of software and hardware weaknesses. The mapping was conducted following an opportunistic approach in which the authors browsed the CWE database[1] using the keywords found on each SSATD item as a guide. Overall, we identified **26 different CWEs** across the items included in our final dataset.

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

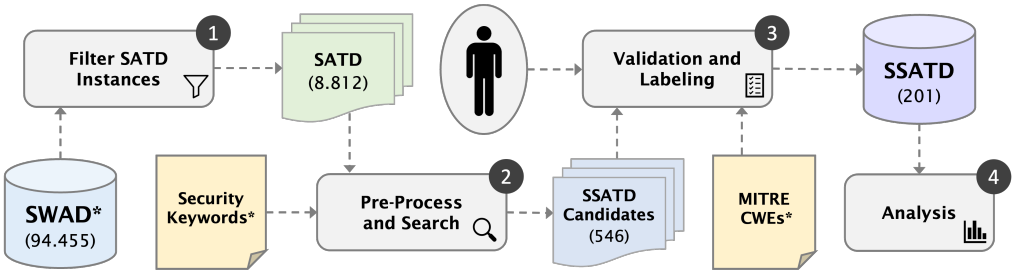

Figure 1: Methodology applied for curating the SSATD dataset. Note: External input is indicated with an asterisk.

3.2 Survey Study

To gather insights into developers perceptions about SSATD we conducted an online survey addressing different behavioural and motivational aspects. It consisted of closed-ended and 2 open-ended questions that took participants around 8 minutes to respond.

3.2.1 Structure. We designed our survey based on the findings collected during the repositories study. Particularly, we used information about the most frequent CWEs in the SSATD dataset to elaborate a set of representative examples that guided participants throughout the study. Such examples (e.g., the excerpt of an issue mentioning "`cache producer is not thread safe`") aimed to prime participants about situations they may have encountered while developing software. That is, about cases in which security pointers get disseminated across code comments, commit messages, pull requests, or issue sections. We referred participants to these examples on each of the reminder survey questions and asked them to answer based on their own experience working on OSS projects:

- **SSATD Prevalence:** To understand SSATD prevalence in OSS repositories, we asked developers (i) how often do *they (themselves)* disclose security pointers inside comments, commits, issues, or pull requests (**Q1**), and (ii) how often do they see *other developers* doing so (**Q2**). These two questions were ranked using a five-point scale as "Never", "Rarely", "Sometimes", "Often", and "Always".
- **SSATD Motivations:** We also elicited participants' motivations to engage in this practice by asking them to assess a set of statements. Assal and Chiasson [2] suggest that developers' (i) work environment, (ii) sense of awareness, (iii) perceived rewards, and (iv) perceived negative consequences can drive them to engage in security-related practices. Hence, we elaborated one statement (**Q3**-**Q6**) for each of these 4 prospective motivations (e.g., *"I introduce security pointers inside commits, issues, comments, or pull requests because this is a standard practice at work"*) and asked participants to rate them using a six-point scale ("Strongly Agree", "Agree", "Somehow Agree", "Somehow Disagree", "Disagree", and "Strongly Disagree"). We also added an open-ended question (**Q7**) to ask them whether they could think of any other drivers besides the ones mentioned in the statements.
- **SSATD Risks:** Finally, we asked participants about their perceptions of risks stemming from the disclosure of security pointers inside software artefacts. Particularly, we asked them to indicate how much they agreed (or disagreed) with the following statement: *"I think security pointers inside commits, issues, comments, or pull requests may introduce risks in the software being deployed"* (**Q8**) using a six-point scale ("Strongly Agree", "Agree", "Somehow Agree", "Somehow Disagree", "Disagree", and "Strongly Disagree"). As with the motivation statements, we also included an open-ended question (**Q9**) to ask them which specific risks they think such pointers may introduce in OSS projects.

A subset of the main survey questions can be found in Appendix A. The complete survey instrument is available in the paper's **Replication Package**.

3.2.2 Population and Recruitment. Participants were recruited via Prolific[2], a crowdsourcing platform frequently adopted to conduct empirical studies in software engineering [16, 36], and prescreened using its built-in qualification features. We targeted individuals who reported to have (i) knowledge of software development techniques and (ii) computer programming skills. We validated these qualifications through a short technical questionnaire similar to the one proposed by [17]. It consisted of 3 closed-ended questions, the first one asking whether the participants had contributed to OSS projects in the past (answer options: "yes"/"no"), while the remaining two asked them to indicate the *parameter* and *output* of a given pseudocode, respectively (answer options: multiple choices). This screening questionnaire took around 2 minutes, for which we rewarded participants with 0.2 GBP each. Only those who passed it were invited to take the *main survey* (Section 3.2.1) and received an additional compensation of 1.10 GBP. As a general rule, participants had to be at least 18 years old to join both parts of the study (i.e., screening questionnaire and main survey), have taken part in at least 10 other studies in Prolific, and have a minimum approval rate of 98%. Two attention questions were also included in the main survey to identify and discard answers from unengaged participants.

After running a pilot study with 10 participants, we observed that around 30% passed the technical questionnaire. Based on this ratio and our study budget, we targeted a sample of size N=200. Hence, we repeated the process with 740 additional participants, from which 242 were deemed suitable for undertaking the main survey. That is, after validating their technical knowledge and prior experience with OSS projects. As a result, we collected **222 valid answers** from 227 main survey submissions.

