WeakSATD: Detecting Weak Self-admitted Technical Debt

Barbara Russo
Free University of Bozen-Bolzano
Italy
barbara.russo@unibz.it

Matteo Camilli
Free University of Bozen-Bolzano
Italy
matteo.camilli@unibz.it

Moritz Mock
Free University of Bozen-Bolzano
Italy
moritz.mock@stud-inf.unibz.it

ABSTRACT

Speeding up development may produce technical debt, i.e., not-quite-right code for which the effort to make it right increases with time as a sort of interest. Developers may be aware of the debt as they admit it in their code comments. Literature reports that such a self-admitted technical debt survives for a long time in a program, but it is not yet clear its impact on the quality of the code in the long term. We argue that self-admitted technical debt contains a number of different weaknesses that may affect the security of a program. Therefore, the longer a debt is not paid back the higher is the risk that the weaknesses can be exploited. To discuss our claim and rise the developers' awareness of the vulnerability of the self-admitted technical debt that is not paid back, we explore the self-admitted technical debt in the Chromium C-code to detect any known weaknesses. In this preliminary study, we first mine the Common Weakness Enumeration repository to define heuristics for the automatic detection and fix of weak code. Then, we parse the C-code to find self-admitted technical debt and the code block it refers to. Finally, we use the heuristics to find weak code snippets associated to self-admitted technical debt and recommend their potential mitigation to developers. Such knowledge can be used to prioritize self-admitted technical debt for repair. A prototype has been developed and applied to the Chromium code. Initial findings report that 55% of self-admitted technical debt code contains weak code of 14 different types.

CCS CONCEPTS

• **Software and its engineering** → *Maintaining software*; • **Security and privacy** → *Software and application security*.

KEYWORDS

Self-admitted technical debt, Weak code, Security, Vulnerability

ACM Reference Format:
Barbara Russo, Matteo Camilli, and Moritz Mock. 2022. WeakSATD: Detecting Weak Self-admitted Technical Debt. In *19th International Conference on Mining Software Repositories (MSR '22), May 23–24, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3524842.3528469

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR '22, May 23–24, 2022, Pittsburgh, PA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9303-4/22/05…$15.00
https://doi.org/10.1145/3524842.3528469

1 INTRODUCTION

Not-quite-right code is introduced for short-term needs [4, 34]. If it is not fixed, it may increase over time with negative impact on code quality [38] and, if it is fixed, it may cause cost of additional rework [20]. Such additional effort is called *technical debt*. Examples of technical debt are code smells and bug hazards. Identifying automatically such a code is the goal of recent lines of research [1, 37, 38]. One option is to trace *Self-Admitted Technical Debt* (SATD), i.e., code comments explicitly introduced by developers to tag the technical debt [1, 11, 20]. Such comments can be automatically detected and the associated code can be identified and removed or modified to mitigate the debt (i.e., paying back technical debt) [37]. According to developers' opinion, SATD is not introduced because of pressure [1], but it is rather included intentionally to track future bugs and areas of the code that need refactoring [10, 12]. It has been hypothesised that SATD may affect software correctness [10], but literature has not yet reported a conclusive answer [1, 32, 35, 37]. Not-quite-right code may indeed contain *weaknesses* (i.e., code that exposes software to security breaches). Such weaknesses can be exploited by a party to cause the product to modify or access unintended data, interrupt proper execution, or perform incorrect actions that were not specifically granted to the party who exploits the weakness. Thus, the longer this code remains in the software, the higher will be the interest to pay it back, and the longer will be the exposure of the software to third party's exploitation. As SATD on average survives for over 1,000 commits [1], weaknesses in the SATD-related code represent a real security risk. Even though developers are aware of the technical debt since they self-admit it, they may not be aware of the portion of such debt that is also weak and the security risks at which their software is exposed. In this work, we consider the following research question:

RQ: Is self-admitted technical debt related to weaknesses in source code?

