Can Knowledge of Technical Debt Help Identify Software Vulnerabilities?

Robert L. Nord, Ipek Ozkaya,
Edward J. Schwartz, Forrest Shull
Carnegie Mellon University
Software Engineering Institute
Pittsburgh, PA, USA

Rick Kazman
Carnegie Mellon University
Software Engineering Institute
and University of Hawaii
Honolulu, HI, USA

Abstract

Software vulnerabilities originating from design decisions are hard to find early and time consuming to fix later. We investigated whether the problematic design decisions themselves might be relatively easier to find, based on the concept of "technical debt," i.e., design or implementation constructs that are expedient in the short term but make future changes and fixes more costly. If so, can knowing which components contain technical debt help developers identify and manage certain classes of vulnerabilities? This paper provides our approach for using knowledge of technical debt to identify software vulnerabilities that are difficult to find using only static analysis of the code. We present initial findings from a study of the Chromium open source project that motivates the need to examine a combination of evidence: quantitative static analysis of anomalies in code, qualitative classification of design consequences in issue trackers, and software development indicators in the commit history.

1. Introduction

Technical debt concisely labels a universal problem that software engineers face when developing software: how to balance near-term value with long-term quality. In software-intensive systems, technical debt is a design or implementation construct that is expedient in the short term but sets up a technical context that can make a future change more costly or even impossible [22].

For example, code quality issues such as dead code or duplicate code add to the technical debt. They do not affect the functionality seen by the end user but can impede progress and make development more costly over time. Software architecture plays a significant role in the development of large systems; flaws in a software system's design, such as a frequently changing interface between two classes (an unstable interface) [30], can also add significantly to the technical debt.

Research has shown that technical debt correlates with greater likelihood of defects, unintended rework, and increased time for implementing new system capabilities if not paid back in time [15][17][20][24].

Technical debt can have observable adverse consequences on software security as well, meaning that allowing debt to accumulate may be even more costly. Some vulnerabilities may be inadvertently introduced as the result of technical debt: for example, if a vulnerability is fixed in one location but is not fixed in a similar duplicated code fragment, or if overly complex code makes it harder to reason about whether a dangerous corner-case condition is feasible or not. Alternatively, as we will show, technical debt can also be caused by addressing a vulnerability's symptoms rather than its root cause. Both of these relationships motivate a better understanding of the complex relationship between software vulnerabilities and technical debt.

Although security research is often focused on the study of vulnerabilities, the field of software engineering includes the more general study of anomalous (i.e., buggy) software (Figure 1), with its own well-established vocabulary and techniques for analyzing various types of software anomalies. These anomalies include security vulnerabilities, technical debt, and defects, which are errors in coding or logic that cause a program to malfunction or produce incorrect or unexpected results [19].

There is a growing body of work on these three forms of software anomalies that seeks to understand how they relate to each other [4]. Establishing the causality and potential relationships between these different forms of anomalous software behavior could enable the use of complimentary techniques for detection, prevention, and mitigation. For example, we might be able to use technical debt techniques to trace vulnerabilities from their

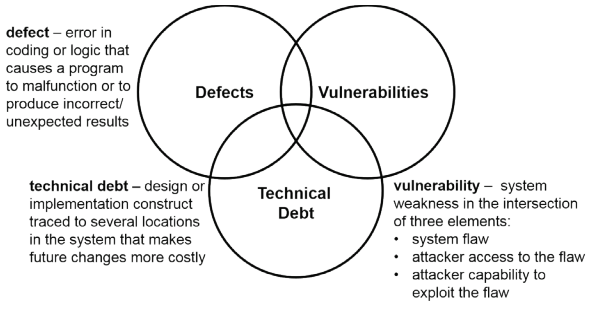

Figure 1. Software anomalies.

Published in the Proceedings of the 2016 Workshop on Cyber Security Experimentation and Test

symptoms (i.e., crashes or memory corruption) to their root cause, or to discover design flaws that are indicative of certain classes of vulnerabilities.

In this paper, we study the relationship between software vulnerabilities and technical debt to address the following question: Are software components with accrued technical debt more likely to be vulnerability-prone?