3.2.3 Ethical Considerations. We received approval from the Ethics Committee of the [BLINDED INSTITUTION] to conduct this

[2]https://www.prolific.com/

study and followed the guidelines prescribed in the Declaration of Helsinki. We informed participants about the study procedure along with details concerning the privacy of their personal data before joining the experiment. We also asked for their informed consent and allowed them to withdraw at any time without their answers being recorded.

4 RESULTS

We conducted multiple analyses of the data collected through the repositories and survey studies. Our main findings and characteristics of the resulting datasets are summarised in the following subsections.

4.1 Security Pointers in SATD (RQ1 and RQ2)

As mentioned in Section 3.1.3, we gathered 201 SSATD instances, which represent 2.28% of the total SATD available in the reference dataset. Since instances included in the reference dataset were labelled with the type of TD they refer to, so are the SSATD cases included in ours. Table 2 shows that most of them correspond to sub-optimal choices taken at the code/design levels (72.1%), followed by unsatisfied or partially-implemented requirements (16.9%). Uncorrected known defects and poor testing come next with 4% of the total cases each. Other TD types like documentation, architecture and build debt represent another 3% altogether. We observe that most of the code/design and requirement debt is concentrated around issue sections, whereas defects are often self-admitted through code comments. On the other hand, commit messages seem to gather most of the test debt cases and half of the documentation debt ones. The rest of the TD types, namely architecture and build debt, were found inside one issue section and one commit message, respectively.

Table 2: Types of TD found in the SSATD dataset.

SATD Type	PR	CC	IS	CM	TOTAL
Defect Debt	0	7	1	0	8
Code/Design Debt	12	17	96	20	145
Requirement Debt	3	5	21	5	34
Test Debt	1	0	2	5	8
Documentation Debt	0	0	2	2	4
Architecture Debt	0	0	1	0	1
Build Debt	0	0	0	1	1

Note: PR=Pull Request; CC=Code Comment; IS=Issue Section; CM=Commit Message

Table 3 summarises the frequency of each CWE we identified in our final dataset, along with a representative example. Overall, we spotted 25 different CWE pointer types, from which 8 appear in MITRE's Top-25 list of the most dangerous software weaknesses in 2023 (i.e., CWEs 362, 119, 20, 79, 287, 89, 352, and 798). We observe that *CWE-362: 'Concurrent Execution using Shared Resource with Improper Synchronization (Race Condition)'* is the most frequent one, followed by *CWE-119: 'Improper Restriction of Operations within the Bounds of Memory Buffer'* with 34 and 31 cases, respectively. As shown in Figure 2, these two most popular security pointers are often disclosed inside issue sections. However, we also found them inside pull requests, code comments, and commit messages with clear references to (i) non-thread-safe implementations in the case of CWE-362 and (ii) memory leakages in the code for CWE-119.

We also encountered pointers to *CWE-402: 'Transmission of Private Resources into a New Sphere (Resource Leak)'* and *CWE-20: 'Improper Input Validation'* with a relatively high frequency of 18 and 15 cases, respectively. CWE-20 was found in all artefact types but most frequently inside issue sections and code comments. These pointers often refer to missing or improper input validation, as in a code comment mentioning "`TODO: Validate port as numeric`". On the other hand, the majority of CWE-402 pointers were grouped around issue sections, with some of them also disclosed in the form of commit messages. Such is the case of an issue section in which a developer asks to fix a resource leak caused by a connection failure (see example in Table 3). We also found 15 pointers to *CWE 404: 'Improper Resource Shutdown or Release'*, all of them corresponding to commit messages. In this case, pointers describe situations in which resources are not properly released and end up exposing sensitive data in subsequent memory allocations.

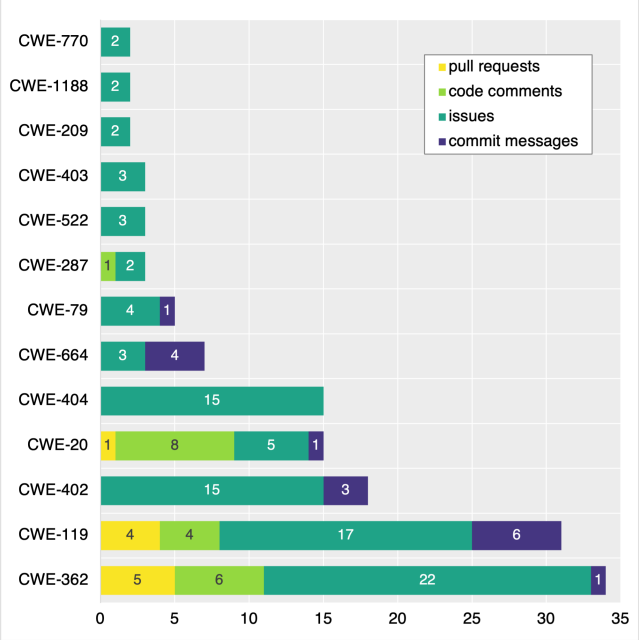

Figure 2: Distributions of CWE across SSATD artefacts. Note: CWEs with a frequency < 2 are not displayed in the graph.

CWE-664: 'Improper Control of a Resource Though its Lifetime' and *CWE-79: 'Improper Neutralisation of Input During Web Page Generation (Cross-site Scripting)'* were also found inside commit messages and issue sections but with a lower frequency. Particularly, we retrieved 7 pointers to CWE-664 and 5 to CWE-79. In the case of the former, the pointers describe cases where resources are not properly created, used, or destroyed, which can lead to exploitable system states and unexpected behaviours. For the latter, the issue sections or commit messages make reference to the presence of Cross-site Scripting (XSS) vulnerabilities due to improper neutralisation of user-controllable input. The remaining CWE pointers cover 18 different CWEs and have 3 or less observations each (e.g., CWE-209

Table 3: Examples and frequencies of security pointers included in the final SSATD dataset.