We argue that *detecting weaknesses associated to SATD can increase developers' awareness on the vulnerability of their code and on the risk of not paying the technical debt back and help them plan maintenance activities for security concerns (e.g., prioritize SATD for repair).*

In this paper, we present a preliminary study in which we analyzed the source code written in C of the Chromium project [2] to understand whether code blocks to which SATD comments refer may contain weaknesses. To achieve this goal, we developed the *WeakSATD* approach that mines the public Common Weaknesses and Enumeration (CWE) repository [24] to derive a set of heuristics to detect known weaknesses in software code and recommend their mitigation. SATD comments and related code are then retrieved. Finally, the heuristics are used to automatically detect the presence and the types of weaknesses in the code to which SATD comments refer. Mitigation to such weaknesses are then recommended. We

believe that such knowledge can be leveraged to prioritize SATD for repair (e.g., with or without weaknesses or with one or more weaknesses) and speed up SATD removal.

The remainder of this paper is as follows. We introduce *Weak-SATD* in Sec. 2. We discuss our initial findings with the Chromium project in Sec. 3. We briefly report related work in Sec. 4. We conclude the paper and we present future directions in Sec. 5.

2 WEAKSATD

In this section, we introduce an overview of the whole approach (Sec. 2.1), we describe how we mine and leverage the information in the CWE issues (Sec. 2.2), how we derive our weakness heuristics (Sec. 2.3) used to analyze the Chromium project, the detection of SATD-blocks (Sec. 2.4), and the WealSATD prototype (Sec. 2.5).

2.1 Overview

Figure 1 illustrates a high-level overview of our approach that includes the following steps.

Define relevant CWE issues. We first mine the CWE repository that contains the state-of-the-art list of existing weaknesses (hardware and software) and extract the weakness types (called *CWE issues*) that pertain to software code written in C and include in their description of some C-code examples.

Mine CWE repository. Then, we define weakness heuristics as regex rules in C by analyzing the code snippets reported as *weak-code examples* in these CWE issues' description (we call them *weak-code snippets*). For the rest of the paper, we refer to *weak code* as a portion of the code that contains one or more weak-code snippets. *Detect SATD in code.* At the same time, we determine the *SATD-blocks* as the blocks (e.g., method block or block of a loop) in the C-code of Chromium to which SATD instances refer. To this aim, we parse the code to identify SATD comments with regex rules matching the 62 patterns defined by Potdar and Shihab [20] and to detect code blocks associated with them.

Find SATD-blocks that contain CWE issues. Finally, we use regex rules to find weak-code snippets in the SATD-blocks and provide mitigation actions by exploiting the information contained in the description of the CWE issues. The same information is leveraged to provide mitigation actions.

We have developed a prototype tool that automatizes our approach, visualizes instances of SATD-blocks that contain potential weaknesses, and suggests mitigation actions[28]. Finally, we implemented the approach also using GitHub Actions [6] to enable automation of our approach in CI/CD pipelines. A more detailed description of our approach follows in the next sections.

An anonymized package containing the implementation of *Weak-SATD* demo, the list of heuristics, the Github Actions, and the data extracted from CWE is publicly available [28].

2.2 Exploring the CWE repository

CWE is part of a larger MITRE [3] initiative for collecting, classifying, and publishing data on weaknesses, vulnerabilities, and attacks to software and hardware. CWE is a public community-maintained moderated repository of over 900 types of software and hardware known weaknesses. Weakness types are entered in CWE as issues (*CWE issues*). Each issue may include a description, relationships with other issues (if any), platforms and programming languages that can be affected, known consequences related to attacks or system malfunctioning, demonstrative code examples, and potential mitigation actions as in Table 1. Search in the CWE repo can be performed by issue-ID or keywords. No API is available, and the dataset can be downloaded as a set of `.cvs` or `.hml` files. CWE issues are also linked to records stored in the Common Vulnerabilities and Enumerations (CVE) [23] or National Vulnerabilities Database (NVD) [19], i.e., the U.S. government repository of standards-based vulnerability management data represented using the Security Content Automation Protocol (SCAP). CVE contains data on vendors, products and versions of products as well as vulnerabilities per type (e.g., Denial of Service, Code Execution, Overflow, etc.). Finally, the exploitation of vulnerabilities (e.g., exploitation code and information) are maintained in the exploit database [30]. Entries in the exploit DB, CVE/NVD and CWE repositories are all linked through the identifier *CWE-ID*. Some bug tracking systems (e.g., Mozilla [15]) are also using the CWE-ID to annotate issues related to security. Thus, by linking SATD to CWE issues, *we can trace technical debt to weaknesses, their effects (vulnerabilities and bugs) and possible attacks (exploitation).*

