Software Engineering Institute

Results of SEI Line-Funded Exploratory New Starts Projects

Len Bass, Nanette Brown, Gene Cahill, William Casey, Sagar Chaki, Cory Cohen, Dionisio de Niz,
David French, Arie Gurfinkel, Rick Kazman, Ed Morris, Brad Myers, William Nichols, Robert L. Nord,
Ipek Ozkaya, Raghvinder S. Sangwan, Soumya Simanta, Ofer Strichman, Peppo Valetto

August 2012

TECHNICAL REPORT
CMU/SEI-2012-TR-004
ESC-TR-2012-004

http://www.sei.cmu.edu

Carnegie Mellon

This material is based upon work funded and supported by the United States 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.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Department of Defense.

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

NO WARRANTY

THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN “AS-IS” BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[Include the following applicable trademark language—needed only if specific SEI trademarks are used in this report. If not needed, delete the following table or elements of it.]

* These restrictions do not apply to U.S. government entities.

Table of Contents

Abstract **v**

1 Introduction **1**
- 1.1 Purpose of the SEI Line-Funded Exploratory New Starts 1
- 1.2 Overview of LENS Projects 1

2 Fuzzy Hashing Techniques in Applied Malware Analysis **2**
- 2.1 Purpose 2
- 2.2 Background 3
- 2.3 Approach 5
- 2.4 Collaborations 6
- 2.5 Evaluation Criteria 6
- 2.6 Results 6
  - 2.6.1 Fuzzy Hashing at Large Scale 7
  - 2.6.2 Why Fuzzy Hashing Works 9
  - 2.6.3 Alternate Fuzzy Hashing Techniques 16
- 2.7 Publications and Presentations 16
- 2.8 Bibliography 16

3 Safe Resource Optimization of Mixed-Criticality Cyber-Physical Systems **19**
- 3.1 Purpose 19
- 3.2 Background 20
- 3.3 Approach 20
- 3.4 Collaborations 22
- 3.5 Evaluation Criteria 22
- 3.6 Results 23
- 3.7 References 24

4 Measuring the Impact of Explicit Architecture Descriptions **26**
- 4.1 Purpose 26
- 4.2 Background 26
- 4.3 Approach 26
  - 4.3.1 Gaining the Overview 27
  - 4.3.2 Expert Interview 28
  - 4.3.3 Directory Structure 29
  - 4.3.4 Tool Support 29
  - 4.3.5 Validation 31
  - 4.3.6 Publication 31
- 4.4 Collaborations 31
- 4.5 Evaluation Criteria 31
- 4.6 Results 31
  - 4.6.1 Repository Data Collection 32
  - 4.6.2 Analyses 33
  - 4.6.3 Result Conclusions 34
- 4.7 Publications and Presentations 35
- 4.8 References 35

5 Regression Verification of Embedded Software **36**
- 5.1 Purpose 36
- 5.2 Background 38
- 5.3 Approach 39

5.4 Collaborations 40
5.5 Evaluation Criteria 40
5.6 Results 40
5.7 Bibliography/References 41

6 Sparse Representation Modeling for Software Corpora 43

6.1 Purpose: Accurate, Reliable, and Efficient Malware Discovery 43
6.2 Approach 45
6.2.1 Query an Artifact Against a Malware Reference Data Set 47
6.2.2 Search and Retrieval of Long Common Substrings to Investigate Alleged Code Sharing 48
6.2.3 LCS Application to Malware Family Clustering 51
6.3 Challenges, Open Problems and Future Work 54
6.4 Collaborations 57
6.5 References 57

7 Learning a Portfolio-Based Checker for Provenance-Similarity of Binaries 61

7.1 Purpose 61
7.2 Background 64
7.3 Approach 65
7.4 Collaborations 68
7.5 Evaluation Criteria 68
7.6 Results 68
7.7 Bibliography/References 70

8 Edge-Enabled Tactical Systems: Edge Mission-Oriented Tactical App Generator 72

8.1 Purpose 72
8.2 Background 72
8.3 Approach 73
8.4 Collaboration 78
8.5 Evaluation Criterion 78
8.6 Results 79

9 Communicating the Benefits of Architecting within Agile Development: Quantifying the Value of Architecting within Agile Software Development via Technical Debt Analysis 80

9.1 Purpose 80
9.2 Background 80
9.3 Approach 82
9.4 Collaborations 82
9.5 Evaluation Criteria 83
9.6 Results 83
9.7 Publications and Presentations 90
9.8 References 90

List of Figures

Figure 1: ssdeep v2.6 True Positives vs. False Negatives 11

Figure 2: ssdeep-dc3 v2.5 True Positives vs. False Negatives 12

Figure 3: sdhash v0.3 True Positives vs. False Negatives 12

Figure 4: ssdeep v2.6 True Negatives vs. False Positives 13

Figure 5: ssdeep-dc3 v2.5 True Negatives vs. False Positives 13

Figure 6: sdhash v0.3 True Negatives vs. False Positives 14

Figure 7: ssdeep vs. sdhashPrecision 15

Figure 8: ssdeep vs. sdhashRecall 15

Figure 9: HDFS Architecture Diagram From Hadoop Website 27

Figure 10: Elicitation of Architectural Information 29

Figure 11: Module Relationships in HDFS 30

Figure 12: Jira Issue Turnaround Time Statistics. 34

Figure 13: Plot of Matches Against Aliser Corpus 48

Figure 14: A Visualization of File Matches 51

Figure 15: Poison Ivy Vectorization Based on LCS Features 53

Figure 16: Dendogram Using Hierarchical Clustering 54

Figure 17: Overview of a Portfolio-Based Checker for Provenance Similarity 67

Figure 18. Comparison of Effective Classifiers 69

Figure 19. Performance of RandomForest with Varying Number of Trees 69

Figure 20. Comparison of Semantic and Syntactic Attributes 70

Figure 21: Creating Basic Types 75

Figure 22: Adding Fields to a Type 75

Figure 23: A Type (person) With Fields Added 76

Figure 24: Input Form for Person 76

Figure 25: Information About a Specific Person 77

Figure 26: Information Displayed on a Map 77

Figure 27: Technical Debt [Fowler 2009] 81

Figure 28: Value of Capabilities Delivered Over Total Effort for What-If Development Paths 88

List of Tables

Table 1: File Size/Section Collision Breakdown for Artifact Catalog 8

Table 2: Sample Task Set 23

Table 3: Allocation of Stories to Release in Each Path 87

Table 4: Comparison of the Costs of the Three What-If Development Paths 88

Abstract

The Software Engineering Institute (SEI) annually undertakes several line-funded exploratory new starts (LENS) projects. These projects serve to (1) support feasibility studies investigating whether further work by the SEI would be of potential benefit and (2) support further exploratory work to determine whether there is sufficient value in eventually funding the feasibility study work as an SEI initiative. Projects are chosen based on their potential to mature and/or transition software engineering practices, develop information that will help in deciding whether further work is worth funding, and set new directions for SEI work. This report describes the LENS projects that were conducted during fiscal year 2011 (October 2010 through September 2011).