Pointer to	Freq.	Example
CWE-362: Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	34	"Cache producer is not thread safe" [IS]
CWE-119: Improper Restriction of Operations within the Bounds of a Memory Buffer	31	"Memory leak from the main thread on shutdown" [CM]
CWE-402: Transmission of Private Resources into a New Sphere ('Resource Leak')	18	"Fix Python TSocket resource leak on connection failure" [IS]
CWE-20: Improper Input Validation	15	"TODO: Validate port as numeric" [CC]
CWE-404: Improper Resource Shutdown or Release	15	"Potential resource leak in due to unclosed stream" [CC]
CWE-664: Improper Control of a Resource Through its Lifetime	7	"Fix open file leak in log cleaner integration tests" [CM]
CWE-79: Improper Neutralization of Input During Web Page Generation ('Cross-site Scripting')	5	"The ")]}" prefix in the JSON response is intended ... It's a security measure to prevent XSSI" [IS]
CWE-287: Improper Authentication	3	"TODO: Add dialog to inform user that the smtp server does not support authentication" [CC]
CWE-522: Insufficiently Protected Credentials	3	"Plain-text information leak of due to autosuggest" [IS]
CWE-403: Exposure of File Descriptor to Unintended Control Sphere ('File Descriptor Leak')	3	"Metrics system FileSink can leak file descriptor" [IS]
CWE-209: Generation of Error Message Containing Sensitive Information	2	"A new user was trying to perform a "repo init" for the first time on his workstation and saw the following output: ... "fatal: not a Gerrit project" ...I later discovered that the user was not a member of previously established LDAP groups with read permissions in Gerrit. That being the case, I would have expected to see a "permission denied" type message, not a message stating that the requested project did not exist, when in fact it did..." [IS]
CWE-1188: Insecure Default Initialization of Resource	2	"SSHD default MAC/KEX/cipher settings are rather insecure" [IS]
CWE-770: Allocation of Resources Without Limits or Throttling	2	"The JobTracker can be prone to a denial-of-service attack if a user submits a job that has a very large number of tasks ... It would be nice to have a configuration setting that limits the maximum tasks that a single job can have" [IS]
Other CWEs with just one observation (CWE: 89, 121, 668, 319, 312, 352, 477, 326, 798, 295, 172, 825)	12	"silly hack to avoid stack overflow" [CC | CWE-121]
Generic security pointers	69	"The original logic looks not safe. Do you think if there is any other better way?" [PR]

Note: PR=Pull Request; CC=Code Comment; IS=Issue Section; CM=Commit Message

and CWE-121). They account for 27 cases in total, most of them found inside issue sections. We also found generic pointers whose content is not specific enough to map them to concrete CWEs. For instance, a pull request describing "The original logic looks not safe. Do you think if there is any other better way?" suggests that the current implementation is not entirely secure but does not offer much detail about the specific threats or flaws in the code. All in all, such generic pointers add up to 31.22% of the observations in the SSATD dataset.

4.2 Developers' Perspective (RQ3)

Of 222 study participants, 190 were male, 29 were female, and 3 were non-binary. Around 23.9% reported having more than 5 years of experience working with OSS projects, 13.9% less than a year, and the rest between 1 and 5 years. Regarding their experience with closed-source development, about 31.1% had more than 5 years of working experience, 23.9% less than 1, whereas the remainder reported having between 1 and 5 years. A complete overview of the sample's demographics is shown in Table 4.

4.2.1 SSATD Prevalence. We can observe from Figure 3 that the majority of participants have disclosed security pointers (themselves) inside SATD artefacts at some point (89.6%). That is, either *rarely*, *sometimes*, *often*, or *always*. Still, a relatively high percentage reported to *rarely* engage in this practice (33.3%) or not having engaged on it at all while working with OSS repositories (10.4%). We also observe that most respondents have seen at some point another developer adding pointers inside commits, code comments,

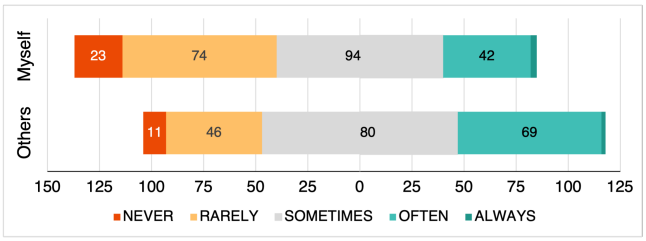

Figure 3: Prevalence of security pointers in SATD. Note: Frequencies < 10 are not labelled in the graph.

Table 4: Survey self-reported demographic data.

Demographic	Ranges	Freq.	%
Gender	Male	190	85.6%
	Female	29	13.1%
	Non-Binary	3	1.3%
Educational level	High School or Less	11	5.0%
	Some College	39	17.6%
	Undergraduate (BSc, BA)	98	44.1%
	Graduate (MSc, PhD)	74	33.3%
Experience with OSS projects	<1 year	31	13.9%
	1-2 years	58	26.1%
	2-3 years	49	22.1%
	3-4 years	19	8.6%
	4-5 years	12	5.4%
	>5 years	53	23.9%
Experience with closed-source projects	<1 year	53	23.9%
	1-2 years	38	17.1%
	2-3 years	34	15.3%
	3-4 years	20	9.0%
	4-5 years	8	3.6%
	>5 years	69	31.1%

issue sections, or pull requests (95%). However, 20.7% of participants claimed to have *rarely* identified such behaviour in others, whereas just 5% reported not to have seen it at all.