To answer this research question, we must examine a system's technical debt and software vulnerability history. In this paper, we present our experimental approach and preliminary analysis. We also reflect on the lessons learned with respect to experimentation, measurement, and data sources, using Release 17.0.963.46 of the Chromium open source project as an experimental test bed. We data mined the textual comments from the developers in the issue tracker, applied static analysis to the code, and extracted development data such as churn from the commit histories to find evidence of technical debt and vulnerabilities. A summary of our preliminary findings includes the following:

Finding 1: When they address security issues, software developers use technical debt concepts to discuss design limitations and their consequences on future work.

Finding 2: Correlations between vulnerabilities and technical debt indicators warrant further research.

Finding 3: One time-consuming relationship between vulnerabilities and technical debt is tracing a vulnerability to its root cause when it is the result of technical debt.

While these findings may corroborate the opinions of many developers, the contribution of this paper is to provide hard evidence in support of them. The rest of the paper describes our approach and initial findings, including threats to validity and an overview of our next steps.

2. Related Work

Existing approaches for finding anomalies in software focus on different ways of abstracting artifacts of software development and different scales of analysis.

There are many forms of automated vulnerability discovery, including dynamic analysis [1][8][26][29] and static analysis [5][6] tools. However, these tools have scalability limitations that manifest themselves in different ways. Dynamic analyses tend to attain poor coverage in larger systems and result in identifying only a subset of vulnerabilities. In contrast, static analysis systems usually over-approximate, which can result in an unmanageable number of false positives for large systems. There is also increasing focus on understanding the design and architectural sources of security issues in general [18] [25][28].

Research has shown mixed results at finding relationships between traditional defects and software vulnerabilities or common techniques for detecting both defects and vulnerabilities [7][12][27]. Increasingly, software development organizations are finding that a large number of vulnerabilities arise from design weaknesses and not from coding problems. MITRE released its list of 25 most dangerous software errors in 2011, and approximately 75 percent of these errors represented design weaknesses.

Static analysis of code quality has been used to analyze design issues [17][21]. Recent progress demonstrates that identifying design patterns or anti-patterns (i.e., abstractions that capture descriptions of how classes are designed or interact with one another, such as god classes, unstable interfaces, cycles between classes) can help developers locate and manage defects [20][23]. However, this is in contrast to the effectiveness of lower-level measures of design properties (e.g., average depth of inheritance trees). An analysis of Microsoft data showed that design measures such as Max FanIn/FanOut, Max InheritanceDepth, or Max ClassCoupling exhibited fairly low correlations with vulnerabilities [31].

When applied strategically and managed well, technical debt shows promise in accelerating design exploration and yields short-term market benefits. However, results of recent empirical research demonstrate that allowing design-related issues to accumulate into technical debt can result in unanticipated side effects when these issues are not remedied early [14][22][24].

While this related work has produced results relevant to analyzing software anomalies, challenges remain in timely detection and resolution. This research focuses on the relationship of technical debt and software vulnerabilities to increase our understanding of this area in the space of software anomalies, bringing to bear knowledge of design and architecture abstractions.

3. Approach

To understand whether software components with accrued technical debt are more likely to be vulnerability-prone, we need to take into account data from multiple sources: quantitative static analysis of anomalies (faults, vulnerabilities, design flaws) in code, qualitative classification of design consequences in issue trackers, and software development indicators in the commit history.

3.1. Technical Approach

Our technical approach is as follows:

1. Identify software vulnerabilities.
    a. Enumerate issues in the Chromium issue tracker [9] that have the `security` label.

b. Classify each issue in terms of its Common Weakness Enumeration (CWE) using the issue's description, comments, metadata, and patch.

c. For each issue, identify the set of files changed by commits that reference the issue.

2. Identify technical debt.
   a. Classify issues for technical debt.
   b. Classify the type of design problem and rework based on the issue description, comments, and metadata.
   c. Detect design flaws that co-exist in the same files changed to fix the issues labeled `security`.
3. Model the relationships between technical debt issues and vulnerabilities in the common artifacts they represent (code files, issues, commits).
   a. Extract concepts related to vulnerability types.
   b. Test whether technical debt indicators (e.g., number and type of design flaws, number of traditional bugs, number of bugs labeled security, and the lines of code that change to fix a bug) correlate with the number of vulnerabilities reported.
   c. Manually investigate how selected vulnerabilities are influenced by the correlated technical debt indicators.