1 Introduction

1.1 Purpose of the SEI Line-Funded Exploratory New Starts

Software Engineering Institute (SEI) line-funded exploratory new starts (LENS) funds are used in two ways: (1) to support feasibility studies investigating whether further work by the SEI would be of potential benefit and (2) to support further exploratory work to determine whether there is sufficient value in eventually funding the feasibility study work as an SEI initiative. It is anticipated that each year there will be three or four feasibility studies and that one or two of these studies will be further funded to lay the foundation for the work possibly becoming an initiative.

Feasibility studies are evaluated against the following criteria:

- mission criticality: To what extent is there a potentially dramatic increase in maturing and/or transitioning software engineering practices if work on the proposed topic yields positive results? What will the impact be on the Department of Defense (DoD)?
- sufficiency of study results: To what extent will information developed by the study help in deciding whether further work is worth funding?
- new directions: To what extent does the work set new directions as contrasted with building on current work? Ideally, the SEI seeks a mix of studies that build on current work and studies that set new directions.

1.2 Overview of LENS Projects

The following research projects were undertaken in FY 2011:

- Sparse Representation Modeling of Software Corpora (William Casey, Team Lead)
- Regression Verification of Embedded Software (Sagar Chaki, Team Lead)
- Learning a Portfolio-Based Checker for Provenance-Similarity of Binaries (Sagar Chaki, Team Lead)
- Fuzzy Hashing Techniques in Applied Malware Analysis (David French, Team Lead)
- Safe Resource Optimization of Mixed-Criticality Cyber-Physical Systems (Dionisio de Niz, Team Lead)
- Communicating the Benefits of Architecting within Agile Development – Year 2 (Ipek Ozkaya, Team Lead)
- Edge-Enabled Tactical Systems (Ed Morris, Team Lead)
- Measuring the Impact of Explicit Architecture Descriptions (Rick Kazman, Team Lead)

These projects are summarized in this technical report.

2 Fuzzy Hashing Techniques in Applied Malware Analysis

David French, William Casey

2.1 Purpose

The purpose of this effort is to obtain greater insights into the practice of fuzzy hashing, as applied to malicious software similarity detection. Fuzzy hashing is relevant to malicious code analysis in that, for a given pair of input files, a percentage metric, or score, can be generated which attempts to indicate the relative percentage of content the two input files have in common. Using this score, one can immediately ascertain, with some degree of confidence, the extent to which two files are similar to each other. Making this assessment has applications in digital forensics (where digital artifacts must be located, and derivations thereof noted), computer security incident response (in which responders must locate all copies or near-copies of a particular malicious executable on several systems), and malware triage (where a file suspected of being malicious must be quickly analyzed and determined to be “clean” or worthy of further analysis). However, there are several problems with using fuzzy hashing in these endeavors.

First is the problem of specificity. Since a fuzzy hash represents a substantial information loss from the original input data, it is unclear what any particular similarity score signifies. Traditional cryptographic hashes (such as MD5[1]) are used to ascertain identity; in this usage, hash collision is significant. In contrast, fuzzy hashes are typically compared to generate a normalized score between 0 and 100 inclusive, which are intuitively interpreted to mean “completely different” and “completely identical” respectively. Any score in between must be subjectively judged, and this subjective analysis can lead to very different interpretations between analysts. For a common reporting infrastructure, the possibility of multiple well-meaning analysts to draw different conclusions based on the same data is inherently undesirable.

The second problem is the scale at which these comparisons can be made. Comparing two hashes for a single person represents a trivial cost; comparing a single hash against a known database of tens of millions of files (such as the NIST National Software Reference Library[2]) is a quadratic operation, especially when no information informs the selection of data against which to compare (such as might lead to exploitation of triangle inequality). Thus, when applying fuzzy hashes to malicious code, we apply an expensive operation to an unknown file for dubiously valuable results.

Finally, the third problem with fuzzy hashing of malicious code is the relative ease with which it is thwarted. Malware has properties that make it unlike many other types of information against which similarity metrics may be applied. Its structure is somewhat loosely defined by a formal specification.[3] The semantic information contained in an executable must be made available to a

[1] Message Digest algorithm 5, a 128-bit cryptographic hash, see IETF RFC 1321

[2] A reference data set for use by government, industry, & law enforcement, see http://www.nsrl.nist.gov

[3] The Portable Executable/Common Object File Format, or PE/COFF, see http://www.microsoft.com/whdc/system/platform/firmware/pecoff.mspx

computer processor upon execution, but it may be ambiguously presented *in situ*, and may be transformed, obfuscated, or omitted as needed. Any sequence of instructions that result in the same output state may be executed, without regard for brevity, clarity, or even correctness. Further, any amount of semi-formatted information may be present or absent in an executable, from extra or misleading sections, to restructuring into a single limited section with unaddressed slack containing the executable code. Contrast this with, for example, English language documents that are suspected of being plagiarized; in order to communicate information, the document must unambiguously deliver semantic content to the reader, and the degree to which a single set of statements may be modified while preserving semantic content and comprehensibility is limited. We would like for fuzzy hashing techniques to give us the same sense of derivation that plagiarism detection or generative text detection techniques do. Unfortunately they may do so only under rigid circumstances which may not be obviously detectable. Thus, while fuzzy hashing has been introduced into the domain of malware analysis, it has not yet been adapted for use with malware analysis, and it is this adaptation that we desire to support.

This situation creates an opportunity for us to positively affect the de-facto concept of operations (CONOPS) for malicious code analysis and triage. We are uniquely positioned to make a comprehensive evaluation of fuzzy hashing, specifically using ssdeep, as it applies to millions of malicious files in the CERT Artifact Catalog. We do not propose to simply generate ssdeep hashes for these files, publish the list, and be done with it. Rather, we hope to use known malicious files to systematically exercise the ssdeep hash generation and comparison algorithms, for the purpose of revealing and documenting the expected behavior of the algorithms in as many real-world scenarios as we have available to us. Since several government organizations are actively using ssdeep in operational systems,[4] we feel it is imperative for us to act.