The CWE repository contains more than 920 different weaknesses types (CWE issues) for software and hardware linked each other through their CWE-ID when this is specified in the *Relationships* field, as shown in Table 1. For instance, CWE-242 is a "member of" the general category CWE-699. The relationships field can include different types of associations between CWE issues like MemberOf, ChildOf and ParentOf.

To select the relevant list of CWE issues, we have inspected the 900 CWE types also exploiting the relationship they declare in their description. We have then manually isolated the issues that: (1) pertain only to software development (total 419), (2) can be found in the C programming language, and (3) contain a code example in the C language. By applying these criteria, we finally obtained a sample of 80 CWE issues.

Table 1: Relevant information in a CWE issue.

Title	**CWE-242: Use of Inherently Dangerous Function**
Description	The program calls a function that can never be guaranteed to work safely
Extended Description	Certain functions behave in dangerous ways regardless of how they are used. ... The gets() function is unsafe because does not perform bounds checking on the size of its input. An attacker can easily send arbitrarily-sized input to gets() and overflow the destination buffer. ...
Relationships	Relevant to the view "Software Development" (CWE-699) *Nature* | *Type* | *ID* | *Name* MemberOf | C | 1228 | API/Function Errors
Modes of Introduction:	Phase: Implementation
Applicable Platform	Languages: C, C++
Common Consequences	Technical Impact: Varies by *Context*
Likelihood Of Exploit	High

Demonstrative Examples	The code below calls gets() to read information into a buffer.
	Example. Language: C
	`char buf[BUFSIZE];`
	`gets(buf);`

Potential mitigations	Phases: Implementation; Requirements ban the use of dangerous functions. Use their safe equivalent. Phase: Testing; Use grep or static analysis tools to spot usage of dangerous functions.

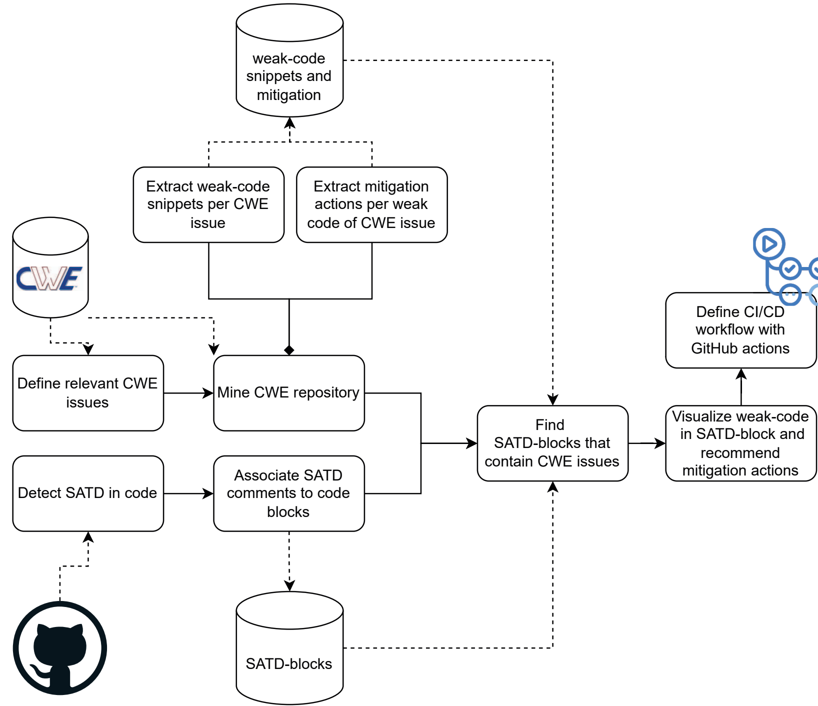

Figure 1: Overview of our approach to detect weaknesses in code associated to SATD.

