2016 13th Working IEEE/IFIP Conference on Software Architecture

Towards an Architecture-centric Approach to Security Analysis

Qiong Feng*, Rick Kazman†, Yuanfang Cai*, Ran Mo*, Lu Xiao*

* Drexel University
Philadelphia, PA, USA
{qf28, yc349, rm859, lx52}@drexel.edu

†University of Hawaii & SEI/CMU
Honolulu, HI, USA
kazman@hawaii.edu

***Abstract*—Recently there has been increased attention to the consequences of architecture design decisions and their impact on security. Architectural design decisions have been identified as being critical for achieving high levels of software system security. However the majority of this research has been anecdotal and there are few tools or methods for understanding the architectural relations among files, and their impact on security. In this paper we employ a DRSpace-based analysis approach to identify architectural design flaws and we show, via an empirical study of 10 open source projects, that areas of a software architecture that suffer from greater numbers of design flaws are highly correlated with security bugs, and high levels of churn associated with those security bugs. Finally, we show that a specific type of design flaw—unstable interface—is correlated with the greatest increase in software security bugs.**

***Keywords*-software architecture; software security; design flaw**

I. Introduction

Currently the vast majority of the research, methods, and tools that address software security focus on secure coding and testing. However, it is difficult to achieve a high level of any system quality by focusing solely on coding and testing. Architectural issues can overwhelm even the most heroic coding efforts, as systems grow in size. After investigating numerous security issue fixes, we observed that many of the fixes made to security bugs are *ad hoc*: when security bug is reported, checks and filters are added to one or a few files to patch it; given that there is no guarantee that the fixes to these files were complete or sufficient, it is not uncommon that the same bugs are reopened over and over again, more and more files are patched to fix the same problem, and similar patches are added to more and more files.

The impact of the architectural relations among these files on software security, as well as the impact of security patches to architecture, were never fully understood. We hypothesize that architectural flaws may be one of the significant underlying causes of the difficulty of fixing security issues, and the *ad hoc* patches may deteriorate software architecture, which will incur more, and wide-spreading security issues.

While there has been increasing interest in the consequences of architectural design for security, thus far there has only been anecdotal evidence that architecture actually matters. Our research goal is to reveal the significant relationship between architectural design and security and, in particular, to show how architectural flaws are strongly correlated with high rates of security bugs.

To achieve this objective we utilize an architecture-centric analysis method and tool to analyze the relation between files involved in architectural flaws, and files that have proven to be most security-critical. First, we analyze a project's repositories—its code, its revisions, and its issues—and from this information we calculate a set of *Design Rule Spaces (DRSpaces)*—a new architecture model we recently proposed [21]—that collectively represent the architecture of a project. We can then automatically analyze these DRSpaces for a set of architectural flaws, which we call *hotspot patterns* [14] that are violations of proper design principles. The architectural flaws are a form of technical debt [5]. Research into technical debt has, until now, focused primarily on the modifiability [12] or performance of a system. Hotspot patterns is one way to present these architecture flaws. Second, after we detect these hotspots we show how these hotspots highly correlated with high rates of security bugs.

To take a concrete example, in the Apache Tomcat project, as we will show in section V, there are a small number of files that are consistently implicated in security bugs. Why is this? We hypothesize that it is because these files are architecturally connected via hotspots—in the Tomcat case the hotspots are *modularity violations* and *improper inheritance* [14]—which inevitably lead to bugs. And this is not a problem that is in any way unique to Tomcat. We have observed these same types of issues in every project that we analyzed.

As we will show in our empirical study of 10 open source projects, we find a very high correlation in *all* projects between rates of hotspots and security bugs. This is good news for a project manager or architect, in two ways: 1) by identifying these hotspots we can guide the project to focus their bug-fixing, refactoring, and quality assurance activities on the relatively small portion of the project that is causing the vast majority of security bugs, and 2) the hotspots not only identify where there are architectural design flaws, but also the reasons for those flaws, which aids in analyzing and fixing them.

The results presented in this paper is the first step towards our objective: identify the highest security risks in the architecture, and guide an architect or analyst in their assurance and refactoring activities. By adopting

978-1-5090-2131-4/16 $31.00 © 2016 IEEE
DOI 10.1109/WICSA.2016.41

a strategic (architectural) approach to locating potential security problems, we can constrain further fine-grained (more expensive) analyses. In this way, we can both identify potential vulnerabilities that code-based techniques alone may miss and also reduce overall security assurance costs by providing more global, systematic and reusable security solutions across the entire software.

II. Background

There are a number of threads of research that form the foundation for the work presented here. In each of the following subsections we will describe them and their influence on our current research.

A. Security Analysis

The Open-Source Vulnerability Database (OSVDB)[1], classifies software vulnerabilities into six categories: Buffer Overflow, Cross-Site Scripting (XSS), File Inclusion, Denial of Service (DoS), Cross-Site Request Forgery (CSRF) and SQL Injection. Substantial research has been conducted to detect and manage these vulnerabilities. For example, static code analysis [7] is one method that has proved effective to detect potential buffer overflows. By examining source code and applying certain rules (such as forbidding *strcpy()* in the C language) static code analysis can reduce the incidences of buffer overflow exploits.

Fuzzing [19] is another mature technique to reduce software vulnerabilities by sending invalid data to a software component to cause a fault and hence detect a vulnerability. For example, a protocol fuzzer can send malicious packets, including a forged IP address, to a server. In a three-way handshake to establish a connection the server may wait for an ACK signal from the forged IP address but will never receive it. An attacker can exploit this vulnerability and mount a denial-of-service attack. A protocol fuzzer can detect such a vulnerability to prevent exploits.

Penetration testing [1] is used to detect security problems at the system and unit level. For example, by applying penetration testing and creating test cases from risk analyses, security experts can detect many potential vulnerabilities such as improperly configured firewalls, attackable ports, etc.

A disadvantage of static code analysis, however, is that it only detects bugs in source code. It can not detect design flaws. And a disadvantage of fuzzing and penetration testing is that, while they may detect potential exploits, they provide no indication as to why the code is exploitable or how to fix it. Our approach in this paper is complementary to these techniques: it locates architectural design flaws that are potential sources of security bugs, and provides rationale for why these are flaws and hence how to fix them.