2.2 Background

Assessing binary similarity of malicious code is essentially a triage function. Given a new executable introduced into a system, the ability to rapidly assess whether the new file is substantially similar to an existing file is crucial to making resource expenditure decisions. If a file is similar (but not identical) to a known non-malicious file, this may be an indicator of file infection, or it may indicate a new revision of the non-malicious file (and thus the provenance of the file may inform the decision to analyze the file further). If a file is similar to a known malicious file, then one may assume the file is also malicious, and can quarantine or otherwise respond immediately. Thus, rapidly and confidently comparing one new file to a large quantity of known files has enormous utility in network defense.

Since a similarity function is of such high value, it is essential to understand the properties of the similarity function, in order to derive expectations from known parameters. Understanding the precision[5]and recall[6]of a similarity function allows such a function to perform in circumstances

[4] Primarily as a triage function

[5] A measure of the number of relevant matches returned by a search divided by the total number of matches that were actually returned

[6] A measure of the number of relevant matches returned by a search divided by the total number of matches that should have been returned

where it is expensive or impossible for a human to perform the same comparison, e.g., in an automated system, or at sufficiently large scale. Without this understanding, application of a similarity function is naïve at best, dangerous at worst, for it acts essentially as a "black box", into which an operator may not peer. However, this understanding should encompass not only the expected response of the function, but also deep understanding of the data itself. In this case, the data represents potentially malicious executables, which are the product of an intelligent adversary who benefits directly[7]by thwarting both superficial and deep analysis of the executable itself.

The original concept behind the ssdeep fuzzy hashing algorithm was created by Dr. Andrew Tridgell, and was applied mainly to the detection of generative spam (that is, spam messages which represent variations on a central core of information) [Tridgell, 2002]. This application is called *spamsum*, and provides two distinct capabilities; a *hashing* algorithm, which uses a rolling hash[8] to produce data points for a stronger non-cryptographic hash (similar to FNV[9]), and a *comparison* algorithm for comparing two hashes, which is implemented in terms of Levenshtein[10]distance. By utilizing the spamsum program, one can take two input text sequences and produce a metric for how similar the sequences are to each other.

The ssdeep program represents a direct application of the hashing and comparison algorithms from spamsum to binary sequences. ssdeep is distributed as open source software, and so one can observe (by examination of the source code for both ssdeep and spamsum) that the major functions from spamsum were copied directly into ssdeep without modification [Kornblum 2010]. The motivation for doing so was for forensics applications, including altered document matching, and partial file matching (such as might occur during data carving[11] in digital forensics procedures) [Kornblum, 2006]. Specific application to malware was made later, and seemed focused on using ssdeep to point a human investigator in the right direction during their investigation [Mandiant 2006]. Using ssdeep in this way allows a human to be the final arbiter of whether similarity measures have worked, or worked correctly, and can be a productive tool in performing their jobs.

Using ssdeep in an automated fashion seems like a natural derivation from single-file human-directed activities. In this light, fuzzy hash generation may be viewed as simply generating additional metadata for a human to use in his or her duties (network defense assessment, incident handling, legal investigation, etc). Several prominent organizations and figures have advocated doing just that [Zeltser, 2010, Hispasec 2010]. Even NIST has begun generating fuzzy hashes for the entire National Software Reference Library [NIST 2010]. Government organizations routinely include ssdeep hashes in their malware reporting. It is our experience that individual anecdotes have informed the decision to use ssdeep widely, and that no systematic treatment of the algorithms have been applied to malicious code.

[7] If a target of attack cannot reliably understand or analyze the attack, they become susceptible to future identical attacks

[8] A hash function where the input is hashed in a window that moves through the input

[9] Fowler-Noll-Vo algorithm, which uses multiples of primes and XORs to hash the input data

[10] Commonly referred to as "edit distance", tracks the minimum changes to mutate one string into another

[11] The practice of extracting fragments of data from a larger stream, based on partial structure identification

Fuzzy hashes are often listed in the same metadata section as other strong cryptographic hashes such as MD5 or SHA.[12] These hashes share the property that hash collision is determined to be highly significant,[13]and thus collision is used to assert that two input data sources that hash to the same value must in fact be identical. A reasonable yet uninformed person might conclude that fuzzy hashing, appearing in the same context as strong cryptographic hashes, shares the same kind of weight when collision occurs, and we have developed anecdotal evidence that this is true. However, fuzzy hash comparisons exist in a Euclidean space; two hashes are between zero and one hundred percent identical. Thus, the significance of the result of comparing two fuzzy hashes has room for interpretation; what does it say when two hashes from two data sources are 99 percent similar? 70 percent similar? 5 percent similar? It is these questions that we propose to answer, when applying fuzzy hashes specifically to malicious code.

Treatment of fuzzy hashes as applied to malware in the literature is slight; while dozens of relevant papers have been written on rolling hashes and their applications , as well as applying these kinds of hashes to digital media, the concept of using these hashes in security-based scenarios is relatively new[Cohen 1997, Karp 1987, Brin 1995]. Typical applications include digital forensics, and are oriented toward document matching and file fragment identification [Aquilina 2008, Hurlbut, 2009]. YAP3 is an application of traditional similarity metrics to computer software, but is limited to the source code (and other textual information) and is geared toward detecting plagiarism [Wise 1996]. Specifically applying ssdeep to malware has been treated before; however, the treatment was limited to several thousand known malicious files comprising four families [DigitalNinja 2007]. Given that we have access to several thousand malware families from the Artifact Catalog (which can contain several hundred thousand files each), we are poised to develop much more comprehensive understanding based on real malware. Some of the limitations of ssdeep have been more closely examined, and their treatment of hash selection in general and ssdeep in particular is sound [Roussev 2007]. However, ssdeep has already gained currency in malware analysis, so simply pointing out the limitations and proposing alternatives is not practical; we need to understand the failure modes, and determine if there are alternate analysis opportunities utilizing the same data that can maximize precision and recall.

2.3 Approach

We approached this research effort by first assembling our data set. At the time this work was proposed, the CERT Artifact Catalog contained approximately ten million files, which grew to approximately twenty-four million files by the conclusion of this effort. Of these many million files, we separated them into three gross categories: *executables*, *documents* (including Office documents, PDF, etc), and *other*. For the purposes of this effort, we concentrated primarily on executables. Within the class of executables, we noted exploitable characteristics, which we used to classify these executables into homogenous groups. These classes include: *section identity* (wherein files are comprised of the exact same sections, when headers and slack data is ignored); *section sharing*, where files share one or more sections but differ in others; *function identity* (wherein files are comprised of the exact same functions, even when their sections differ); *func-*

[12] Secure Hash Algorithm

[13] These algorithms are designed to minimize accidental collision

tion sharing (where files share one or more functions but differ in others); *unknown* (wherein we have no structural information as to whether files are similar or not).