2.3 Weakness Heuristics

To identify weaknesses in the code, we define a set of heuristics by manually inspecting the 80 CWE issues. Firstly, we further reduce the number of CWE issues to analyze by exploiting the dependency structures among them. Specifically, we re-use the heuristic of the parent node when possible and we implement only the heuristics for the child nodes when the parent's issue description is too coarse to detect its children. At the end, we come up with *34 relevant heuristics*. After applying the aforementioned selection criteria, we implement regex rules to parse the C-code by analyzing the description and the code snippets included as examples of vulnerability in C in the description of the corresponding CWE issue. The recommended mitigation actions of the CWE issue are further extracted from the CWE description and associated to the weak-code snippet. The heuristics we find depend on the specific programming language, the available information in the CWE repository, and researchers' personal knowledge of the language, although they are independent from the Chromium project. Thus, they are far from being a complete set for any C-written project and represent only a demonstrative sample to illustrate our research idea. Table 2 illustrates an example of heuristic we defined for the issue CWE-676 in the C language. The weak-code snippet in CWE-676 creates a local copy of a buffer to perform some data manipulations. The function `strcpy()` copies a string into a buffer with no control on the size of the string potentially exposing the software to buffer overflow exploits. The heuristic we defined searches for the potentially dangerous functions in C, `strcpy()` illustrated in the CWE-676 example and also for each of the potentially vulnerable functions in the banned list maintained by Microsoft [14]. More complex regex rules have been defined in other cases. The set of regex rules we implemented are available in the replication package [28].

Table 2: A heuristic for CWE-676.

Title	**CWE-676: Use of Potentially Dangerous Function**
Example in C	`void manipulate_string(char * string){ char buf[24]; strcpy(buf, string); ... }`
Heuristic	Search for "`strcpy(`" with no alphanumerical character in front of the keyword

2.4 SATD-blocks

As per common practice [1, 37], we implement a set of regex rules to search among all comments for the 62 patterns defined by Potdar and Shihab [20] to detect SATD in code [21] and use srcML to decompose C-code in comments and blocks, [1, 20, 37]. Then, we associate a SATD instance to a block following a proximity rule as we derived as in the following. Firstly, in all comments in the code we search the ones that include one or more SATD patterns (*SATD-comments*). Secondly, we consider all code blocks and associate each of them to a SATD comment by a proximity rule based on the results of a Wilcoxon test between the two distributions of lines of code between SATD comments and enclosing or following code blocks. For instance, for the Chromium project, we decided to

select blocks that occur *after* a comment since the Wilcoxon test significantly (α=0.05) reported the block after the comment as the nearest one[1]. SATD comments that are not associated to any code block are not included in the analysis. We finally call *SATD-block*, the code block associated in this way to a SATD comment.

2.5 Weak SATD

With the regex rules defined from the heuristics, we search weak-code snippets in SATD-blocks and called *Weak SATD* the positive results. It is worth noting that a block can contain more instances of such snippets from the same or different CWE issues, Listing 3. For each of the found weak-code snippet, we also retrieve the mitigation actions recommended by the CWE issue in the field "potential mitigations", as shown in Table 1.

Prototype. We developed a prototype tool implementing the *WeakSATD* approach. The tool is a web-based application that uses react.js and npm-packages (e.g., `lodash` and `adm-zip`). A RESTful API is implemented by using `express` and `mongoose` to interact with `MongoDB` and `babel` to allow compliance with the react syntax. The API handles saving of relevant CWE data and statistics on weak-code snippets in MongoDB as JSON record. The tool uses `SrcML` to retrieve relevant items from the code (comments, blocks). Finally, we developed GitHub Actions for the analysed CWE issues to exemplify the integration of *WeakSATD* into CI/CD pipelines. The action is executed upon new release commits to spot weak-code snippets in SATD-blocks. Then it notifies the developer with potential mitigation actions. The GitHub action runs as a job configured through a `.yml` file as illustrated in Listings 1.