B. Security and Architectural design

The relationship between security and the design of a software architecture has been emphasized in recent years. For example, the CWE (Common Weakness Enumeration) database describes mitigations for each class of vulnerability and these include architectural solutions. CWEs are generalizations of groups of CVEs (Common Vulnerabilities and Exposures) which are publicly recorded and disseminated security vulnerabilities. The IEEE created the "Center for Secure Design"[2], several books on security design patterns have appeared over the past decade (e.g. [4] [9] [10]), and published methods for architectural analysis of security have begun to appear, e.g. [17].

While these are important contributions to our understanding of security and its relationship to design concepts, all of these techniques still rely heavily on the experience and skill of the designer and analyst.

C. Design Rule Spaces

Design rules [2] are the critical architectural decisions that decouple the rest of the system into independent modules. A design rule is usually manifested as an interface or abstract class. For example, if an Observer Pattern [11] is used, then there must exist an observer interface that decouples the subject and concrete observers into independent modules. As long as the interface is stable, addition, removal, or changes to concrete observers should not influence the subject. In this case, the observer interface is considered to be a *design rule*, decoupling the subject and concrete observers into independent modules. Similarly, if a system uses a pipe-and-filter pattern then all the filters are independent modules, connected by pipes. Reflected in code, the abstract *Pipe* class can be seen as an instance of design rule.

Since a system can use multiple design patterns, each pattern being led by a design rule and involving a set of files, we can consider that each pattern forms its own design space led by design rules, which we call a *Design Rule Space* (DRSpace). Thus we have proposed that an architecture should be viewed as a set of overlapping DRSpaces [21].

A DRSpace contains a set of files and a selected set of relations, such as inheritance, aggregation, or dependency. These files are clustered into a design rule hierarchy (DRH) [20] which reveals the dependencies between design rules and independent modules. A DRH structure has the following features: 1) the top layer of the hierarchy contains the most influential files in the system, such as important base classes, key interfaces, etc. These files are called the *leading* files. 2) Files in higher layers should not depend on files in lower layers. 3) Files within the same layer are grouped into mutually independent modules. If the system is designed with key architectural design rules, then the files containing these design rules will be among the leading files.

We represent a DRSpace using *Design Structure Matrix* (DSM) [2], a square matrix whose rows and columns are labeled with the files of the DRSpace in the same order. If a cell in row x, column y, $c : (rx, cy)$, is marked, it means that the file on row x is structurally related to the file on column y, or that they are evolutionarily coupled [5], i.e.,

[1] http://osvdb.org/

[2] http://cybersecurity.ieee.org/center-for-secure-design/

they changed together, as recorded in the revision history. The cells along the diagonal represent self-dependency. For example, cell $(r2, c1)$ in Figure 1 is marked with "*ex, cl*", which means *cassandra.dht.Range* "*extends*" and "*calls*" *cassandra.dht.AbstractBounds*. In Figure 2, cell $(r1, c9)$ is marked with "*,118*", meaning that these two files have no structural dependences, but changed together 118 times in the revision history. Similarly, cell $(r1, c2)$ is marked with "*dp,44*", meaning that *cassandra.config.DatabaseDescriptor* depends on *cassandra.utils.FBUtilities*, and they changed together 44 times.

Using our tool Titan [21], the user can view and manipulate DRSpaces. The DSM in Figure 1 presents a DRSpace clustered into a DRH with 2 layers: $l1 : (rc1 - rc16)$ and $l2 : (rc17 - rc24)$. No class from layer $l1$ depends on any class in layer $l2$, and all the modules in layer 2 are mutually independent from each other. If a module is complex, the DRH algorithm recursively applies clustering on it, attempting to find independent modules. For example, although the first 16 files form one big module, our DHR algorithm further split it into two modules, $m1 : (rc1 - rc6)$ and $m2 : (rc7 - rc16)$, and $m1$, in turn, is split into two mutually independent submodules: $m11 : (rc1 - rc5)$ and $m12 : (rc6 - rc6)$.

This DRSpace reveals a Strategy pattern, in which the abstract strategy class, $rc16$:*cassandra.locator.AbstractReplicationStrategy*, is the key interface and the design rule. This abstract class has several subclasses, each being a concrete strategy, including $rc18$:*cassandra.locator.OldNetworkTopologyStrategy*, $rc19$:*cassandra.locator.SimpleStrategy*, and $rc20$:*cassandra.locator.LocalStrategy*. The abstract class belongs to the first layer, and decouples the client classes, such as $rc21$ and $rc22$, from concrete strategies. In other words, both client classes and strategy subclasses depend on the abstract class (the design rule), but do not depend on each other, showing that this design follows the Liskov Substitution Principle: concrete strategies can substitute with each other at runtime without influencing the clients.

D. Hotspot Patterns

A DRSpace aids in identifying architectural design flaws—the most error-prone or change-prone file groups—because it reveals both structure and history information simultaneously. In our prior work we observed that there are just a few distinct types of flaws that contribute to technical debt [5]. And we showed that these flaws occur over and over, in both open-source and industrial software systems [14]. Based on Baldwin and Clark's design rule theory [3] and fundamental software design principles, we summarized these recurring flaws into five architecture *hotspot patterns* [14], which we have named: (1) *Unstable Interface*, (2) *Modularity Violation*, (3) *Improper Inheritance*, (4) *Cross-Module Cycle*, (5) and *Cross-Package Cycle*. The first four hotspot patterns capture relations among *files* while the last one is defined at the *package* level. Since this paper is focusing on the file level, *Cross-Package Cycles* are not included.

Figure 2 depicts some examples of these hotspot pattern instances[3]. For example, 14 classes ($rc2$-$rc5$, $rc8$-$rc9$, $rc11$-$rc17$) co-changed with the class in $rc1$, *cassandra.config.DatabaseDescriptor*, 10 or more times, suggesting that $rc1$ is not stable. Otherwise, other classes would only need to call its methods without co-changing with it so many times. The fact that these classes change together so frequently strongly suggests an architectural design flaw, which we call an *unstable interface* hotspot pattern.

Cell $(r1, c9)$ in Figure 2 exemplifies another hotspot pattern, which we call *modularity violation*: this cell is marked with ",118", meaning that there is no structural relation between *cassandra.config.DatabaseDescriptor* and *cassandra.config.CFMetaData*, but they have changed together 118 times as recorded in the revision history, suggesting that they have strong, but implicit dependencies.