Once we assembled our data, we considered generating ssdeep hashes for all files in these classes. Since we had *a priori* knowledge of the extent to which files are similar within each of these classes (by rigid metrics obtained by cryptographic hash collisions), we attempted to predict the cost of fuzzy hash comparison. We decided to analyze the files in our corpus for the characteristics used by ssdeep to compute fuzzy hashing, and see whether we could make rational decisions about which files to hash and compare based on the correlation between these statistics and other statistics (such as section hash collisions). Subsequently, in an attempt to understand why fuzzy hashing works at all, we decomposed and analyzed the Portable Executable format (in which we have established clear expertise over the years), and using this knowledge, posited several models for why two different executable files (where difference is measured by the cryptographic hashes computed over the entire file contents) might have some amount of content in common. We used our data set to confirm our suspicions, both with statistics and human analysis, and then corroborated our findings using fuzzy hashing.

We then analyzed the actual ssdeep hashing and comparison algorithms, to determine whether the assumptions exigent in the original spamsum algorithm apply to executable comparisons. We concentrated primarily on continuity of the hashing function, and explored alternatives.

2.4 Collaborations

David French and William Casey, both members of the Malicious Code component of CERT, were the primary SEI researchers involved in this project.

Jesse Kornblum (author of ssdeep, currently working with Kyrus Technology) graciously agreed to collaborate on this effort. Discussions with Kornblum were the primary motivator to consider alternate algorithms, including sdhash, and his contributions have been invaluable.

2.5 Evaluation Criteria

For each of the fuzzy hashing areas selected, different evaluation criteria were used. In assessing performance at scale, we used alternate human-vetted and deployed techniques to provide contrast to how fuzzy hashing performed. In assessing precision and recall, we used human analysis to identify known files, and compared the results produced by fuzzy hashing tools against human performance. Success criteria for the entire project were transition to operational usage, and communication with a broad audience, both of which have been achieved as part of this effort.

2.6 Results

In all, there were three focus areas for this research:

- Understanding ssdeep behavior in large-scale corpora of malware
- Understanding why fuzzy hashing works at all
- Alternate fuzzy hashing techniques

These are primarily operational considerations, and are intended to uncover and explore how fuzzy hashing may actually be applied to real malware analysis, rather than as a theoretical algorithm against controlled data. As such, we cannot make generalized statements about the efficacy of fuzzy hashing against all malware as a class; however, we have tried to select classes of malware that enjoy common usage, or that are part of real current security incidents, in order to make a more meaningful if less general statement about effectiveness.

2.6.1 Fuzzy Hashing at Large Scale

Fuzzy hashing at scale poses two fundamental challenges to operational usage: timeliness of results, and usefulness of results. In order for fuzzy hashing matches to be assessed against a particular file, they must be made available within a small amount of time (relative to whatever time drivers a security incident presents). Additionally, any files matching a fuzzy hash over some threshold must have meaningful relation to the file being compared. To understand what expectations we might hold when comparing a single file against potentially millions of files, we analyzed the Artifact Catalog in order to understand two things: 1) given the format and size of a particular file, how many files might we reasonable expect to comprise the search space for the file? and 2) given the format and size of a particular file, what might we expect the fuzzy hash comparison results to be, given other things we know about these types of files?

We specifically considered Portable Executable (or PE) files as the target for this investigation, and divided up the Artifact Catalog into several sets of files. We applied section hashing (described in detail in the 2010 CERT Annual Research Report article *Beyond Section Hashing*) to the PE files in the Artifact Catalog (approximately 10.7 million files as of the time of the experiment). We divided PE files into three categories: files that shared all their sections (which we refer to as the *identical* category); files that share one or more (but not all) sections (which we refer to as the *colliding* category); files that had no sections in common with any other file (which we refer to as the *singleton* category). Files that share section data should also produce a non-zero fuzzy hash comparison, provided the colliding sections comprise a relatively large proportion of the total bytes of the files that contain them. Thus, categorizing PE files in this way gives us a sense of not only what we should expect of fuzzy hash comparisons (in terms of relative number of files producing non-zero comparisons), but also which files are most productively compared (since comparing fuzzy hashes for files we already know share section data is at best redundant).

We computed the ssdeep block size for each file in each category, assigned these files into bins based on the block and file sizes, and determined the percentage of files within each category (identical, colliding, singleton) that populated each bin. Each bin is referred to by the minimum number of bytes required for a file to fall into the bin (for example, bin 6144 contains all files with sizes 6,144 bytes to 12,287 bytes inclusive). Since ssdeep produces two hashes (one at the file's block size, and one at double the file's block size), it is capable of comparing any two files that are within one block size of each other. Thus, a maximum of three block size bins will be searched for any given file's fuzzy hash. By observing the size and section character of malware (and accounting for the fact that the Artifact Catalog likely has sample bias and may not be representative of all malware), we can with some confidence describe the files that are most fruitful compared when using fuzzy hashes at scale. Table 1 presents the size/section breakdown for 10.7 million PE files from the Artifact Catalog.

Table 1: File Size/Section Collision Breakdown for Artifact Catalog

File size bin	Singleton	Colliding	Identical	Total hashable	% with collisions	% no collisions
192	7	0	0	7	0.00%	100.00%
384	176	15	66	257	31.52%	68.48%
768	868	818	324	2010	56.82%	43.18%
1536	3584	24746	2002	30332	88.18%	11.82%
3072	9979	43961	9069	63009	84.16%	15.84%
6144	50064	147488	35433	232985	78.51%	21.49%
12288	87359	318993	288182	694534	87.42%	12.58%
24576	**138269**	**685512**	**252091**	**1075872**	**87.15%**	**12.85%**
49152	**433210**	**963848**	**460945**	**1858003**	**76.68%**	**23.32%**
98304	**221043**	**899679**	**1069170**	**2189892**	**89.91%**	**10.09%**
196608	**308380**	**821387**	**656008**	**1785775**	**82.73%**	**17.27%**
393216	**311598**	**611459**	**524396**	**1447453**	**78.47%**	**21.53%**
786432	73727	285747	451415	810889	90.91%	9.09%
1572864	37208	185950	154244	377402	90.14%	9.86%
3145728	15461	74439	60315	150215	89.71%	10.29%
6291456	2497	23266	26318	52081	95.21%	4.79%
12582912	181	859	479	1519	88.08%	11.92%
25165824	33	200	110	343	90.38%	9.62%
50331648	9	63	58	130	93.08%	6.92%
100663296	0	7	7	14	100.00%	0.00%
201326592	0	4	1	5	100.00%	0.00%
402653184	0	3	0	3	100.00%	0.00%
Total	**1693653**	**5088444**	**3990633**	**10772730**	**84.28%**	**15.72%**