We will show how design knowledge can help identify other related issues and files so that developers can more efficiently diagnose the root cause of vulnerabilities and provide a long-term fix.

3.2. Prioritizing Security Issues

The results of the analysis are meant to support better decision making by assisting in locating the design roots of vulnerabilities. Thus we are also interested in the question: If we can validate that software components with accrued technical debt are more likely to be vulnerability-prone, then can we use this information to assign priorities to vulnerabilities?

The Chromium project (and many others) attempts to fix all outstanding security issues, but there are often many security issues open at a time. Chromium developers are accustomed to setting a priority based on a number of factors, such as the potential damage if the issue is exploited and whether the issue is known to the broader public.

Setting the priority of each issue is not trivial, however, because the symptoms of a security issue may not give insight into the root cause. Unfortunately, finding the root cause can often be the most time-intensive part of resolving an issue. We envision developers supplementing their review of the issue backlog with this information about design implications to prioritize their work assignments and diagnose the problem.

4. Classification Results

We are working with a data set from the Chromium open source project [2][10][11]. This is a complex web-based application that operates on sensitive information and allows untrusted input from both web clients and servers. We use it as a representative test bed of typical technical debt issues and types of vulnerabilities. The Chromium open source project released Version 17.0.963.46 (referred to as Chromium 17 from here on) on February 8, 2012. This release contained 18,730 files. From February 1, 2010, to February 8, 2012, there were 14,119 bug issues reported as fixed [9].

4.1. Identify Software Vulnerabilities

To identify vulnerabilities, we used the issues labeled `security`. Using the Chromium project's issue tracker [9], we identified 79 software vulnerability issues, which were related to 289 files in which we detected design flaws (described in the next section). An issue labeled `security` may have a well-identified security bug, such as a null pointer exception. Such an issue may not represent technical debt but could simply be an implementation oversight. On the other hand, some issues may manifest themselves with multiple symptoms. This can hint that technical debt contributed to the vulnerability.

4.2. Identify Technical Debt

We apply a classification approach to classify issues as either related to technical debt or not, and we classify source code files as containing debt if they have design flaws (e.g., unstable interface, API mismatch, and package and class cycles) [30].

To classify issues, we applied a classification approach that we developed to analyze a project issue repository and tag issues as technical debt based on developer discussions of design limitations and accumulating rework. Experts apply unspoken heuristics when determining whether an issue represents technical debt. Our goal in developing the technical debt classification was to capture their expertise and allow repeatable classification of issues [4]. Multiple researchers applied this classification to the 79 security issues to determine whether some of the security issues also carry technical debt.

To classify source code files, we used the results of a study that analyzed Chromium 17 and reported 289 files associated with design flaws that can be detected in the code. The approach analyzes a project's repositories—

its code and its revisions—to calculate a model of the design as a set of design rule spaces (DRSpaces) [30]. These DRSpaces are automatically analyzed for design flaws that violate proper design principles.

Four types of design flaws can be identified from the DRSpace analysis: modularity violation, unstable interface, clique, and improper inheritance. A modularity violation occurs when files with no structural relation frequently change together. This suggests that those files share some secret or knowledge and that information has not been encapsulated or modularized. An unstable interface occurs when there is an important class or interface that many other files depend on, and this class is buggy and changes frequently, requiring its "followers" to also change. Clique refers to a cross-module cycle that prevents groups of modules from being independent of each other. Improper inheritance occurs when the parent class depends on the child or when another file depends on both a parent and its child class. We consider these flaws as indicators of technical debt.

4.3. Design Flaws and Technical Debt Classified as Vulnerability Issues

Table 1 shows the combined results of our classification of the issues based on security labels, technical debt categories, and design flaws in the related code.

We manually classified 15 issues as technical debt for which the DRSpace analysis also found evidence of design flaws. In their discussion of the issue, developers demonstrated that they were aware of design problems and longer-term consequences related to these issues. These are the hard problems that take significant time to diagnose. It makes sense to prioritize these from a risk perspective since work is likely to accumulate over time if a fix is delayed.