Classes in $rc13$ and $rc14$ form an example of *improper inheritance*. As we can see from Figure 2, the child class *cassandra.io.sstable.SSTableReader* inherits from its parent class *cassandra.io.sstable.SSTable*. But the parent class also depends on the child and both classes changed together 68 times. Finally, if DRH clusters a set of files into two groups, but they are not mutually independent, it means that these files form a *Cross-Module Cycle*, the fourth type of hotspot pattern, that prevents the two groups from being mutually independent from each other.

Our prior study [14] showed that the files involved in these 4 types of file-level hotspots have significantly higher bug and change rates than average files in a project, even when normalizing for file length. Furthermore, the more hotspots a source file participates in, the higher its rate of bugs, changes, and churn (the number of lines of code committed to fix bugs and make changes) [14]. While correlation does not prove causation, this high level of correlation is consistent with software design theory and with our interviews of architects, suggesting that these hotspots are the root causes of technical debt. Based on our prior work, we now describe how we detect hotspots, and analyze the relation between these hotspots and security bugs.

III. Methodology

As stated in the introduction, the majority of research into the root causes of security problems has focused on secure coding, and the majority of effort on security problem detection has been focused on testing. We were interested in determining whether there were any predictable effects of architectural choices on security.

In attempting to understand the relationship between architectural design and security flaws, we decided to examine the correlation between security bugs and architectural flaws (hotspots) at the file level, for two reasons: First, this allows

[3]An *instance* of a hotspot is a concrete example of a hotspot pattern, found in a project.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
1 cassandra.dht.AbstractBounds	(1)	cl	cl																					
2 cassandra.dht.Range	ex,cl	(2)					cl																	
3 cassandra.dht.Bounds	ex,cl	cl	(3)				cl																	
4 cassandra.locator.TokenMetadata		cl		(4)			cl									cl								
5 cassandra.locator.NetworkTopologyStrategy				cl	(5)											ex,cl								
6 cassandra.service.WriteResponseHandler						(6)								cl		cl								
7 cassandra.service.StorageService		cl		cl			(7)	cl		cl	cl	cl		cl		cl								
8 cassandra.config.Schema								(8)		cl														
9 cassandra.service.Readclback								cl	(9)					cl		cl								
10 cassandra.config.KSMetaData										(10)						cl								
11 cassandra.dht.BootStrapper		cl		cl			cl	cl			(11)			cl		cl								
12 cassandra.service.StorageProxy	cl		cl	cl		cl	cl	cl	cl			(12)		cl	cl	cl								
13 cassandra.service.DatacenterWriteResponseHandler					cl	ex,cl							(13)	cl										
14 cassandra.db.Table				cl			cl	cl		cl				(14)		cl								
15 cassandra.service.DatacenterReadclback					cl				ex,cl					cl	(15)									
16 cassandra.locator.AbstractReplicationStrategy				cl		cl							cl			(16)								
17 cassandra.SchemaLoader								cl		cl							(17)							
18 cassandra.locator.OldNetworkTopologyStrategy				cl												ex,cl		(18)						
19 cassandra.locator.SimpleStrategy				cl												ex,cl			(19)					
20 cassandra.locator.LocalStrategy																ex,cl				(20)				
21 cassandra.cql.CreateKeyspaceStatement							cl									cl					(21)			
22 cassandra.thrift.ThriftValidation							cl	cl		cl				cl		cl						(22)		
23 cassandra.service.StorageServiceAccessor							cl																(23)	
24 cassandra.db.index.keys.KeysIndex							cl																	(24)

Fig. 1. DRSpace Clustered as a DRH by Structural Relations (cl : Call, ex: Extend)

UnstableInterface (column 1) ModularityViolation (column 9)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
1 cassandra.config.DatabaseDescriptor	(1)	dp,44	,14	,10	,10	,6	,14	,36	**,118**	,	,12	,12	,16	,42	,52	,30	,18	,4
2 cassandra.utils.FBUtilities	**dp,44**	(2)	,40	,4	,6	,10	,6	,12	,38	,12	,28	,14	,8	,24	,46	,28	,18	,6
3 cassandra.utils.ByteBufferUtil	**,14**	dp,40	(3)	,	,	,	,	,4	,10	,4	,20	,	,4	,10	,26	,4	,12	,
4 cassandra.service.WriteResponseHandler	**,10**	dp,4	,	(4)	,4	,6	,18	dp,22	,	,	,	,	,	,	,6	,	,	,
5 cassandra.locator.TokenMetadata	**,10**	,6	,	,4	(5)	,4	,10	dp,24	,	,	,8	,	,	,	,4	,	,4	,6
6 cassandra.locator.NetworkTopologyStrategy	,6	dp,10	,	,6	dp,4	(6)	,10	ih,22	,4	,	,	,	,	,	,16	,8	,	,
7 cassandra.service.DatacenterWriteResponseHandler	**dp,14**	dp,6	,	ih,18	,10	dp,10	(7)	,20	,	,	,	,	,	,	,6	,	,	,6
8 cassandra.locator.AbstractReplicationStrategy	**,36**	dp,12	,4	dp,22	ag,24	,22	dp,20	(8)	,6	,	,	,	,	,	,16	,10	,	,10
9 cassandra.config.CFMetaData	**,118**	dp,38	dp,10	,	,	,4	,	,6	(9)	,	,	,	,16	,36	,46	,56	,	,
10 cassandra.utils.GuidGenerator	,	dp,12	,4	,	,	,	,	,	,	(10)	,4	,	,	,	,4	,	,	,
11 cassandra.dht.RandomPartitioner	**,12**	dp,28	dp,20	,	,8	,	,	,	,	dp,4	(11)	,	,	,4	,16	,	,50	,
12 cassandra.utils.CLibrary	**,12**	dp,14	,	,	,	,	,	,	,	,	,	(12)	,4	,12	,	,	,	,
13 cassandra.io.sstable.SSTable	**,16**	,8	dp,4	,	,	,	,	,	ag,16	,	,	,4	(13)	**dp,68**	,10	,	,	,
14 cassandra.io.sstable.SSTableReader	**dp,42**	,24	dp,10	,	,	,	,	,	,36	,	,4	dp,12	**ih,68**	(14)	,22	,10	,	,4
15 cassandra.cli.CliClient	**,52**	dp,46	dp,26	,6	,4	,16	,6	,16	,46	,4	,16	,	,10	,22	(15)	,48	,14	,6
16 cassandra.thrift.ThriftValidation	**dp,30**	,28	dp,4	,	,	,8	,	dp,10	dp,56	,	,	,	,	,10	,48	(16)	,	,4
17 cassandra.dht.OrderPreservingPartitioner	**dp,18**	dp,18	dp,12	,	,4	,	,	,	,	,	,50	,	,	,	,14	,	(17)	,
18 cassandra.locator.PropertyFileSnitch	,4	dp,6	,	,	dp,6	,	,6	,10	,	,	,	,	,	,4	,6	,4	,	(18)