As we can see from this table, 8,356,995 of 10,772,730 files (comprising 77.57 percent of all files for which we can compute section hashes) fall into file size bins 24576, 49152, 98304, 196608, and 393216. These bins are all adjacent to each other, and a file populating any of these bins will be compared against 33.6 percent to 54 percent of all files in the Artifact Catalog (percentages obtained by summing the number of files in adjacent bins for each bin). Since the ssdeep comparison algorithm is O(m*n) (where m is the size of the fuzzy hash and n is the number of files being compared against), this represents a very large and relatively expensive search space. However, 83.1 percent of these files (or 6,944,459 files) share at least one section in common with another file. Consequently, searching these files using fuzzy hashing of any kind will produce answers redundant with section hashing. This leaves 16.9 percent of these files (or 1,412,536 files) to compare fuzzy hashes in the five most populous file size bins, which is a dramatic reduction in the search space. The semantics of section hashing are well-understood for many types of sections; if the .text section is in common with another file and the .text sections contain the entry points of the programs, then the programs share code in common. In contrast, non-zero fuzzy hash comparisons must be interpreted to discover the cause, which is a non-trivial proposition for large numbers of files. Limiting the number of files that must be interpreted, by pre-filtering searches using section hashes, is thus an effective way to accomplish the goal of finding possibly related files, while minimizing the amount of interpretation that must be applied to files matching fuzzy hash comparisons.

2.6.2 Why Fuzzy Hashing Works

In order to understand why fuzzy hashing works at all, we again return to the PE format to help guide our understanding. Malware is software combining three elements: 1) code, whether compiled source code written in a high-level language or hand-crafted assembly, 2) data, which is some set of numerical, textual, or other types of discrete values intended to drive the logic of the code in specific ways, and 3) process, which is loosely a set of operations (for example, compiling and linking) applied to the code and data that ultimately produce an executable sequence of bytes in a particular format, subject to specific operating constraints. Given a distinct set of code, data, and consistent processes applied thereto, it is reasonable to conclude that—barring changes to any of these—we will produce an identical executable file every time we apply the process to the code and data (where identity is measured using a cryptographic hash, such as MD5). We now consider how the permutation of any of these components will affect the resulting executable file.

First, let us consider the effect of modifying the data used to drive a particular executable. With respect to malicious software, such data may include remote access information (such as IP addresses, hostnames, usernames and passwords, commands, and the like), installation and configuration information (such as registry keys, temporary filenames, mutexes, etc.), or any other values which cause the malware to execute in specific ways. Generally speaking, changing the values of these data may cause different behavior in the malware at runtime but should have little impact on the structure of the malware.

Malware authors may modify their source code to use different data values for each new program instance or may construct their program to access these data values outside the context of the compiled program (for example, by embedding the data within or at the end of the PE file). In the case of malicious code, data may also include bytes whose presence does not alter the behavior of the code in any way, and whose purpose is to confuse analysis. Regardless, the expected changes to the resulting executable file are directly proportional to the amount of data changed. Since we only changed the values of data—not the way in they are referenced (in particular, we have not changed the code)—we can expect that the structure of the output file is modified only to support any different storage requirements for the new data.

Similarly, let us consider the effect of modifying the code found in a particular executable. The code defines the essential logic of the malware and describes the behavior of the program under specified conditions. To modify program behavior, the code must generally be modified. The expected changes to the resulting executable file are proportional to the amount of code changed, much as we expect when changing data. However, code—especially compiled code—differs from data in that the representation of the code in its final form is often drastically different from its original form. Compiling and linking source code represents a semantic transformation, with the resulting product intended for consumption by a processor, not a human reader.

To accomplish semantic transformation most effectively, the compiler and linker may perform all manner of permutations, such as rewriting blocks of code to execute more efficiently, reordering code in memory to take up less space, and even removing code that is not referenced within the original source. If we assume that the process to create the executable remains constant (for example, that optimization settings are not changed between compilations), we must still allow that minor changes in the original source code may have unpredictably large changes in the resulting executable. As a consequence, code changes are more likely to produce executables with larger structural differences between revisions than executables where only data changes.

Thus, we have described two general cases in which structurally different files (measured by cryptographic hashing, such as MD5) may be produced from a common source. We refer to malware families whose primary or sole permutation is in their data as generative malware, and use the analogy of a malware factory cranking out different MD5s by modifying data bytes in some way. We refer to malware families whose primary permutation is in their code as evolutionary malware, in that the behavior of the program evolves over time. When considering the effects of similarity measurements such as fuzzy hashing, we may expect that fuzzy hashing will perform differently against these different general types of malware.

In attempting to discover how fuzzy hashing works against different types of malware, we assembled 1,500 files comprising 13 different known malware families, and a number of packers. We tried to use malware families whose behavior and relationships were well understood, and for which a sufficient number of exemplars existed to describe how the families varied. We used three different fuzzy hashing techniques to compare these files: ssdeep v2.6,[14] ssdeep-dc3 v2.5,[15] and sdhash v0.3.[16]For each of these fuzzy hashing tools, we performed all-pairs comparison of these known malware files, and compute the true and false positive and negative rates. For ssdeep and sdhash, we also computed the precision and recall of these tools, based on the true/false positive/negative rates. Each of these files was labeled by a human as belonging to a particular family.

True positive refers to the case where a human labeled two files as belonging to the same family, and the fuzzy hash comparison was non-zero between those two files (indicating some form of relatedness).

False positive refers to the case where a human labeled two files as belonging to two different families, and the fuzzy hash comparison was non-zero between those two files (indicating some form of relatedness).

True negative refers to the case where a human labeled two files as belonging to two different families, and the fuzzy hash comparison was zero between those two files (indicating no relationship).

False negative refers to the case where a human labeled two files as belonging to the same family, and the fuzzy hash comparison was zero between those two files (indicating no relationship).

The rates are determined by comparing each file of each family to every other file, and counting the number of expected results for each family. Since there are 1,500 files total, for each family of size N, the true positive rate is determined by counting the number of assigned positives and dividing by N. Likewise, for each family of size N, the true negative rate is determined by counting the number of assigned negatives and dividing by (1,500-N). The false negative rate is determined

[14] Obtained from http://ssdeep.sourceforge.net/

[15] Obtained from http://ssdeep.svn.sourceforge.net/viewvc/ssdeep/branches/dc3/

[16] Obtained from http://roussev.net/sdhash/

by subtracting the actual true positive rate from the expected true positive rate of 100 percent. The false positive rate is determined by subtracting the actual true negative rate from the expected true negative rate of 100 percent.