4.2.2 SSATD Motivations. For the analysis of developers' motivations to disclose security pointers, we only considered the answers of those who claimed to have done so *themselves* at some point (i.e., N=199 participants). As illustrated in Figure 4, most of them follow this practice because they either (i) consider security important or (ii) believe this can prevent potential security risks. Many also do so because it is standard practice at their workplace and, to a lesser extent, because such efforts get somehow recognised. However, these two factors, namely work practices and recognition, also gathered the highest number of disagreements among participants. Particularly, several developers tend to believe that placing security pointers inside code comments, commit messages, pull requests, or issue sections is not sufficiently recognised in their workplace.

We identified some additional motivations by analysing participants' answers to the open-ended question. For this, we followed an inductive method in which the first author performed open coding and, together with another author, discussed emerging themes and patterns in the data. In the case of any discrepancies, these were solved following a negotiated agreement approach [4]. We found **5 extra motivations**, which we describe below and report their frequencies using the (×) symbol.

(i) Improve project quality *(×17)* Many respondents stressed the importance of security pointers for the overall quality of OSS projects. For instance, they believed this was important to improve the performance of code review activities and the early detection of vulnerabilities.

> *P60: "Security pointers can help to improve the quality of code reviews by providing reviewers with information about potential security risks in the code being reviewed."*

(ii) Comply with regulations and standards *(×17)* Compliance with current standards and regulations also motivates developers to include security cues. Particularly, to demonstrate the adherence to legal requirements and security best practices.

> *P212: "The company I work for does not truly care about the benefits of good security pointers, instead the company is forced to do them due to compliance and regulation..."*

(iii) Facilitate collaboration *(×51)* Many reported that security pointers help others to spot flaky code sections and fix them timely. Likewise, they believed these are important documentation instruments that ultimately promote and enhance collaboration among developers.

> *P92: "I use annotations and comments related to security in software as a note to collaborators and myself to be aware of these issues, especially if problems are easily introduced or something is confusing without more intimate knowledge of why something was done. That is, I add this information as form of documentation and to be proactive."*

(iv) Self-reminders *(×10)* Participants also mentioned that this helps them to keep track of the progress and status of their coding activities. Moreover, some also said it helps them to strengthen their secure development skills.

> *P98: "Just to remember what could be wrong in the future. I mean, there is a high probability that if I do not write something I will forget it in some months, when I may retake/reanalyse the project."*

(v) Promote a security culture *(×40)* Fostering a security-conscious culture across development teams was also a recurrent topic. That is, in terms of educating others and encouraging them to address software security in a proactive manner.

> *P59: "Educating my fellow contributors in producing better code, preventing mistakes if they modify my contributions and improving the community I am in by nudging them towards best practices."*

4.2.3 SSATD Risks. Finally, we considered the answers of all participants when analysing their risk perceptions (i.e., N=222 answers). Figure 5 shows an even distribution of results, with 50% of respondents tending to agree that security self-admissions may introduce risks in OSS projects. Conversely, the other 50% reported having an opposite view in this regard. As with the motivations, we analysed the open-ended question about the potential risks of security pointers, following the same inductive approach to identify emerging topics. Overall, we gathered **3 different risks** that participants considered likely to emerge from the disclosure of security pointers.

(i) Exposing vulnerabilities *(×131)* A large number of respondents believed that security cues inside code comments, commit

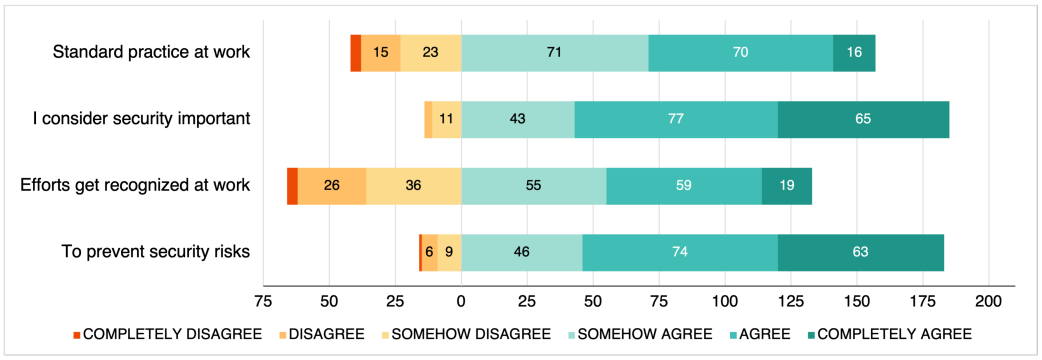

Figure 4: Motivations to disclose security pointers in SATD. Note: Frequencies < 10 are not labelled in the graph.

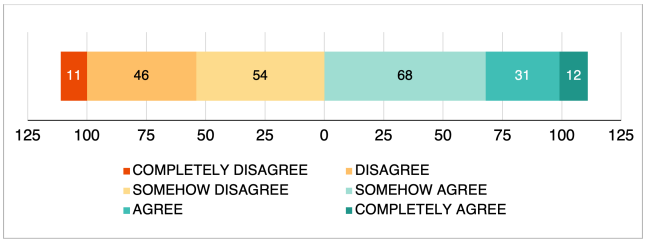

Figure 5: Risk perceptions of security pointers in SATD.

messages, pull requests, and issue sections may facilitate vulnerability exploits. Particularly, they considered that such pointers could create a roadmap of the security flaws inside OSS repositories and that attackers may easily take advantage of them.