ImproperInheritance

Fig. 2. DRSpace with History Coupling Clustered as a DRH by Structural Relations (ag: Aggregate, dp: Depend, ih: Inherit)[14]

us to directly test the consequences of architectural flaws on security for each file. Second, this approach, when applied to bugs in general, had already revealed significant insights, as described in section II-D. Hence, based on our prior research, the hypotheses we formulated are as follows:

> *H1: The rate of security bugs affecting a source file is strongly correlated with the number of architectural flaws (hotspots) that file is implicated in.*

Additionally we wanted to know whether the number of hotspots affecting a file is correlated with the churn—the number of lines of code changed—for security-bug-fixing. Thus we formulated hypotheses H2:

> *H2: The amount of security-related churn affecting a source file is strongly correlated with the number of hotspots that file is implicated in.*

To test hypotheses H1 and H2 we extracted the source code, revision histories, and issue databases from 10 large, well-known open source projects: HTTP Server, PHP, Tomcat, Avro, Camel, CXF, Derby, Hadoop, Chromium, and HBase. However, of these projects, only the Chromium projects consistently indicated, in their issue-tracking system, whether

a bug was security-related. Three other projects, HTTP Server, PHP, and Tomcat, noted a CVE number in their issue-tracking system but this information was insufficient to adequately analyze a project for security problems, as we will show.

A. Classifying Security Issues

The projects that we chose as our subjects covered a wide variety of application domains. For example, Camel is a integration framework based on Enterprise Integration Patterns; CXF is a fully featured Web services framework; Hadoop is a framework for reliable, scalable, distributed computing; and HBase is Hadoop's database, a distributed, scalable, big data store.

Given that the majority of projects do not distinguish security bugs from other kinds of bugs (with the Chromium projects being a notable exception), we were unable to appropriately categorize the issues from the issue-tracking systems of Avro, Camel, CXF, Derby, Hadoop, HBase, HTTP Server, PHP, and Tomcat as security-related or not security-related. In fact, our analysis of over 100,000 project repositories from Github, a popular open source project management system, showed that only 1.5% of projects using a labeling system for tasks made a "security" label available to developers. What is worse, even though some open-source projects have a "security" labeling system, not all developers labelled these issues correctly, as they are open-source projects and employees have great freedom in their processes. Take PHP for example; from 2007 to 2010 there was just one issue labeled as "security bug" per year. All of these situations made it difficult for us to retrieve enough data directly from these projects' issue tracking systems. To address this problem—the paucity of security-related data—we built a security issue classification tool called SIM (Security Issues Miner) [8]. This classifier used natural language processing methods to derive feature sets from the unstructured text data of an issue-tracking system. Given this data-set we used a machine learning classification method (Naive Bayes classification) to determine whether each text snippet was likely to represent a security-related topic [8].

The issue-tracking repository for Chromium included 875 issues labeled as "security", dating from 3/10/13 to 1/9/15. The summary statements of these issues were extracted as positive examples of software security-related text snippets, while the summary statements of additional Chromium issues not tagged as "security" were used as negative examples. The data set used in this study contained 1874 text samples (875 security-related, 999 not security-related). The full experimental data set was distributed into 80/20% training/test distributions, resulting in randomized training sets containing 1499 samples and randomized test sets containing 375 samples. The performance of the SIM classifier from our experiments can be seen in Table I.

Overall, the trained SIM classifier performed very well at software security topic detection. The average precision (or positive predictive value), which measures the fraction of text snippets classified as "security" by our classifier that were proven to be correct classifications, was between 91% and 93% in our experiments, with precision rising slightly as we included bigram (adjacent word pair) features and trigram (adjacent word triplet) features. This gave us confidence that we could use the SIM classifier for further studies on the security implications of design flaws.

B. Empirical Study

Using the SIM classifier we were able to analyze projects that did not natively label security issues. Information about the projects that we ended up choosing for analysis is summarized in Table II. This shows the name and analyzed version of each of the 10 projects. It also shows each project's size, in terms of number of files and lines of code (LOC) (as determined by Understand[4]). Finally, for each project we show the number of commits and bugs that we collected, from the project's revision control system and issue-tracking system respectively.

Given this basis of information, we analyzed each project's data as follows: first, we reverse-engineered the project's source code to analyze all of the static relationships between each project's files (e.g. calling, inheritance, typing, etc.). Second, from the reverse-engineered relationships we built a DSM, clustered as a DRH, as described in section II-C. Third, from the revision history we built a history DSM where the historical co-change relationships between pairs of files are recorded (e.g. if two files A and B change together in the project's revision history 10 times, then the value of *cell(A,B)* is 10). Fourth, based on the output of the prior two steps—the DSM, clustered as a DRH, and the history DSM—we ran the hotspot detection algorithms in the Titan tool [14]. This process detects all instances of the hotspots in all the projects. From this information we are able to calculate the number of hotspot instances each file is involved in. Fifth, we applied the SIM classifier to identify all security-related bugs and further obtain the frequency of security-related bugs each file is involved in. The major processing steps and data flow for our analysis are shown in Figure 3.

From above analysis steps, we were able to establish the number of hotspot pattern instances each file is involved in and the frequency of each file's security-related bugs (i.e., those commits in the revision history that were applied to fix an issue identified as being security-related bugs). Using this information we calculated the average of the frequency of security-related bugs for all files with the same value of hotspot instances. That is, we calculated the average number of security-related bugs for files with 0 hotspot instances, 1 hotspot instance, 2 hotspot instances, and so forth.