The following figures describe the experimental results. We first treat the true positive vs. false negative rates, which give us a sense of whether these fuzzy hashing methods are capable of identifying files that are known to be related. True positive rates are displayed in green, and false negative rates are displayed in red.

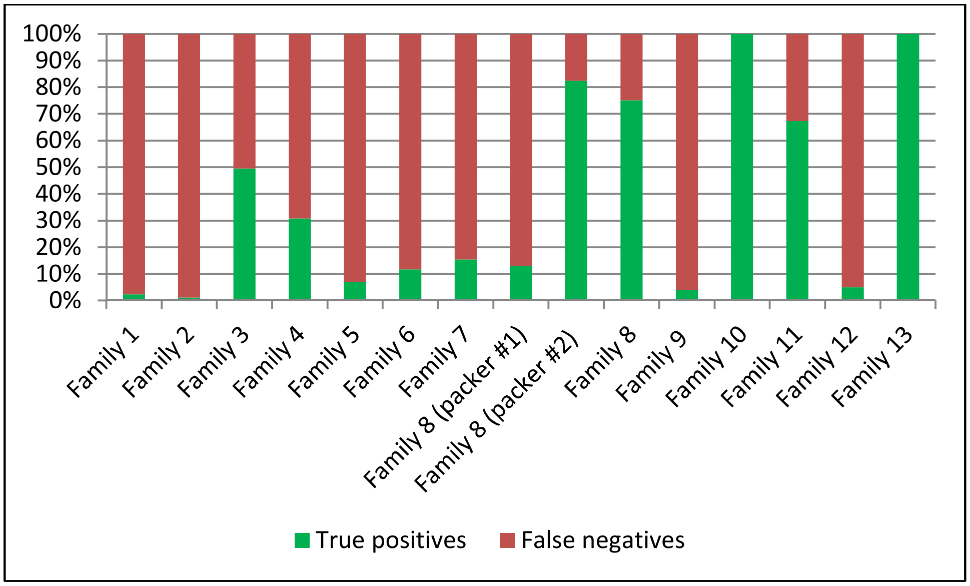


Figure 1: ssdeep v2.6 True Positives vs. False Negatives

Figure 1 shows the true positive vs. false negative rates for ssdeep v2.6. As we can see, ssdeep's performance against different families fluctuates quite a bit, from perfect association for some families, and almost negligible association for others.

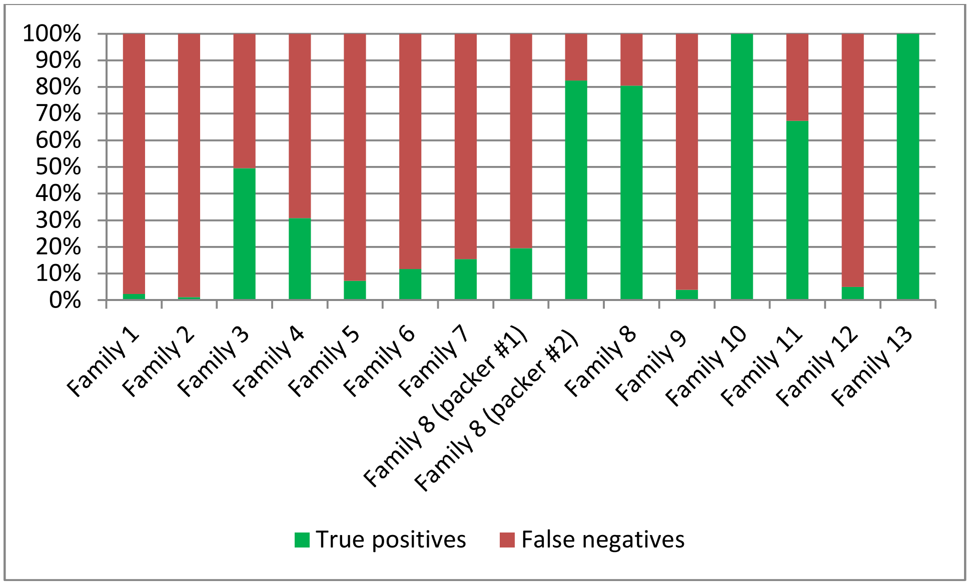


Figure 2: ssdeep-dc3 v2.5 True Positives vs. False Negatives

Figure 2 shows the true positive vs. false negative rates for ssdeep-dc3 v2.5. As we can see, ssdeep-dc3's performance against different families also fluctuates quite a bit, from perfect association for some families, and almost negligible association for others. Since ssdeep-dc3 is using three computed hashes instead of two, it has the opportunity to make additional comparisons for files that are related but whose sizes vary by more than one block size. Even with these additional computations, the overall performance of ssdeep-dc3 is only marginally better than ssdeep's.

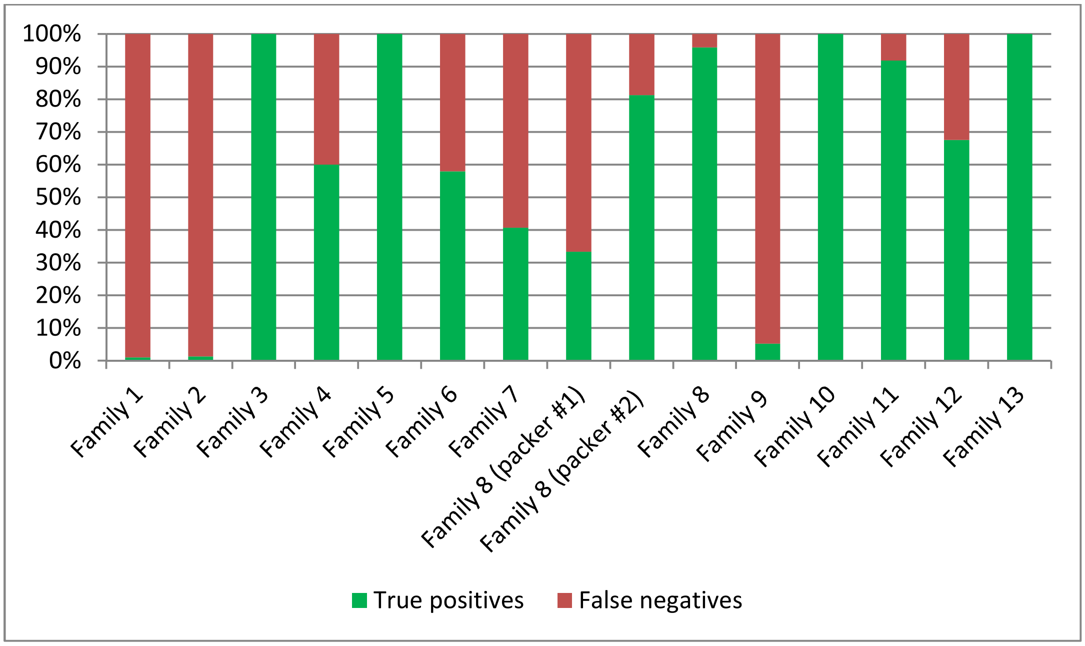


Figure 3: sdhash v0.3 True Positives vs. False Negatives

Figure 3 shows the true positive vs. false negative rates for sdhash v0.3. Sdhash performs considerably better than either ssdeep version at identifying files that are actually related, even in the presence of packing (as evidenced by family 8). However, there are still types of malware (namely the file infectors) against which fuzzy hashing does not seem effective at all.