Listing 1: **A GitHub Action for *WeakSATD*.**

```
jobs:
  weakSATDAction:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Run weakSATD Action
        uses: <account>/WeakSATD-Action@v0.0.1
```

3 INITIAL FINDINGS IN CHROMIUM

We applied our approach to the C sources of Chromium [25], version 88.0.4323. We selected this project as it was originally used by Nord *et al.* [17] for their initial findings on the relation between vulnerabilities and technical debt and by Potdar and Shihab [20] in the definition of the 62 SATD patterns.

By mining the CWE repository, with our selection criteria (Sec. 2), we obtained and implemented 34 weakness heuristics. For each CWE issue, we refined the initial implementation of the weakness heuristics in our demo tool after manually analyzing a sample of 775 files sampled with 90% confidence level and 5% margin of error. With this manual inspection, we also found that the SATD pattern "take care" produced quite a few false positives in the SATD comments of Chromium as illustrated in Listing 2. To stay on the conservative side, we did not consider this pattern in our analysis.

[1]Note this result may be different in other software projects.

Listing 2: **False positive comment for the "take care" SATD pattern.**

```
// Callers are encouraged to use the setters
    provided which take care of setting |options|
    as desired.
```

Finally, we made sure that for this sample, the tool predicted weak-SATD blocks that were actual weak-code according to the CWE description. We applied our weakness heuristics and our SATD-block rule to 41753 C files. Table 3 shows that the percentage of files with SATD and the percentage of comments with SATD found with our approach is within the ranges reported in literature.

Table 3: % SATD files and comments in recent literature.

	% files with SATD comments	% SATD comments
WeakSATD [28]	1.6%*	0.1%*
Potdar et al. [20]	2.4-31%	-
Bavota et al. [1]	-	0.2-0.4%
Maldonado et al. [12]	-	0.02-0.21%
Zampetti et al. [37]	-	0.02-0.21%
Iammarino et al. [8]	-	0.02-0.21%
Fucci et al. [5]	-	0.02-0.21%

*Computed considering SATD comments with blocks only.

We found that 10885 files (26% of all C files) contain a number of weak-code snippets ranging from 0 for issue CWE-243 to 4091 for issue CWE-783 (average=1197, Q1=14, median Q2=88, Q3=365). It is worth noting that issue CWE-783 is very common as it refers to the use of an expression in which operator precedence causes incorrect logic. We also found 847 distinct SATD blocks for about 0.1% of all comments. We found SATD blocks in 634 different files (about 1.6% of all files). Out of those SATD blocks, 465, that is, 55% of them contain at least one potential weakness according to the CWE catalog. We found that weak SATD-blocks are distributed over 14 different CWE issues (about 41% of the implemented issues) with a maximum of 197 SATD-blocks for issue CWE-483 "Incorrect Block Delimitation" (e.g., missing brackets for an "if statement").

Our initial findings with the Chromium project have no ambition of generalization also because, at the current stage, we do not cover all CWE issues. However, we can summarize our findings as follows to answer our initial research question.

RQ summary: In our experiments with the Chromium project, we found that 26% of all source files contain at least one weak-code snippet. Furthermore, 55% of the SATD blocks contain weak-code snippets of 14 different CWE issues. Indeed, correlation between weaknesses and SATD warrant further research.

Listing 3: **SATD comment and relative block with multiple CWE instances.**

```
1 /* FIXME: this code assumes that sigmask is an
      even multiple of the size of a long
     integer. */
2
3 unsigned long *src = (unsigned long const *)
     set;
4 unsigned long *dest = (unsigned long *) &(
     thread.p->sigmask);
```

```
5
6  switch (how)
7  {
8      case SIG_BLOCK:
9      for (i = 0; i < (sizeof (sigset_t) /
           sizeof (unsigned long)); i++)
10     {
11       /* OR the bit field longword-wise. */
12       *dest++ |= *src++;
13     }
14     break;
15     case SIG_UNBLOCK:
16     for (i = 0; i < (sizeof (sigset_t) /
           sizeof (unsigned long)); i++)
17     {
18       /* XOR the bitfield longword-wise. */
19       *dest++ ^= *src++;
20     }
21     case SIG_SETMASK:
22     /* Replace the whole sigmask. */
23     memcpy (&(thread.p->sigmask), set, sizeof
           (sigset_t));
24     break;
25 }
```

The Chromium code in Listing 3 shows an example of SATD comment and its SATD block in which our tool detects the following instances of CWE-issues: CWE-478 "Missing Default Case in Switch-statement" (at the end of the switch), CWE-484 "Omitted Break Statement in Switch" (after the second case), CWE-242/676 "Use of Potentially/Inherently Dangerous Function" (line 23).