In this way, we could calculate several sets of data for each project, as a means of determining the correlation between hotspot instances and various extrinsic measures of project quality. We calculated the number of hotspot instances per file, and then grouped all the files in the project into those with 0 hotspot instances, 1 hotspot instance, 2 hotspot instances,

[4] http://scitools.com/

TABLE I
RESULTS OF N-GRAM FEATURE STUDIES

Feature Space	Average Precision	Average Recall	Average F-Measure
S_1 (Unigrams, 12674 total features)	0.91 ± 0.026	0.89 ± 0.023	0.90 ± 0.019
S_2 (Unigrams + Bigrams, 25347 total features)	0.92 ± 0.022	0.88 ± 0.022	0.90 ± 0.015
S_3 (Unigrams + Bigrams + Trigrams, 38019 total features)	0.93 ± 0.017	0.88 ± 0.020	0.91 ± 0.016

TABLE II
CANDIDATE PROJECT CHARACTERISTICS

C: Commits; BI: bug issues; BF: fixed bugs

Project	Version	#Files	KLOC	#C	#BI	#BF
Avro	1.6.3	305	38	642	387	305
Camel	2.8.4	6678	391	7832	1494	2663
CXF	2.5.2	4535	429	5492	2429	2502
Derby	10.9.1.0	2716	634	6551	3375	1923
Hadoop	1.0.3	2102	296	6981	3847	482
HBase	0.94.0	1053	246	4969	3164	2911
Httpd	2.0.58	290	91	26790	68	844
PHP	4.4.6	1020	190	46340	51	2414
Tomcat	6.0.0	1103	158	24485	1372	3150
Chrome	17.0.963.46	18730	5,425	42406	14119	94758

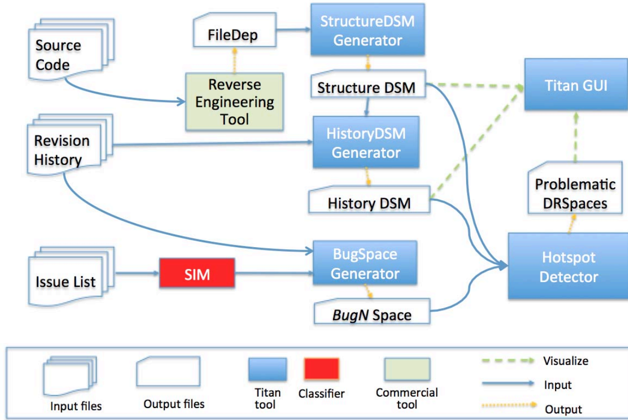

Fig. 3. Tool Chain to Detect Security Hotspots

and so forth. For each group we then calculated: the average number of bugs per file, the average number of security bugs per file (which are a subset of the total number of bugs), the average bug churn per file, and the average security churn per file. Finally, we calculated the Pearson correlation coefficient between these sets of data.

IV. RESULTS

In Table III we present correlations between the hotspot pattern instances and several extrinsic measures of project quality. The first result of note is that the correlations between hotspots and bugs are consistent with our prior research results that looked at bugs in general [14]. This strengthens our empirical evidence that these architectural flaws are indeed the root causes of high rates of bugs. Hence such flaws are an important source of modularity debt.

But in addition to strengthening the empirical basis for our prior results, we had a second, more specific research goal here: we were interested in knowing whether there is a positive correlation between hotspots and security bugs, and the churn related to these bugs. The results in Table III show that there is indeed a strong positive correlation between hotspots and security bugs and churn. The Pearson correlation coefficient between hotspots and security bugs ranges from a low of 0.861 (for Avro) to a high of 0.988 (for the Chrome browser), with an average of 0.92 over these 10 projects. The Pearson correlation coefficient between hotspots and security churn ranges from a low of 0.349 (for Hadoop) to a high of 0.970 (for Derby), with an average of 0.79 over these 10 projects. These results therefore provide strong empirical support for our hypotheses H1 and H2: architectural flaws (hotspots) are strongly correlated with high rates of security bugs and the churn associated with security bugs.

Furthermore, we can eliminate a possible explanation for this result: that security bugs are simply strongly correlated with bugs in general. Prior research [6] already showed that general bugs do not foreshadow, i.e. predict, vulnerabilities. In a study of more than 5 years of data from the Chromium project the correlation between pre-release bugs and post-release vulnerabilities was found to be weak. We found similarly weak correlations in our study of the 10 projects in our dataset, as shown in Table IV. Although some projects (e.g. Tomcat, PHP, Httpd) showed a moderate correlation between these two datasets, other projects (e.g. CXF, Chrome, Hadoop, Derby) showed far weaker correlations. Clearly one would expect *some* correlation, as vulnerabilities are themselves reported as bugs. But the lack of a consistently high correlation between bugs in general and security bugs in particular means that one can not reliably predict the other, and hence that they have different root causes.

A. Correlation between Hotspot Patterns and CVEs

We also calculated the same correlation between hotspots and CVEs, for three projects—Httpd, PHP, and Tomcat—that used the CVE numbers in their issue tracking database. These results of this analysis are shown in the last three rows of Table III.

Note that the correlations with CVEs shown in Table III are significantly lower for two of the three projects. In just one project—PHP—the correlation between CVEs and architecture flaws is strong and, in fact, stronger than the correlation with bugs determined by the SIM classifier. The question then arises: why, at least in two of the three cases that we studied, do we observe lower correlations between hotspot patterns and CVEs?