> *P60: "Security pointers can make OSS more attractive to attackers, as they can provide a roadmap for exploiting vulnerabilities."*
>
> *P91: "If they are picked up by bad actors and exploited, this is risky."*

(ii) Security misconceptions *(×38)* Some participants said that pointers may not be accurate and lead either to a false sense of security or to deviate the attention of developers to non-existing vulnerabilities. Moreover, they considered that this sort of "security advice" could slow down the development process if not handled properly.

> *P190: "...if these tips are not clear or if they give the wrong advice, people might make mistakes and accidentally make the software less secure. So, it's really important that the tips are correct and easy to follow for them to be helpful."*

(iii) Exposure of sensitive information *(×21)* Another emerging topic concerned the possibility of revealing sensitive information to untrusted audiences, such as passwords, secrets, and credentials that unauthorized parties may exploit.

> *P127: "Security pointers may contain sensitive information such as passwords, API keys, or other credentials that can be exposed to unauthorized parties if not handled properly"*

5 DISCUSSION

In this section, we discuss our findings in the context of existing literature and provide implications for research and practice.

SSATD and Software Security. The outcome of our study provides valuable insights into the role and impact of SSATD on software security. On the one hand, we observe that security pointers disclosed in the form of commit messages, pull requests, issue sections, and code comments can provide enough detail to indicate the presence of vulnerabilities. Furthermore, as reported in Section 4.1, the level of granularity of many of these self-admissions can suggest the existence of dangerous software weaknesses (i.e., CWEs listed in MITRE's Top-25). In this regard, our results align with the ones of Russo et al. [31], who identified 14 different CWE issues associated with SATD instances found in the Chromium C-Code project (i.e., in the form of code comments). Moreover, a recent study by Pan et al. [27] revealed that issue reports gathered across 1.390 OSS projects from GitHub contained information about 132 different CWE vulnerability types. In line with these findings, ours also accounts for the threats that security pointers may pose for the OSS ecosystem if disclosed early to the public. That is, before vulnerabilities get reported privately to the corresponding project maintainers. Particularly, malicious actors could take advantage of the leaked information (namely, the reproducible steps of a vulnerability) and launch zero-day attacks [27]. Furthermore, such a practice can severely affect commercial closed-source software solutions, as they often use OSS resources (e.g., packages and libraries) in their deployment [18]. Hence, current vulnerability disclosure protocols (e.g., ISO/IEC 29147:2018 [12]) should acknowledge the side effects of SSATD to become more resilient against this type of attack.

SSATD and Privacy Behaviour. The insights gathered through the online survey are also nuanced. We can see that most developers are keen to report security issues throughout different SATD sources but, at the same time, seem to acknowledge that this is a risky practice (especially when working in an OSS environment). Prior work by Díaz Ferreyra et al. [8] suggests that software practitioners tend to underestimate the potential negative consequences of revealing personal and project-related information in social coding platforms and may exhibit paradoxical behaviour in this regard. That is, they often engage in unsavvy privacy/security practices

despite their concerns. In part, such behaviour can be associated with their perceptions of trust in the OSS ecosystem and prior experiences of privacy invasion [8]. In this sense, those who have not suffered from any privacy harm or consider OSS platforms as safe collaboration channels are prone to reveal more information about themselves or the projects they work on. Previous findings concerning software documentation [32, 37] and security best practices [2, 28] show that extrinsic/intrinsic motivations (e.g., self-reminders, peer pressure and organisational culture) also play an important role in this process. Overall, our results align with the ones reported in the current literature and show that such motivations can also foster SSATD. Therefore, it is important to balance out these drivers with higher levels of risk awareness to promote the incorporation of security pointers within certain confidentiality boundaries.

Sensitive Information in SSATD. From the qualitative analysis of the open-ended questions, we observe concerns stemming from the unintended disclosure of sensitive/exploitable information to untrusted audiences. These concerns are closely related to *secret sprawling*, an emerging phenomenon in software engineering where secrets (e.g., passwords, certificates, and authentication tokens) get accidentally pushed/committed into publicly-available OSS repositories [17]. Particularly, it is estimated that secret sprawling on GitHub has more than doubled since 2020, making it a significant security threat for software projects [9]. In turn, several commercial and OSS tools have been proposed to facilitate the "sanitization" of code repositories through the automatic identification of secrets [23]. Still, our research shows that this is a multifaceted, complex problem, and security self-admissions can also enlarge a system's attack surface. This calls for developer-centred tools and instruments that could help prevent the dissemination of SSATD to untrusted parties by preserving the contextual integrity of security pointers.

SSATD in Vulnerability Prediction Methods. As mentioned in Section 2, recent work has started investigating whether TD indicators can improve the performance of ML-based vulnerability prediction methods [34]. Our findings also contribute in this regard as they could be leveraged to enhance these approaches. That is, by using SSATD as a complementary feature in ML models for automatically detecting security weaknesses. Moreover, as we reported in Section 4.1, issue sections have been shown to concentrate the vast majority of the security pointers in our dataset. Therefore, future investigations in ML for security should strongly consider expanding the scope of their analysis to multiple SSATD sources and explore their suitability for developing more accurate automatic vulnerability detection solutions. Still, further analysis is required to check whether these pointers effectively refer to vulnerable code implementations and account for a significant number of true positive cases.