Static analysis found no design flaws in 6 issues that we tagged as technical debt. If we only ran static analysis to detect these anomalies, these issues would be missed (representing the false negatives). Thus it appears that we need to supplement static analysis with human-based knowledge of design limitations. Running tools first can focus the attention of the experts (who are a limited resource) to parts of the problem not already addressed by the tools.

Table 1. Design flaws and issues classified as technical debt.

	Classified Not TD	Classified TD
No Design Flaw	8	6
Design Flaw	50	15

We classified 8 issues as basic coding flaws, where static analysis reported no design flaws, and we tagged these issues as not containing technical debt. These are local problems that developers know how to fix based on the observed symptoms. The effort to fix such bugs does not accumulate extra work over time.

Static analysis found design flaws in 50 issues that we classified as not containing technical debt. Of these, 23 issues showed partial evidence of technical debt in the form of a design problem, but they lacked evidence of rework or additional accumulation for making a definitive diagnosis of technical debt. These could be areas where static analysis is too sensitive (representing false-positive indicators for technical debt) or where the debt has not yet manifested itself. The other 27 were classified as bugs (incorrect functionality), indicating that the developers did not comment on an underlying design issue. If missed, the technical debt remains despite the local fix, and the problem is likely to surface again over time.

There is an opportunity here for making the design root cause visible to the developers to aid them in diagnosing and fixing the problem.

5. Preliminary Design Concept Analysis

We have preliminary results from the third step of our approach, in which we extracted design concepts related to vulnerabilities, looked for overarching correlations, and manually examined security issues to assess the impact of the design root cause. We use Issue 10977: *Crash due to large negative number* as an example throughout this section to illustrate the data we analyzed to prioritize security issues from a design perspective. The manual inspection revealed that the issue description of this security issue contained technical debt concepts, and the DRSpace analysis also traced the issue back to design flaws in the code.

Getting to the root cause of an issue can lead us to find related vulnerabilities (traced to the same root) in the backlog of issues and prevent new symptoms from surfacing in duplicate vulnerability reports. The example in this section illustrates that such analysis necessitates using code, issue trackers, and commit history in concert.

5.1. Vulnerability and Technical Debt Concepts

Symptoms of software vulnerabilities are visible in the development artifacts. These symptoms may be visible to the user in the form of a bug, and the user can therefore provide information in the bug issue report. Or the symptoms may be visible only to developers via the results from a static analysis tool, crash trace, or automated checks such as Chromium's ClusterFuzz tool.

Table 2 summarizes the vulnerabilities of those issues that also reported design problems from the 79 issues we classified, in the form of CWE categories [13]. Although the initial description in the problem report provides guidance on isolating the problem, we see developers discussing design problems related to overarching security and sustainability concerns as they seek to replicate the problem, arrive at a diagnosis, and provide a fix.

In our example, evidence of an integer overflow vulnerability can be traced to a design concern with external dependency. This is an occurrence of *CWE-703: Improper Check or Handling of Exceptional Conditions* in the *boundary conditions* group in Table 2. There is also a discussion of the consequences of alternative solutions to motivate looking beyond the local fix to the root cause.

> *"We could just fend off negative numbers near the crash site or we can dig deeper and find out how this -10000 is happening."*
>
> *"If we patch it here, it will pop-up somewhere else later."*

Developers addressing security issues are using concepts related to technical debt (italicized):

- getting to the *root cause*
- understanding the *underlying design* issues
- recording symptoms where changes are taking *longer than usual* or problems are *reoccurring*
- predicting consequences for the *longer term*
- building evidence for a more *substantial fix*

In a few cases, developers are looking at technical debt holistically over its lifetime. They are aware of potential technical debt at the time a short-term fix is made, tracking changing priority (in relation to security severity over time), balancing available resources (by grouping related issues and moving to later milestones), and reusing previous investments in a fix (and justifying the investment where it is used multiple times).

Finding: Software developers use concepts related to technical debt—such as getting to the root cause, discussing consequences of immediate patches over the longer term, and building evidence for a more substantial fix—to address security issues.

Table 2. Affinity groups of vulnerability types.