4 RELATED WORK

In our work we adopt a simple pattern-based SATD detection method, while other existing approaches may be in principle adopted to achieve higher accuracy [22, 36]. Supervised approaches may be more precise in detecting SATD instances (60 − 85% precision with ML [36] vs. 75% with unsupervised search [1]). Nevertheless, they require ground truth and training effort that is not needed in pattern-based SATD detection.

Considering the detection (or prediction) of security threats, recent approaches exploit the information in the history of software projects and connecting it to software vulnerabilities, [7, 9, 13, 16, 17, 29, 33]. For instance, Mazuera-Rozo *et al.* [13] classify code functions with Deep and Machine Learning techniques to classify them by known vulnerabilities, such as deadlock, race condition, and null pointer dereference. The information on software vulnerabilities is used to label the categories of the classifiers and the focus is on vulnerabilities (e.g., buffer overflow) and not on software weaknesses (e.g., inadequate container capacity).

Table 4 lists recent pieces of work explicitly linking their results to a set of CWE issues. As shown in the table, there is little overlapping with the issues considered in *WeakSATD*. Our approach is indeed different. It exploits the information in the weakness repository (CWE) to detect weak code in software projects.

The relationship between technical debts and weaknesses has been recently studied [17, 31]. Initial findings confirm software developers use technical debt concepts to discuss design limitations and their consequences. However, correlations between vulnerabilities and technical debt indicators requires further research. The approach in [31] uses security bugs in issue tracking systems to identify vulnerabilities. Such bugs are not weaknesses but errors in code. According to the MITRE definition, weaknesses are code snippets that can be potentially (not yet) be exploited. Different from bugs and vulnerabilities, weaknesses represent, in our vision, a sort of debt. None of the aforementioned approaches aim at detecting SATD or connect SATD (or TD) with weaknesses or exploit the information in CWE to identify weaknesses as *WeakSATD*.

Table 4: Recent literature, number of CWE issues analysed and CWE-isse in common with *WeakSATD*.

Datasets	CWE issues	CWE issues in common with *WeakSATD*
Shallow-Deep (2021) [13]	N/A	N/A
Juliet Test Suite (2018) [18, 26]	118	195, 196, 401, 415, 416
DRAPER VDISC (2018) [26, 27]	4	none
Zou et al. (2019)[39]	33	467, 676
WeakSATD [28]	34	-

5 CONCLUSION AND FUTURE DIRECTIONS

More research is needed to understand the relation between vulnerabilities and technical debt [17]. We believe that *WeakSATD* represents a novel contribution in this direction. We applied *WeakSATD* to the Chromium project that has been previously used as test bed in research on technical debt (e.g., [17, 20]). We found that more than 55% of SATD instances contain weak code of 34 different CWE issues. It is worth noting that the weak SATD report must be taken as a warning only as some of the found weak code snippets may be perfectly fine in some circumstances. At the current stage, our work is not meant to predict vulnerabilities or attacks with high precision or recall, but it can be used by developers to prioritize and boost the repair of SATD with the help of the recommended mitigation. In addition, our initial findings with the Chromium project have no ambition of generalization especially because we were able to define a substantial amount if weakness heuristics but still not covering all CWE issues. This allowed us to better focus on their definition and implementation, but it prevented us from being more extensive with our findings. Our plan for future work, is to explore other publicly available datasets that are connected with the CWE issues (e.g., the Juliet Test Suite [18]) and can be used to define additional heuristics. We also plan to exploit the whole chain of information available in MITRE (or MITRE-linked repositories) and provide an instrument that recommend developers on the whole chain of consequences (e.g., vulnerabilities, exposure, bugs, and attacks) of leaving SATD in code. The CWE indexing can be used for this scope, but not only. Indeed, some projects do not encode the CWE-ID in their bug issues and security issues can be traced in other ways, like regexes in commit messages, such as the approach introduced in [17]. We also plan to study the perceived usefulness of *weakSATD* by involving professional developers in controlled experiments with humans.