RESEARCH OUTLOOK.

(1) Security pointers disclosed inside SATD sources can provide enough detail to suggest the presence of well-known software vulnerabilities (i.e., listed as CWEs).
(2) Vulnerability disclosure protocols should take SSATD into account to better safeguard both commercial and open-source software projects against zero-day attacks.
(3) Developers should gain awareness of the risks of disseminating security pointers inside SATD sources and count on means for reporting them safely to trusted parties only.
(4) ML solutions for automatically predicting software vulnerabilities should consider improving their performance by incorporating SSATD sources as model features.

6 THREATS TO VALIDITY

To a certain extent, the results of our study are subject to limitations related to its experimental design. In particular, the following *construct*, *external*, and *internal* threats may affect the validity of our findings and conclusions:

Internal Validity: As described in Section 3, we conducted a keyword-based search to identify SSATD candidates in the reference dataset. Such an approach is not error-free and may produce some false-positive results. For instance, security keywords like 'hack' and 'hash' triggered many cases in which the candidates did not contain any security pointer (e.g., "`Hack to allow external control`"). Likewise, since the keyword list we employed is not exhaustive, some true positive cases may have been overlooked. We sought to minimize these threats by performing a manual check of each SSATD candidate and employing an extensive list of security keywords. As mentioned earlier, the check was conducted by two of the paper co-authors, who are both knowledgeable in cybersecurity and solved their discrepancies through negotiation. As for the keywords, we used a list that is well-established and documented in the current literature. Still, we acknowledge we may have missed some SSATD instances throughout the data curation process.

The recruitment process followed in the survey study also suffers from limitations. Particularly, it is known that prospective participants recruited via crowd-sourcing platforms may not always meet the study eligibility criteria (e.g., some of them may claim to have prior OSS development experience when, in fact, they do not). We have addressed this issue by (i) using Prolific's in-built qualification features and (ii) running a screening questionnaire before the main survey, as suggested when conducting studies of this nature (see [36]). Additionally, we introduced attention questions to spot random answers from dishonest participants. Overall, the detailed answers observed in the open-ended questions gave us the confidence that the vast majority of participants had the right level of expertise. However, this recruitment approach may also introduce a "survivorship bias" in the studied sample, failing to collect the views from those OSS practitioners outside Prolific. This threat was mitigated thanks to the answers of several experienced participants within our sample (Section 4.2).

Construct Validity: The manual mapping of CWE identifiers to SSATD instances is prone to subjective judgements and may lead to spurious conclusions. Unlike the labelling of SSATD instances, this process is more complex, given the high number of CWE identifiers available. Therefore, we followed an opportunistic approach in which two authors browsed the CWE database guided by the keywords found on each SSATD item (Section 3.1.3). This process

was first performed by one of them and then validated by the other one. Follow-up discussions between the two sought to enhance the construct validity at this step. For dubious cases, a third author (also with extensive cybersecurity expertise) was consulted to facilitate reaching a decision and further improve the labelling outcome. We followed a similar approach to mitigate subjective biases while coding the survey's open-ended questions (i.e., Q7 and Q9). Here, one author performed an initial coding of the data, whereas another checked them and completed/fixed them when needed.

External Validity: Although the reference dataset used in the repositories study is fairly large (94,455 software artefacts), it may not offer a complete picture of the SATD phenomenon in OSS. Moreover, our SSATD dataset contains 201 instances, which may pose a potential threat to the generalizability of our results. We have addressed this concern by comparing and discussing our findings with the ones reported in the current literature to minimize the risk of interpretation bias. Generalizability threats also arise from the survey study, given the size and composition of the studied sample. On the one hand, because of its size, we cannot generalise the study results to the entire community of OSS developers. Instead, the insights gained from it should be seen as preliminary and motivate further research in this area. On the other hand, because of its demographic composition, we may have failed to capture the perspectives of underrepresented gender groups (e.g., women and gender-diverse people). Hence, future investigations should target larger and more gender-diverse samples for the sake of generalizability.

7 CONCLUSION AND FUTURE WORK

The findings gathered throughout our mixed-methods study suggest that security pointers disclosed across SATD sources are like a double-edged sword. On the one hand, they aid developers in finding and fixing security weaknesses spread across OSS projects. Furthermore, they are often considered important for fostering a security culture across development teams and complying with current standards and regulations. However, we also see that practitioners often engage in this practice despite the potential risks, particularly concerning the role of SSATD in identifying exploitable vulnerabilities.

Having mapped several SSATD instances to concrete CWEs speaks about the severity of spreading security pointers carelessly across OSS repositories. Hence, preserving the contextual integrity of this information is critical to protect both commercial and OSS solutions against zero-day attacks. As discussed in Section 5, vulnerability disclosure protocols should take SSATD explicitly into account to enhance their robustness against this type of attack. Likewise, secret scanning tools should extend their scope with SSATD identification features to assist developers in keeping this information only accessible to trusted parties. Last but not least, developers' awareness and training about the inherent risks of this practice are key for promoting best practices in this regard.

Several directions for future work arise from the results of this study. One relates to the role of privacy-related antecedents when disclosing security-related information across SATD artefacts. Particularly, prior work has reported that developers' perceptions of trust, risk, and control over personal and project-related information may influence their privacy practices inside Social Coding Platforms (SCPs) like GitHub [8]. Since SCPs are major sources of OSS repositories, it would be relevant to determine whether such perceptions contribute (or not) to the dissemination of SSATD across code comments, commit messages, issue sections and pull requests. In addition, a closer inspection of the code sections linked to SSATD instances would help us to get a more complete picture of their actual implications. That is, whether they are good indicators of vulnerabilities and we can leverage them to elaborate ML models for their automatic prediction.