The next set of figures examines the false positive/true negative rates for each of these fuzzy hashing techniques.

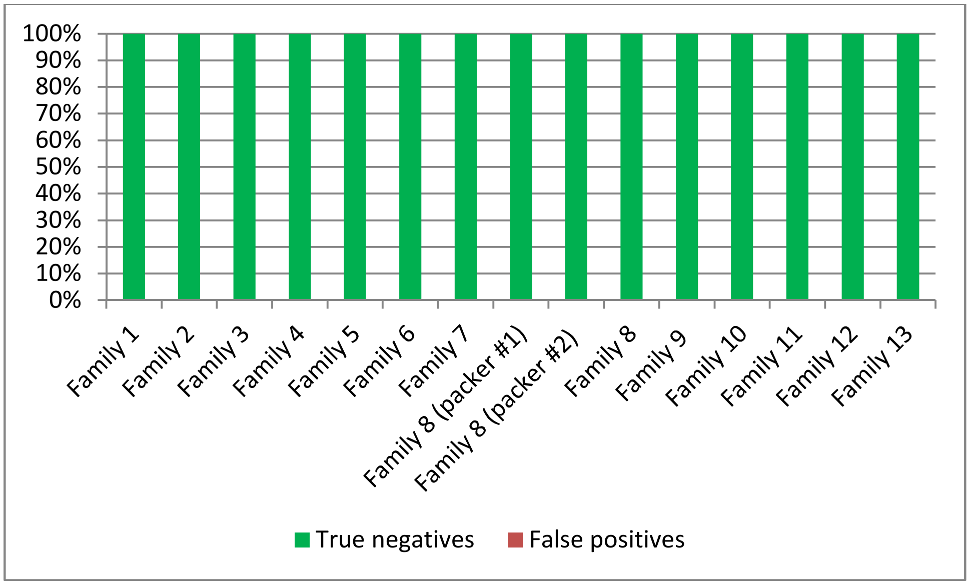


Figure 4: ssdeep v2.6 True Negatives vs. False Positives

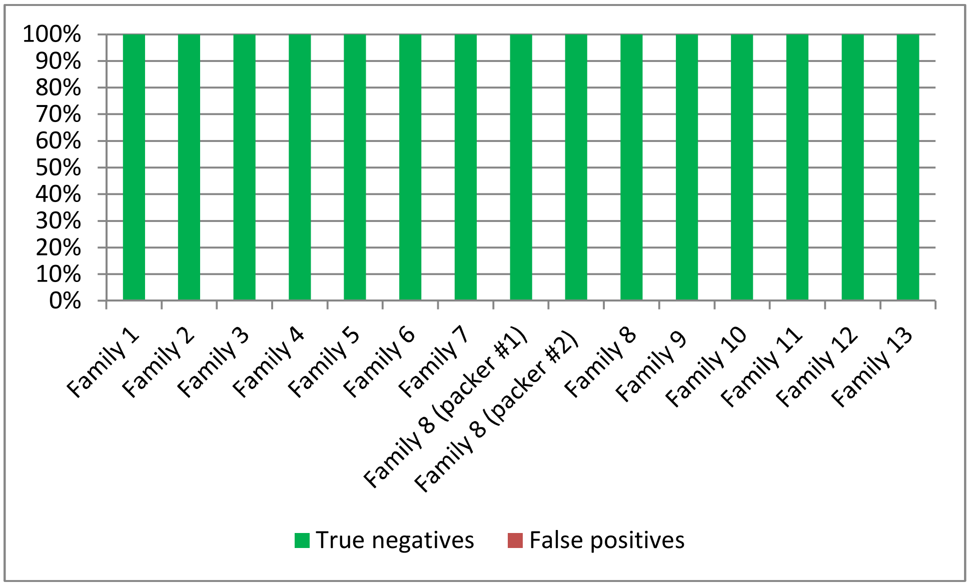


Figure 5: ssdeep-dc3 v2.5 True Negatives vs. False Positives

We can see from Figures 4 and 5 that the true negative rates for ssdeep and ssdeep-dc3 are 100 percent, meaning neither fuzzy hashing program misclassified any files.

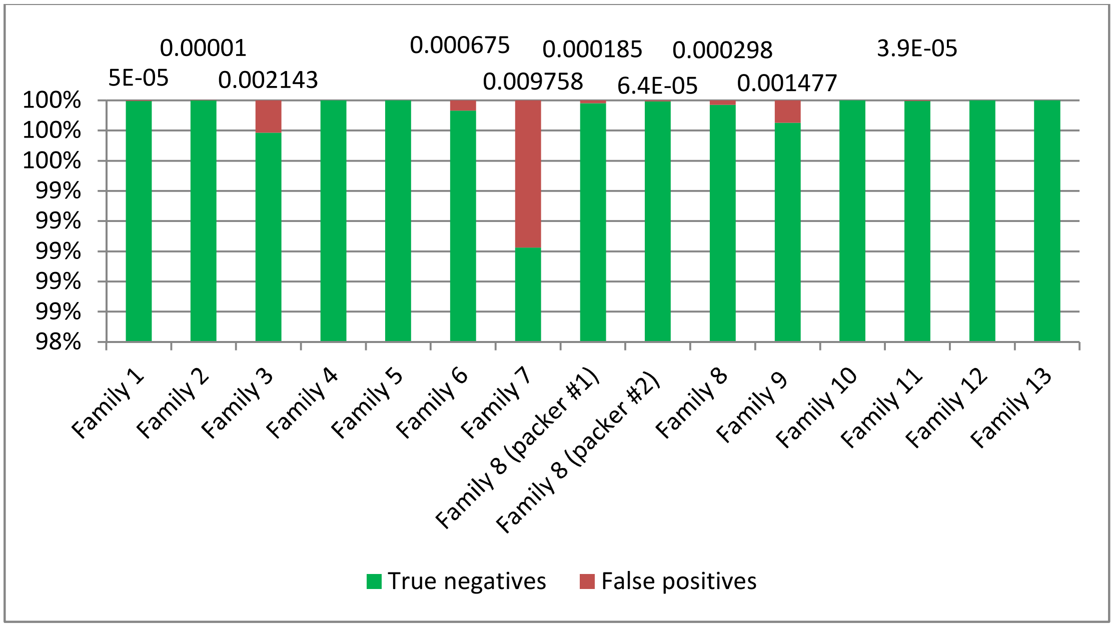


Figure 6: sdhash v0.3 True Negatives vs. False Positives

Figure 6 shows the true negative vs. false positive rate for sdhash. We have added data labels for the non-zero false positive rates, and modified the scale of the chart to attempt to highlight the extremely small false positive rates. Sdhash performs very well in correctly assigning true negatives; however, a very small non-zero false positive rate is introduced. We posit that this is due to the finer resolution of the hash element used (see Roussev 2007).