REFERENCES

[1] Gabriele Bavota and Barbara Russo. 2016. A Large-scale Empirical Study on Self-admitted Technical Debt. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 315–326. https://doi.org/10.1145/2901739.2901742

[2] Chromium. 2022. Chromium project. https://www.chromium.org/Home. Last accessed Jan. 2022.

[3] MITRE Corporation. 2022. Federally Funded Research and Development Centers. https://www.mitre.org/. Accessed: Jan-2022.

[4] Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[5] Gianmarco Fucci, Nathan Cassee, Fiorella Zampetti, Nicole Novielli, Alexander Serebrenik, and Massimiliano Di Penta. 2021. Waiting around or job half-done? Sentiment in self-admitted technical debt. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE/ACM, 403–414. https://doi.org/10.1109/MSR52588.2021.00052

[6] Inc. GitHub. 2022. GitHub Actions. https://github.com/features/actions. Accessed: Jan-2022.

[7] Antonios Gkortzis, Dimitris Mitropoulos, and Diomidis Spinellis. 2018. VulinOSS: A Dataset of Security Vulnerabilities in Open-Source Systems. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018*. ACM, 18–21. https://doi.org/10.1145/3196398.3196454

[8] Martina Iammarino, Fiorella Zampetti, Lerina Aversano, and Massimiliano Di Penta. 2019. Self-Admitted Technical Debt Removal and Refactoring Actions: Co-Occurrence or More?. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 186–190. https://doi.org/10.1109/ICSME.2019.00029

[9] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2021. *Vulnerability Detection with Fine-Grained Interpretations*. Association for Computing Machinery, New York, NY, USA, 292–303. https://doi.org/10.1145/3468264.3468597

[10] E. Lim, N. Taksande, and C. Seaman. 2012. A Balancing Act: What Software Practitioners Have to Say about Technical Debt. *IEEE Software* 29, 6 (Nov 2012), 22–27. https://doi.org/10.1109/MS.2012.130

[11] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2018. SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 9–12. https://doi.org/10.1145/3183440.3183478

[12] E. D. S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik. 2017. An Empirical Study on the Removal of Self-Admitted Technical Debt. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 238–248. https://doi.org/10.1109/ICSME.2017.8

[13] Alejandro Mazuera-Rozo, Anamaria Mojica-Hanke, Mario Linares-Vásquez, and Gabriele Bavota. 2021. Shallow or Deep? An Empirical Study on Detecting Vulnerabilities using Deep Learning. In *Proceedings of the 2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*. IEEE/ACM, 276–287. https://doi.org/10.1109/ICPC52881.2021.00034 arXiv:2103.11940 [cs.SE]

[14] Howard Michael. 2011. Security Development Lifecycle (SDL) Banned Function Calls. http://msdn.microsoft.com/en-us/library/bb288454.aspx

[15] Mozilla. 2022. Bugzilla. https://bugzilla.mozilla.org/show_bug.cgi?id=1106067. Accessed: Jan-2022.

[16] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. 2007. Predicting Vulnerable Software Components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security* (Alexandria, Virginia, USA) *(CCS '07)*. Association for Computing Machinery, New York, NY, USA, 529–540. https://doi.org/10.1145/1315245.1315311

[17] Robert L. Nord, Ipek Ozkaya, Edward J. Schwartz, Forrest Shull, and Rick Kazman. 2016. Can Knowledge of Technical Debt Help Identify Software Vulnerabilities?. In *Proceedings of the 9th USENIX Conference on Cyber Security Experimentation and Test* (Austin, TX) *(CSET'16)*. USENIX Association, USA, 1.

[18] National Institute of Standards and Technology (NIST). 2022. Juliet test suite v1.3. https://samate.nist.gov/SRD/testsuite.php. Accessed: Jan-2022.

[19] National Institute of Standards and Technology (NIST). 2022. National Vulnerabilities Database. https://nvd.nist.gov/. Accessed: Jan-2022.