Affinity	CWE	#Issues
interface	200: Information Exposure	1
resource arbitration	362: Concurrent Execution using Shared Resource with Improper Synchronization	3
	400: Uncontrolled Resource Consumption	3
invalid result	20: Improper Input Validation	2
	451: User Interface (UI) Misrepresentation of Critical Information	2
	476: NULL Pointer Dereference	1
	704: Incorrect Type Conversion or Cast	1
	825: Expired Pointer Dereference	1
boundary conditions	125: Out-of-bounds Read	1
	703: Improper Check or Handling of Exceptional Conditions	4
	787: Out-of-bounds Write	2
privilege	250: Execution with Unnecessary Privileges	2
	269: Improper Privilege Management	1
	285: Improper Authorization	1

5.2. Technical Debt and Vulnerability Correlations

Technical debt indicators include static code analysis measures such as number and type of design flaws, number of traditional bugs, number of bugs labeled `security` (security bugs), and the lines of code that change to fix a traditional bug or security bug (bug churn/security churn) observed during development.

Table 3 shows the results of a prior study that computed the Pearson correlation coefficient between design flaws and (1) number of bugs, (2) bug churn, (3) number of security bugs, and (4) security churn for the issues and files in the data set we studied [16]. The issues labeled `security` (security bugs) and the number of changes required in the corresponding files (security churn) show a correlation with design flaws.

Our further analysis shows that for three of the four types of design flaws (modularity violation, clique, and improper inheritance), files with vulnerabilities are more likely to have design flaws as well. The more types of design flaws a file is involved in, the higher the likelihood of it also having vulnerabilities. The rate is very low, however, in this exploratory data set. We are in the process of replicating the analysis as well as running the same approach through additional data sets.

Table 3. Pearson correlation coefficient between number of design flaws and number of bugs, bug churn, number of security bugs, and security churn.

Bugs	Bug Churn	Security Bugs	Security Churn
0.921	0.908	0.988	0.826

Finding: We see evidence of correlations between vulnerabilities and technical debt indicators such as design flaws and code churn: the more types of design flaws a file is involved in, the higher the likelihood of it also having vulnerabilities; files with vulnerabilities also tend to have more code churn.

5.3. How Vulnerability Analysis Is Influenced by Technical Debt

Returning to our example of *crash due to large negative number,* we see from the developer comments in the issue tracker that it took some time to analyze the problem. Three distinct users initially submitted reports of the crash, which were eventually merged into a single issue. Within a day, developers were able to reproduce the problem and pose a local fix to the related files involved in the integer overflow that caused the crash:

> *"We could just fend off negative numbers near the crash site or we can dig deeper and find out how this -10000 is happening."*

Treating issues one at a time can lead to situations where developers create a localized patch every time they encounter the similar integer overflow issue. Developers are aware of these matters and express their concerns:

> *"Time permitting, I'm inclined to want to know the root cause. My sense is that if we [only] patch it here, it will pop-up somewhere else later."*

Weeks later, as they worked on the problem, the developers noted additional reports of crashes:

> *"There have been 28 reports from 7 clients ... 18 reports from 6 clients."*

They thought that the problem was fixed when the crash did not occur in a more recent release, but further analysis showed the problem remained:

> *"Hmm ... reopening. The test case crashes a debug build, but not the production build. I have confirmed that the original source code does crash the production build, so there must be multiple things going on here."*

Apparently the true root cause of the problem was not found and fixed. Often this is because the developer does not understand the non-local consequences of a bug fix. Defective files seldom exist alone in large-scale software systems [20]. They are usually architecturally connected, so a fix to one source file may cause a problem in another part of the system.

In our example, we classified multiple issues related to integer overflow as technical debt. We traced the design cause of several of these issues to an external package used by Chromium whose API calls inject an out-of-bounds number resulting in crashes related to integer overflows.

In this example, our design flaw analysis supplemented developer knowledge recorded in the issue tracker and detected that one of the files participates in design flaws of cross-module cycles and improper hierarchy that violate architecture principles. The improper hierarchy also has a causal relationship with bug churn and provides additional evidence of technical debt.

Sixteen files participate in the original problem. Identifying the design source that is related to the external component brings in eight more files that provide a more accurate picture of the impact of the problem. Knowing the actual design source, the developers can fix the problem once at the source, rather than patching the files where the API calls create the crash only to see the problem resurface in other files where more API calls create more crashes.

Our vision is to collect evidence from software repositories that allows an analyst to locate the root cause of vulnerabilities, to determine whether those vulnerabilities are caused by improper design decisions.