REPLICATION PACKAGE

Survey instruments, consent forms, and study results are available at https://anonymous.4open.science/r/SSATD-ANONYM/

Appendix A SURVEY QUESTIONS

The most salient questions from the survey described in Section 3.2 are listed below. Note that open-ended questions are indicated with an asterisk (*). The complete survey instrument is available in the paper's replication package.

(1) **SSATD Prevalence:**
- **Q1.** How often have you (yourself) produced an artefact (i.e., commits, issues, comments, or pull requests) containing security pointers?
- **Q2.** How often have you seen others producing an artefact containing security pointers?

Answer options: *Never, Rarely, Sometimes, Often, Always.*

(2) **SSATD Motivations:**
- **Q3.** While working in OSS projects, I introduce security pointers inside an artefact because this is a standard practice at work.
- **Q4.** While working in OSS projects, I introduce security pointers inside an artefact because I consider security an important aspect.
- **Q5.** While working in OSS projects, I introduce security pointers inside an artefact because such efforts are recognized at work.
- **Q6.** While working in OSS projects, I introduce security pointers inside an artefact because they help prevent security risks.
- **Q7*.** What other motives may drive you to introduce security pointers inside an artefact?

Answer options: *Completely Disagree, Disagree, Somehow Disagree, Somehow Agree, Agree, Completely Agree.*

(3) **SSATD Risks:**
- **Q8.** I think security pointers inside an artefact may introduce risks in the OSS being deployed.
- **Q9*.** What kind of risks do you think security pointers inside an artefact introduce in the OSS being deployed?

Answer options: *Completely Disagree, Disagree, Somehow Disagree, Somehow Agree, Agree, Completely Agree.*

REFERENCES

[1] Joshua Aldrich Edbert, Sahrima Jannat Oishwee, Shubhashis Karmakar, Zadia Codabux, and Roberto Verdecchia. 2023. Exploring Technical Debt in Security Questions on Stack Overflow. In *Proceedings of the 76th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '23)*. ACM.

[2] Hala Assal and Sonia Chiasson. 2019. 'Think secure from the beginning' A Survey with Software Developers. In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–13.

[3] Felivel Camilo, Andrew Meneely, and Meiyappan Nagappan. 2015. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 269–279.

[4] John L Campbell, Charles Quincy, Jordan Osserman, and Ove K Pedersen. 2013. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological methods & research* 42, 3 (2013), 294–320.

[5] Roland Croft, Yongzheng Xie, Mansooreh Zahedi, M Ali Babar, and Christoph Treude. 2022. An empirical study of developers' discussions about security challenges of different programming languages. *Empirical Software Engineering* 27 (2022), 1–52.

[6] Everton da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1044–1062.

[7] Mário André de Freitas Farias, Manoel Gomes de Mendonça Neto, Marcos Kalinowski, and Rodrigo Oliveira Spínola. 2020. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology* 121 (2020), 106270.

[8] Nicolás E. Díaz Ferreyra, Abdessamad Imine, Melina Vidoni, and Riccardo Scandariato. 2023. Developers Need Protection, Too: Perspectives and Research Challenges for Privacy in Social Coding Platforms. In *16th International Conference on Cooperative and Human Aspects of Software Engineering (CHASE 2023)*. https://doi.org/10.1109/CHASE58964.2023.00019

[9] GitGuardian. 2022. *The State of Secrets Sprawl 2022*. https://res.cloudinary.com/da8kiytlc/image/upload/v1646148528/GitGuardian_StateOfSecretsSprawl2022.pdf Accessed: 25.10.2023.

[10] Yuepu Guo, Rodrigo Oliveira Spínola, and Carolyn Seaman. 2016. Exploring the costs of technical debt management–a case study. *Empirical Software Engineering* 21 (2016), 159–182.

[11] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23 (2018), 418–451.

[12] ISO/IEC 29147:2018 2018. *Security Techniques – Vulnerability Disclosure*. Standard. International Organization for Standardization, Geneva, CH.

[13] Clemente Izurieta and Mary Prouty. 2019. Leveraging SecDevOps to Tackle the Technical Debt Associated with Cybersecurity Attack Tactics. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 33–37. https://doi.org/10.1109/TechDebt.2019.00012

[14] Clemente Izurieta, David Rice, Kali Kimball, and Tessa Valentien. 2018. A Position Study to Investigate Technical Debt Associated with Security Weaknesses. In *Proceedings of the 2018 International Conference on Technical Debt* (Gothenburg, Sweden) *(TechDebt '18)*. Association for Computing Machinery, New York, NY, USA, 138–142. https://doi.org/10.1145/3194164.3194167

[15] Yutaro Kashiwa, Ryoma Nishikawa, Yasutaka Kamei, Masanari Kondo, Emad Shihab, Ryosuke Sato, and Naoyasu Ubayashi. 2022. An empirical study on self-admitted technical debt in modern code review. *Information and Software Technology* 146 (2022), 106855.

[16] Harjot Kaur, Sabrina Amft, Daniel Votipka, Yasemin Acar, and Sascha Fahl. 2022. Where to recruit for security development studies: Comparing six software developer samples. In *31st USENIX Security Symposium (USENIX Security 22)*. 4041–4058.