[20] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME '14)*. IEEE Computer Society, USA, 91–100. https://doi.org/10.1109/ICSME.2014.31

[21] Aniket Potdar and Emad Shihab. 2014. List of SATD patterns. http://users.encs.concordia.ca/~eshihab/data/ICSME2014/satd.html. Accessed: Jan-2022.

[22] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural Network-Based Detection of Self-Admitted Technical Debt: From Performance to Explainability. *ACM Trans. Softw. Eng. Methodol.* 28, 3, Article 15 (jul 2019), 45 pages. https://doi.org/10.1145/3324916

[23] CVE repository. 2022. Common Vulnerabilities and Enumerations (CVE). https://cve.mitre.org/index.html. Accessed: Jan-2022.

[24] CWE repository. 2022. Common Weakness Enumeration (CWE). https://cwe.mitre.org/. Accessed: Jan-2022.

[25] Chromium repository. 2022. Federally Funded Research and Development Centers. https://github.com/chromium/chromium. Accessed: Jan-2022.

[26] Rebecca L. Russell, Louis Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. Automated Vulnerability Detection in Source Code Using Deep Representation Learning. arXiv:1807.04320 [cs.LG]

[27] Rebecca L. Russell, Louis Kim, Lei H. Hamilton, Tomo Lazovich, Jacob A. Harer, Onur Ozdemir, Paul M. Ellingwood, and Marc W. McConley. 2018. DRAPER VDISC dataset. https://osf.io/d45bw/wiki/home/. Accessed: Jan-2022.

[28] Barbara Russo, Matteo Camilli, and Moritz Mock. 2022. Replication package. https://doi.org/10.5281/zenodo.5569313. Accessed: Jan-2022.

[29] A. Sabetta and M. Bezzi. 2018. A Practical Approach to the Automatic Classification of Security-Relevant Commits. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE Computer Society, Los Alamitos, CA, USA, 579–582. https://doi.org/10.1109/ICSME.2018.00058

[30] Offensive security. 2022. Exploit database. https://www.exploit-db.com/. Accessed: Jan-2022.

[31] 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.

[32] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *Journal of Systems and Software* 152 (2019), 70–82. https://doi.org/10.1016/j.jss.2019.02.056

[33] Jeffrey Stuckman, James Walden, and Riccardo Scandariato. 2017. The Effect of Dimensionality Reduction on Software Vulnerability Prediction Models. *IEEE Trans. Reliab.* 66, 1 (2017), 17–37. https://doi.org/10.1109/TR.2016.2630503

[34] Cunningham Ward. 2009. Ward explains Debt Metaphor. wiki.c2.com/?WardExplainsDebtMetaphor

[35] S. Wehaibi, E. Shihab, and L. Guerrouj. 2016. Examining the Impact of Self-Admitted Technical Debt on Software Quality. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 1. IEEE, 179–188. https://doi.org/10.1109/SANER.2016.72

[36] Z. Yu, F. Fahid, H. Tu, and T. Menzies. 5555. Identifying Self-Admitted Technical Debts with Jitterbug: A Two-Step Approach. *IEEE Transactions on Software Engineering* 01 (oct 5555), 1–1. https://doi.org/10.1109/TSE.2020.3031401

[37] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2018. Was Self-admitted Technical Debt Removal a Real Removal?: An In-depth Perspective. In *Proceedings of the 15th International Conference on Mining Software Repositories, MSR 2018, Gothenburg, Sweden, May 28-29, 2018* (Gothenburg, Sweden) *(MSR '18)*. ACM, New York, NY, USA, 526–536. https://doi.org/10.1145/3196398.3196423

[38] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the Impact of Design Debt on Software Quality. In *Proceedings of the 2nd Workshop on Managing Technical Debt* (Waikiki, Honolulu, HI, USA) *(MTD '11)*. Association for Computing Machinery, New York, NY, USA, 17–23. https://doi.org/10.1145/1985362.1985366

[39] Deqing Zou, Sujuan Wang, Shouhuai Xu, Zhen Li, and Hai Jin. 2019. μVulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. *IEEE Transactions on Dependable and Secure Computing* 18 (2019), 2224–2236. https://doi.org/10.1109/tdsc.2019.2942930