We postulate two reasons: First, it appears that most open source projects, even those few paying attention to

TABLE III
HOTSPOTS CORRELATIONS WITH BUGS, BUG CHURN, SECURITY BUGS, SECURITY CHURN AND CVES

Project	Bugs |Hotspots	Bug Churn |Hotspots	Security Bugs |Hotspots	Security Churn |Hotspots	CVEs |Hotspots
Avro	0.845	0.854	0.861	0.861	NA
Camel	0.957	0.964	0.958	0.919	NA
Chrome	0.921	0.908	0.988	0.826	NA
CXF	0.897	0.957	0.940	0.799	NA
Derby	0.938	0.959	0.897	0.970	NA
Hadoop	0.753	0.952	0.862	0.349	NA
HBase	0.894	0.911	0.962	0.618	NA
Httpd	0.607	0.878	0.885	0.713	0.689
PHP	0.929	0.832	0.924	0.914	0.987
Tomcat	0.901	0.830	0.921	0.902	0.776

TABLE IV
CORRELATIONS BETWEEN BUGS AND SECURITY BUGS

Project	Bugs|Security Bugs
Avro	0.507
Camel	0.594
Chrome	0.438
CXF	0.474
Derby	0.272
Hadoop	0.201
HBase	0.700
Httpd	0.772
PHP	0.750
Tomcat	0.790

security and tagging CVEs, do not tag CVEs consistently. However, the PHP project seems to be different: there appears to be a conscientious effort to tag security bugs using CVEs. For example, their changelog includes CVE information for virtually every release (e.g. see http://php.net/archive/2015.php). This rigorous attention to CVEs is clearly part of their project culture.

The fact is that the vast majority of open source projects do not tag security *at all* in their issue tracking system. And we suspect that many projects that are using CVEs are actually under-reporting their security bugs. For example, when we analyzed the CVEs reported in Chrome, we found a total of 980 issues labeled using the CVE tag. But we found 2175 issues—more than twice as many—labeled security in the same issue-tracking system. Part of the reason for this discrepancy is that CVEs are a subset of all security issues. CVEs are security issues that are publicly known and acknowledged vulnerabilities. But security bugs may emerge and cause problems for a project before they become publicly acknowledged. And some security bugs may be project-specific. Furthermore, it is often the case that many projects do not want to report security issues publicly. For all of these reasons, CVEs appear to be a poor mechanism for analyzing the security properties of most open source projects. Fortunately, using the SIM recognizer, we do not depend on project-specific labeling practices.

B. Correlations by Architecture Hotspot Pattern Type

Given these results—that the rates of hotspot pattern instances are strongly correlated with rates of security bugs—we were led to ask one final research question: which types of hotspot patterns are *most* harmful for security? To answer this question we calculated the average measures of security bugs and the churn of all files. Furthermore, we calculated the percentage increase in these measures in files participating in hotspot patterns, as compared with average files in each project.

For example, in the Avro project, the average file is implicated in 0.154 security bug and 1.544 lines of code (LOC) associated with security bugs (which we call "security churn"), over the time period that we studied. But the average file involved in an unstable interface hotspot pattern is implicated in just over 0.458 security bugs and almost 2.895 LOC, that is, triple the rate of a "normal" file. We calculated the "SBug_inc"("Security Bug Frequency Increase rate") as follows: *(0.458-0.154)/0.154*, which is 197.4%, as shown in the first row in Table V. In PHP the average file is implicated in 0.425 security bugs causing 481.6 LOC security churn, over the time period that we studied. But the average file involved in an unstable interface hotspot is implicated in just over 1.445 security bugs and almost 1239.6 LOC. We saw similar results in every project that we studied.

Given these values we can calculate the percentage increase in files associated with hotspots, for each project, for security-related bugs and churn. This calculation shows us how much more security-vulnerable files are if they participate in a hotspot pattern. The results for unstable interfaces, modularity violations, improper inheritance and cross-module cycle are shown in Table V. Note that the results for Improper Inheritance are not given for PHP and Httpd as these projects are largely written in C and hence seldom employ inheritance.

To better compare the increase rate of security bugs and churn for each hotspot pattern, we show the mean and standard deviation for all 10 projects' security bug frequency increase rates in Figure 4. The mean increase for unstable interfaces is 391.7%. In general, unstable interface is most problematic hotspot pattern. Similar results are shown for security churn increase rates in Figure 5. A large standard deviation means that there is a large difference of the security bug increase rate for different projects. We hypothesize that in some projects, such as Hbase, with lower increase rates, code errors such as inadequate bounds checking or lack of

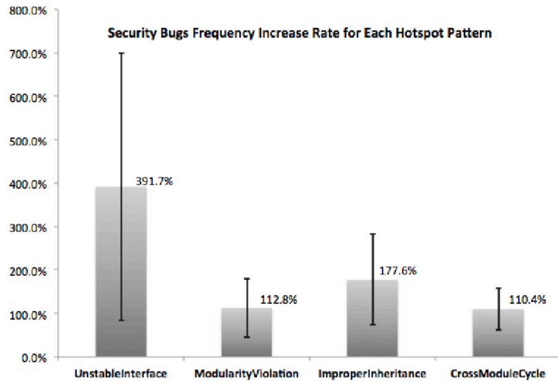

Fig. 4. Security Bugs Frequency Increase Rate for Each Hotspot Pattern

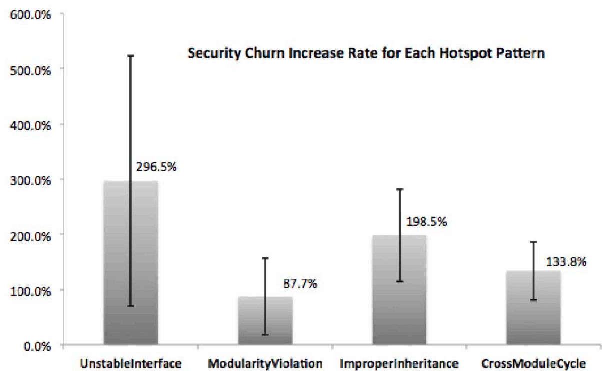

Fig. 5. Security Churn Increase Rate for Each Hotspot Pattern

input verification contribute a large percentage of security bugs and security churn. Since many security bugs are not caused by architectural flaws, the security bugs in hotspots may, on occasion, actually be lower than the average number of security bugs in a project. Although this case appears to be rare, we will explore the conditions under which it occurs in our future work.

V. Discussion

The results presented in section IV are unequivocal: Participating in a hotspot is highly correlated with bugs and changes in general (and the churn associated with fixing those bugs and making those changes) and with security bugs and CVEs in particular. While all hotspot types make things worse, as we can see from Table V, unstable interfaces are particularly problematic.