[17] Alexander Krause, Jan H Klemmer, Nicolas Huaman, Dominik Wermke, Yasemin Acar, and Sascha Fahl. 2023. Pushed by Accident: A {Mixed-Methods} Study on Strategies of Handling Secret Information in Source Code Repositories. In *32nd USENIX Security Symposium (USENIX Security 23)*. 2527–2544.

[18] Piergiorgio Ladisa, Henrik Plate, Matias Martinez, and Olivier Barais. 2023. Sok: Taxonomy of attacks on open-source software supply chains. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1509–1526.

[19] J. Richard Landis and Gary G. Koch. 1977. The measurement of observer agreement for categorical data. *biometrics* (1977), 159–174.

[20] Triet Huynh Minh Le, David Hin, Roland Croft, and M Ali Babar. 2020. Puminer: Mining security posts from developer question and answer websites with pu learning. In *Proceedings of the 17th International Conference on Mining Software Repositories*. 350–361.

[21] Yikun Li, Mohamed Soliman, and Paris Avgeriou. 2022. Identifying self-admitted technical debt in issue tracking systems using machine learning. *Empirical Software Engineering* 27, 6 (2022), 131.

[22] Yikun Li, Mohamed Soliman, and Paris Avgeriou. 2023. Automatic identification of self-admitted technical debt from four different sources. *Empirical Software Engineering* 28, 3 (2023), 1–38.

[23] Sofiane Lounici, Marco Rosa, Carlo Maria Negri, Slim Trabelsi, and Melek Önen. 2021. Optimizing Leak Detection in Open-source Platforms with Machine Learning Techniques.. In *ICISSP*. 145–159.

[24] Everton da S. Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical Debt. In *2015 IEEE 7th International Workshop on Managing Technical Debt (MTD)*. 9–15. https://doi.org/10.1109/MTD.2015.7332619

[25] Jabier Martinez, Nuria Quintano, Alejandra Ruiz, Izaskun Santamaria, Iker Martinez de Soria, and José Arias. 2021. Security Debt: Characteristics, Product Life-Cycle Integration and Items. In *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*. 1–5. https://doi.org/10.1109/TechDebt52882.2021.00009

[26] Mahmood Niazi, Ashraf Mohammed Saeed, Mohammad Alshayeb, Sajjad Mahmood, and Saad Zafar. 2020. A maturity model for secure requirements engineering. *Computers & Security* 95 (2020), 101852.

[27] Shengyi Pan, Jiayuan Zhou, Filipe Roseiro Cogo, Xin Xia, Lingfeng Bao, Xing Hu, Shanping Li, and Ahmed E. Hassan. 2022. Automated Unearthing of Dangerous Issue Reports. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 834–846. https://doi.org/10.1145/3540250.3549156

[28] Irum Rauf, Tamara Lopez, Helen Sharp, Marian Petre, Thein Tun, Mark Levine, John Towse, Dirk van der Linden, Awais Rashid, and Bashar Nuseibeh. 2022. Influences of developers' perspectives on their engagement with security in code. In *Proceedings of the 15th International Conference on Cooperative and Human Aspects of Software Engineering*. 86–95.

[29] Kalle Rindell, Karin Bernsmed, and Martin Gilje Jaatun. 2019. Managing security in software: Or: How I learned to stop worrying and manage the security technical debt. In *Proceedings of the 14th International Conference on Availability, Reliability and Security*. 1–8.

[30] Kalle Rindell and Johannes Holvitie. 2019. Security Risk Assessment and Management as Technical Debt. In *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*. 1–8. https://doi.org/10.1109/CyberSecPODS.2019.8885100

[31] Barbara Russo, Matteo Camilli, and Moritz Mock. 2022. WeakSATD: Detecting Weak Self-Admitted Technical Debt. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 448–453. https://doi.org/10.1145/3524842.3528469

[32] Yulia Shmerlin, Irit Hadar, Doron Kliger, and Hayim Makabee. 2015. To document or not to document? An exploratory study on developers' motivation to document code. In *Advanced Information Systems Engineering Workshops*. Springer, 100–106.

[33] Miltiadis Siavvas, Dimitrios Tsoukalas, Marija Jankovic, Dionysios Kehagias, Alexander Chatzigeorgiou, Dimitrios Tzovaras, Nenad Anicic, and Erol Gelenbe. 2019. An empirical evaluation of the relationship between technical debt and software security. In *9th International Conference on Information society and technology (ICIST)*, Vol. 2019.

[34] Miltiadis Siavvas, Dimitrios Tsoukalas, Marija Jankovic, Dionysios Kehagias, and Dimitrios Tzovaras. 2022. Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises. *Enterprise Information Systems* 16, 5 (2022), 1824017.

[35] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *Journal of Systems and Software* 152 (2019), 70–82.

[36] Mohammad Tahaei and Kami Vaniea. 2022. Recruiting participants with programming skills: A comparison of four crowdsourcing platforms and a CS student mailing list. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*. 1–15.

[37] Melina Vidoni. 2021. Self-admitted technical debt in r packages: An exploratory study. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 179–189.

[38] Laerte Xavier, João Eduardo Montandon, Fabio Ferreira, Rodrigo Brito, and Marco Tulio Valente. 2022. On the documentation of self-admitted technical debt in issues. *Empirical Software Engineering* 27, 7 (2022), 163.

[39] Fiorella Zampetti, Gianmarco Fucci, Alexander Serebrenik, and Massimiliano Di Penta. 2021. Self-admitted technical debt practices: a comparison between industry and open-source. *Empirical Software Engineering* 26 (2021), 1–32.