Finding: One time-consuming relationship between vulnerabilities and technical debt is tracing the vulnerability to its root cause when it is caused by technical debt.

6. Threats to Validity

In this paper we report our initial results from analyzing a complex system for vulnerabilities and technical debt. Our early findings are promising despite the small sample size and partial manual nature of our analysis. We report the threats to validity and how we will address them as we continue our study.

Data quality and size: While Chromium is a large project, for the initial analysis we chose to focus on one version, Version 17, and limited our analysis to the 289 files for which we found design flaws. This is a small subset compared to the overall size of the Chromium project. However, our goal was to set up the experimentation environment, refine our methodology, and investigate whether we could find useful examples of debt. Given our promising results, we are in the process of replicating the analysis and extending it to other versions of Chromium to address the threat to data quality and size, in particular against potential errors related to data extraction.

Manual inspection: Manual inspection of the issues is crucial initially, especially in an exploratory step as we have reported here. It serves as input for creating key concepts and decision criteria for selecting artifacts to

analyze. To counter the threat of making classification and interpretation mistakes, multiple researchers independently tagged the issues and then presented the results to other researchers for review and discussion.

Identification of technical debt and vulnerabilities: We identified technical debt in two ways: manual classification and automated detection of modifiability-related design flaws [30]. We identified vulnerabilities relying on the `security` label that the Chromium 17 developers use. It is possible that we may not have included all relevant issues and files representing flaws. Technical debt is not limited to only the four design flaws we chose to analyze. There may be other design flaws and vulnerabilities that our data set missed. We are addressing this threat for our ongoing analysis by establishing replication criteria for the analysis.

7. Conclusions and Future Work

The goal of this preliminary analysis was to answer the question "Are software components with accrued technical debt more likely to be vulnerability-prone?"

Due to the small number of security issues in the Chromium sample that we examined, correlating raw numbers tends not to be useful. (The relative scarcity of vulnerabilities in real software systems is also remarked upon by other authors [31].) The preliminary analyses presented here are based on differences between files linked to vulnerability issues and those that are not. Measurement requires quantitative static analysis of anomalies (faults, vulnerabilities, design flaws) in code and qualitative issue classification of design consequences. We have seen that combining evidence can provide context and help prioritize the number of issues generated from static analysis in the effort to reduce the number of false positives and focus on what is important.

It is interesting that in a related study that attempted to correlate design metrics with vulnerabilities in a Microsoft system, metrics related to churn or editing frequency also were among those with the highest correlations [31]. In our study, we noted that churn and editing frequency can indicate technical debt, since components with debt are expected to be involved more often in changes and defect fixes.

Refining these conclusions and identifying the most useful predictors of technical debt require more analysis on a larger data set. We will look further into the characteristics of the data to answer the following questions: Are some forms of technical debt more closely related to vulnerabilities than others? What types of vulnerabilities correlate with technical debt? We will experiment with the modifiability and security taxonomies [3][13] to understand which taxonomies are feasible and supply a useful level of insight. Ultimately we would like to codify known sources of security-related technical debt as design flaws that tools can analyze for with increasing accuracy.

7. Acknowledgments

 This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

We thank Tamara Marshall-Keim for her expert input.

8. References

[1] Avgerinos, T., Cha, S.K., Rebert, A., Schwartz, E.J., Woo, M., Brumley, D. Automatic exploit generation. *Communications of the ACM*, 57(2): 74–84, 2014.

[2] Barth, A., Jackson, C., Reis, C., Google Chrome Team. *The Security Architecture of the Chromium Browser*. Stanford Web Security Research, 2008.

[3] Bass, L., Clements, P., Kazman, R. *Software Architecture in Practice*. Addison-Wesley, 3rd edition, 2012.

[4] Bellomo, S., Nord, R.L., Ozkaya, I., Popeck, M. Got technical debt? Surfacing elusive technical debt in issue trackers. *Proceedings of the 13th International Conference on Mining Software Repositories*, 327–338. ACM, 2016.

[5] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., et al. A few billion lines of code later: Using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2): 66–75, 2010.

[6] Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., et al. A static analyzer for large safety-critical software. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 196–207. ACM, 2003.