A. A Qualitative Example

To make this more concrete, let us examine a specific example, in the Apache Tomcat project. The DRSpace of interest is shown in Figure 6, led by the file *Context.java*. Three files in this DRSpace—*StandardContext.java*, *ContextConfig.java*, and *Context.java*—are involved in an architecture flaw: improper inheritance. *StandardContext* is a child of (and implements) the class *Context*. But the *ContextConfig* class calls both the parent class *Context* and the child class *StandardContext*, violating the well-accepted Liskov substitution principle. This means that *StandardContext* (the child class) has some functionality needed by the client, which the parent class, *Context*, does not have. As a result, *ContextConfig* has no option but to call both classes' methods.

These three classes are also involved in security issues. In the project's revision history, *ContextConfig.java* changed together with the child class 35 times, and with the parent class, *Context.java*, 23 times. The parent and the child also changed together 64 times. It is clear that—because of their improper architectural relations—security bugs, and bugs in general, are propagating among these files.

Furthermore, as shown in Figure 6, we found that *ContextConfig* has 8 CVE-related bugs and 224 security-related bugs. It is implicated in the most CVE-related bugs and the second-most security-related bugs of all 1103 files in Tomcat. *StandardContext* has 5 CVE-related bugs (second-most in the project) and 263 security-related bugs (the most in the project). The child class has far more security bugs than the parent class, which is implicated in 2 CVE-related bugs and 62 security-related bugs.

But the problems, and design flaws, do not end here. *ContextConfig.java* and another file—*TldConfig.java* (not shown in Figure 6)—are involved in a separate architecture flaw: a modularity violation. These two files have co-changed together 31 times in the project's revision history, while they have neither static nor dynamic dependencies (e.g., associated with an XML file). When we examined the revision history, we found revision messages for these co-changes including: "*Don't silently swallow Throwables that need to be re-thrown*", and "*fix for memory leak that aligns ContextConfig with TldConfig*". This suggests that these two files suffer from the same problems: not properly handling exceptions and suffering from the same memory leak, and hence they are frequently patched together. Since these two files have no structural dependency, new developers may neglect to patch one file when patching the other. This particular architecture flaw can cause additional security bugs if these vulnerabilities are disclosed but patching is incomplete. This, in fact, appears to be the case: as we stated above, *ContextConfig.java* suffers from 8 CVE-related bugs and 224 Security-related bugs and *TldConfig.java* has 1 CVE bug and 43 security-related bugs.

These cases illustrate the importance of architectural relations: instead of being isolated, security bugs propagate through architectural relations among files, and may impact large numbers of files. For example, there are 115 security bugs involving patching of *ContextConfig.java* and other files. The average number of other files that changed together with *ContextConfig.java* is 12.9, meaning that none of these security patches were localized or encapsulated. These ad hoc patches, over time, may deteriorate the architecture and the degraded architecture will further propagate security issues. The fact that the same set of files are involved in security issues repetitively calls for a systematic, architecture-level solution, rather than ad hoc patches as is commonly done today.

TABLE V
AVERAGE INCREASE IN SECURITY BUGS AND SECURITY CHURN IN 4 HOTSPOT PATTERNS

Project	Unstable Interface		Modularity Violation		Improper Inheritance		Cross-Module Cycle	
	SBug_inc	SChurn_inc	SBug_inc	SChurn_inc	SBug_inc	SChurn_inc	SBug_inc	SChurn_inc
Avro	197.4%	87.5%	52.7%	12.4%	213.3%	125.5%	77.7%	56.5%
Camel	475.7%	505.9%	60.8%	50.5%	377.4%	272.2%	123.7%	125.3%
CXF	1035.5%	413.5%	126.1%	142.1%	161.1%	180.6%	72.0%	115.5%
Derby	403.8%	246.7%	90.6%	30.9%	68.8%	195.5%	76.7%	118.7%
Hadoop	797.1%	824.1%	227.5%	140.6%	232.4%	348.8%	64.8%	93.7%
HBase	101.4%	259.8%	27.2%	-22.8%	62.2%	233.7%	64.4%	253.5%
Tomcat	110.6%	122.2%	102.2%	96.6%	102.4%	100.1%	147.4%	166.0%
PHP	240.5%	157.4%	224.7%	185.8%	NA	NA	215.5%	158.8%
Httpd	191.7%	198.9%	130.1%	167.4%	NA	NA	134.3%	143.8%
Chrome	363.1%	148.7%	86.3%	73.9%	203.0%	131.5%	127.5%	106.4%
Average	**391.7%**	**296.5%**	**112.8%**	**87.7%**	**177.6%**	**198.5%**	**110.4%**	**133.8%**

# of Security-related bugs	# of CVE-related bugs		1	2	3
62 (Rank #34)	2 (Rank #15)	1. catalina.Context.java	(1)	, 64	, 23
263 (Rank #1)	5 (Rank #2)	2. catalina.core.StandardContext.java	Implement, 64	(2)	, 35
224 (Rank #2)	8 (Rank #1)	3. catalina.startup.ContextConfig.java	Call, 23	Call, 35	(3)

Fig. 6. DRSpace Led by Context.java

We have seen similar kinds of architectural flaws, leading to technical debt, in *all* of the projects that we have studied. Taken together, these analyses suggest that architectural flaws are strongly correlated with security bugs. Of course, we realize that correlation does not prove causation. The correlation evidence and the qualitative evidence such as the above Tomcat example can not prove that the architectural flaws that we detect are actually *causing* the high rates of security problems. But seeing that this high level of correlation persists across many projects, suggests that something worthy of investigation is occurring. Even if, for example, the correlation that we observe is due to a third (hidden) variable—perhaps coding practices, or poor project communications, or rapidly fluctuating requirements—our technique will identify the areas in the architecture that are afflicted and these can and should be made the focus of mitigation activities such as refactoring, inspections, code walkthroughs, or additional testing.

Furthermore, the correlations that we have observed make intuitive sense and have been validated in our prior empirical studies (e.g. [12], [14], [18]). If an architecture has many hotspots—cyclic dependencies, improper inheritance, modularity violations, and so forth—it is not surprising that that architecture will be challenging to understand, to modify, and to extend. Such a context "invites" bugs of all kinds, including security bugs.

B. Towards a New Approach

What are the implications for designing for security? The most obvious implication, is that design matters for security. As we stated in the introduction, the vast majority of effort and research into making systems secure goes into secure coding and testing techniques. Design has been given far less attention. Our research not only provides justification for paying attention to design decisions insofar as they affect security, but also points out specific types of flaws that degrade security—these are a specific kind of technical debt that we call modularity debt. Furthermore, by knowing the type of hotspot, we can plan targeted refactoring strategies (as described in [12] to "pay down" the modularity debt. If cycles exist, we need to break those cycles (typically moving some of the functionality from one file to another). If we find modularity violations, we need to modularize the "secrets" shared by the implicated files. If we find improper inheritance, we typically need to move some functionality from one or more child classes to the parent class. The point here is that more than just red-flagging a part of the architecture, knowing the hotspot type and the files involved helps us to construct evidence-based refactoring strategies.

Finally, we envision that a process and tool chain such as we have described here can be part of an architect's dashboard or manufacturing execution system ([13], [16]) that scans a project repository—perhaps nightly, perhaps after each checkin—and alerts the architect to new hotspots or areas of the architecture with increasing complexity [15] or concentrations of hotspots.

C. Threats to Validity

There are two notable threats to validity in this study. First, our dataset consists of just 10 projects, all of which are open source. While it is true that this restriction to open source projects may bias the results, we have not seen any evidence of bias in the results of our other DRSpace-based studies of more than 150 projects (120 open source and 30 closed-source). That is to say, there are no discernible differences, from the perspective of architectural complexity and hotspots, between open- and closed-source projects.

Another threat to validity is that we used SIM to identify

security issues. Clearly this has an effect on the data—the recognizer has been shown to be 93% accurate in our test datasets, which is excellent precision, but it still means that about 7% of our dataset represents mis-classified issues, that is, issues are classified as security-related when they are not. However while such mis-classifications do reduce confidence in the results, they should not introduce bias. There is no reason to assume that the mis-classified issues are somehow more or less related to architectural flaws than true issues.

VI. Conclusions and Future Work

To reiterate our main result: files that participate in architectural hotspots are highly correlated with bugs and changes in general and with security bugs and CVEs in particular, all of which contributes to high amounts of churn, a form of technical debt. And while all hotspot types contribute to this correlation, unstable interfaces are the most problematic.

This strong correlation makes intuitive sense: it is not surprising, in hindsight, that architecture flaws might lead to security problems, but no-one has ever proven this prior to our study. Most existing security analysis tools treat the system as a black box or they analyze the code but not the architecture and its myriad design decisions. Our approach, on the other hand, focuses entirely on the design decisions, as manifested in the relations between a system's files. Doing this analysis does not require access to the source code, just a knowledge of the structural and historical relations between files. And it can be completely automated, hence made part of continuous monitoring and continuous assurance.

We have two major thrusts for future work in this research area: 1) we want to develop (semi-) automated refactoring strategies, to aid the architect in removing the accumulated modularity debt; and 2) we want to perform longitudinal studies of projects to determine how security problems develop over time, how this relates to architecture problems, and how removing these problems affects project health.

Acknowledgment

This work was supported in part by the National Science Foundation of the US under grants CCF-0916891, CCF-1065189, CCF-1116980 and DUE-0837665.

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.

[Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003206

References

[1] B. Arkin, S. Stender, and G. McGraw. Software penetration testing. *IEEE Security & Privacy*, (1):84–87, 2005.

[2] C. Y. Baldwin and K. B. Clark. *Design Rules, Vol. 1: The Power of Modularity*. MIT Press, 2000.

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

[4] C. Blackwell and H. Zhu. *Cyberpatterns: Unifying Design Patterns with Security and Attack Patterns*. Springer, 2014.

[5] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 47–52. ACM, 2010.

[6] F. Camilo, A. Meneely, and M. Nagappan. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 269–279, 2015.

[7] B. Chess and G. McGraw. Static analysis for security. *IEEE Security & Privacy*, (6):76–79, 2004.

[8] C. A. Cois and R. Kazman. Natural language processing to quantify security effort in the software development lifecycle. In *Proceedings of the 27th International Conference on Software Engineering and Knowledge Engineering*, 2015.

[9] C. R. Dougherty, K. Sayre, R. Seacord, D. Svoboda, and K. Togashi. Secure design patterns. 2009.

[10] E. Fernandez-Buglioni. *Security patterns in practice: designing secure architectures using software patterns*. John Wiley & Sons, 2013.

[11] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[12] R. Kazman, Y. Cai, R. Mo, Q. Feng, L. Xiao, S. Haziyev, V. Fedak, and A. Shapochka. A case study in locating the architectural roots of technical debt. In *Proc. 37th International Conference on Software Engineering*, May 2015.

[13] Y. C. Martin Naedele, Rick Kazman. Making the case for a manufacturing execution system for software development. *Commun. ACM*, 57(12):33–36, 2014.

[14] R. Mo, Y. Cai, R. Kazman, and L. Xiao. Hotspot patterns: The formal definition and automatic detection of architecture smells. In *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*, pages 51–60. IEEE, 2015.

[15] R. Mo, Y. Cai, R. Kazman, L. Xiao, and Q. Feng. Decoupling level: A new metric for architectural maintenance complexity. In *Proc. 38th International Conference on Software Engineering*. IEEE, 2016.

[16] M. Naedele, H.-M. Chen, R. Kazman, Y. Cai, L. Xiao, and C. V. Silva. Manufacturing execution systems: A vision for managing software development. *Journal of Systems and Software*, 101:59–68, 2015.

[17] J. Ryoo, R. Kazman, and P. Anand. Architectural analysis of security vulnerabilities. *IEEE Security & Privacy*, 13(6):52–59, 2015.

[18] R. Schwanke, L. Xiao, and Y. Cai. Measuring architecture quality by structure plus history analysis. In *Proc. 35rd International Conference on Software Engineering*, pages 891–900, May 2013.

[19] M. Sutton, A. Greene, and P. Amini. *Fuzzing: brute force vulnerability discovery*. Pearson Education, 2007.

[20] S. Wong, Y. Cai, G. Valetto, G. Simeonov, and K. Sethi. Design rule hierarchies and parallelism in software development tasks. In *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, pages 197–208, Nov. 2009.

[21] L. Xiao, Y. Cai, and R. Kazman. Design rule spaces: A new form of architecture insight. In *Proc. 36rd International Conference on Software Engineering*, 2014.