[7] Camilo, F., Meneely, A., Nagappan, M. Do bugs foreshadow vulnerabilities? A study of the Chromium Project. *Proceedings of the 12th Working Conference on Mining Software Repositories*, 269–279. ACM, 2015.

[8] Cha, S.K., Avgerinos, T., Rebert, A., Brumley, D. Unleashing mayhem on binary code. *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, 380–394. IEEE, 2012.

[9] Chromium Issues. https://code.google.com/p/chromium/issues/list

[10] Chromium Project, Browser Components. http://www.chromium.org/developers/design-documents/browser-components

[11] Chromium Project, Security Overview. http://www.chromium.org/chromium-os/chromiumos-design-docs/security-overview

[12] Clark, S., Frei, S., Blaze, M., Smith, J. Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities. *Proceedings of the 26th Annual Computer Security Applications Conference*, 251–260. ACM, 2010.

[13] Common Weakness Enumeration. https://cwe.mitre.org/about/sources.html

[14] Ernst, N., Bellomo, S., Ozkaya, I., Nord, R.L., Gorton, I. Measure it? Manage it? Ignore it? Software practitioners and technical debt. *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, 50–60. ACM, 2015.

[15] Falessi, D., Reichel, A. Towards an open-source tool for measuring and visualizing the interest of technical debt. *Proceedings of the Seventh International Workshop on Managing Technical Debt*. IEEE, 2015.

[16] Feng, Q., Kazman, R., Cai, Y., Mo, R., Xiao, L. Towards an architecture-centric approach to security analysis. *Proceedings of the 13th Working IEEE/IFIP Conference on Software Architecture*. IEEE, 2016.

[17] Fontana, F.A., Ferme, V., Spinelli, S. Investigating the impact of code smells debt on quality code evaluation. *Proceedings of the Third International Workshop on Managing Technical Debt*, 15–22. IEEE, 2012.

[18] IEEE Center for Secure Design. Avoiding the Top 10 Software Security Design Flaws, 2015. http://cybersecurity.ieee.org/images/files/images/pdf/CybersecurityInitiative-online.pdf

[19] IEEE Standard 1044-2009: IEEE Standard Categorization for Software Anomalies. IEEE Computer Society, 2009.

[20] Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., et al. A case study in locating the architectural roots of technical debt. *Proceedings of the 37th IEEE International Conference on Software Engineering*, 179–188. IEEE, 2015.

[21] Kim, M., Notkin, D. Discovering and representing systematic code changes. *Proceedings of the 31st International Conference on Software Engineering*, 309–319. IEEE, 2009.

[22] Kruchten, P., Nord, R., Ozkaya, I. Technical debt: From metaphor to theory and practice. *IEEE Software*, 29(6): 18–21, 2012.

[23] Letouzey, J.-L., Ilkiewicz, M. Managing technical debt with the SQUALE Method. *IEEE Software*, 29(6): 44–51, 2012.

[24] Li, Z., Liang, P., Avgeriou, P., Guelfi, N., Ampatzoglou, A. An empirical investigation of modularity metrics for indicating architectural technical debt. *Proceedings of the 10th International ACM SIGSOFT Conference on Quality of Software Architectures*, 119–128. ACM, 2014.

[25] McGraw, G. Four software security findings. *Computer*, 49(1): 84–87, 2016.

[26] Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., et al. Optimizing seed selection for fuzzing. *Proceedings of the 23rd USENIX conference on Security Symposium*, 861–875. USENIX, 2014.

[27] Shin, Y., Meneely, A., Williams, L., Osborne, J.A. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6): 772–787, 2011.

[28] Wolf, D. Science of Security and Value of Secure Design. http://cps-vo.org/node/24890, 2016.

[29] Woo, M., Cha, S.K., Gottlieb, S., Brumley, D. Scheduling black-box mutational fuzzing. *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, 511–522. ACM, 2013.

[30] Xiao, L., Cai, Y., Kazman, R. Design rule spaces: A new form of architecture insight. *Proceedings of the 36rd International Conference on Software Engineering*, 967–977. ACM, 2014.

[31] Zimmermann, T., Nagappan, N., Williams, L. Searching for a needle in a haystack: Predicting security vulnerabilities for Windows Vista. *Proceedings of the Third International Conference on Software Testing, Verification and Validation*, 421–428. IEEE, 2010.