We summarize these true/false positive/negative rates by computing the precision and recall for both ssdeep and sdhash. Since ssdeep-dc3 performs comparably to ssdeep, we omit its statistics from consideration, as it introduces significantly more comparisons for slightly better performance. First we present the precision of ssdeep and sdhash. Precision represents the percentage of classified files that are relevant, and is computed by computing the ratio of true positives to the sum of the true and false positives.

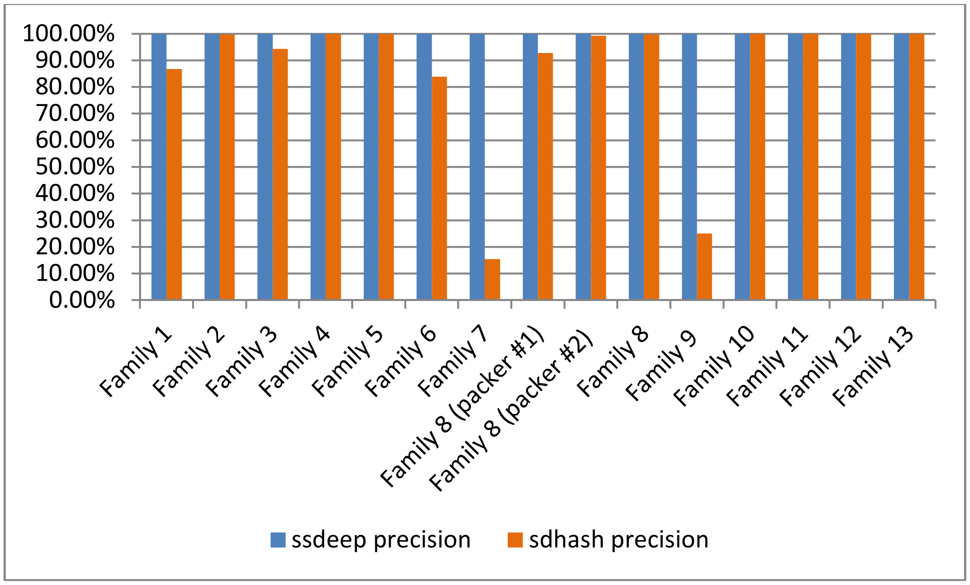


Figure 7: ssdeep vs. sdhashPrecision

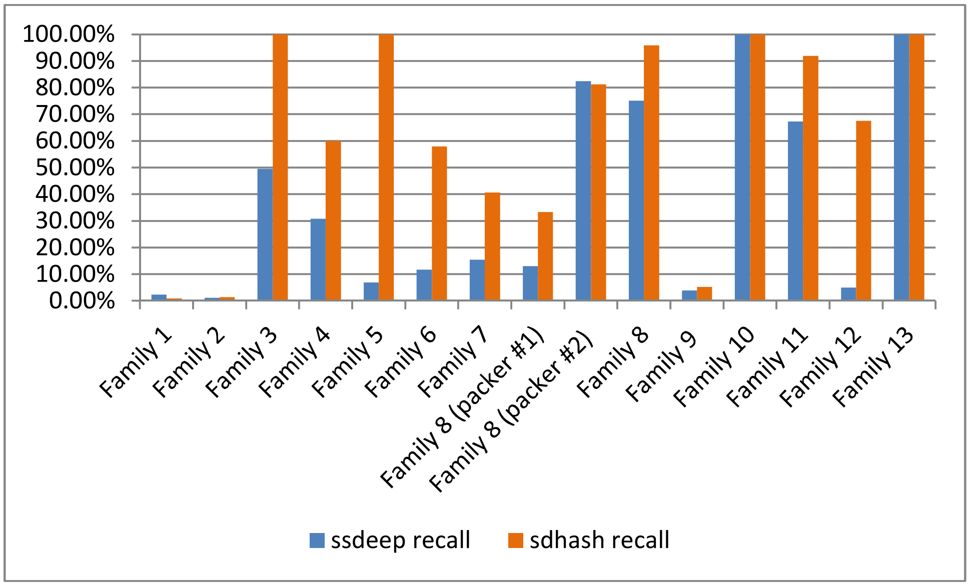


Figure 8: ssdeep vs. sdhashRecall

As shown in Figure 7, ssdeep has generally better precision than sdhash, while sdhash has generally better recall than ssdeep, although the number of files against which these comparisons were performed must be taken into account. These data strongly imply that we may make a rational trade-off between precision and recall when selecting a fuzzy hashing algorithm to use. The generally superior true positive rates of sdhash must be balanced against its non-zero false positive rates, while the high true negative rates of ssdeep offset the generally poorer true positive rates. Operationally, this implies that when maximizing the opportunity to find related files, one might prefer sdhash, and that when minimizing the opportunity to return unrelated files, one might prefer ssdeep.

Of additional note: if we consider the problem of embedding one file into another (for example, one Portable Executable file containing another as a named resource, or a PDF file with an embedded executable), we discover an opportunity for fuzzy hashing to shine in a way not readily accessible to format-specific approaches (such as section hashing). Since available fuzzy hashing techniques treat their input files as unformatted bytes, when the format of two files does not coincide, fuzzy hashing is essentially the only static approach that may possibly detect that a known file is embedded in another. File size considerations become particularly relevant in this scenario (Roussev 2007). This remains a valid (though computationally expensive) application for fuzzy hashing algorithms, and additional research in the cost/benefit of this analysis approach should be pursued.

2.6.3 Alternate Fuzzy Hashing Techniques

In assessing these widely available fuzzy hashing tools, we considered that the modes of hashing the tools presented were not continuous, in that variability in the input sizes and content did not directly correlate to either hash size or hash content. This is generally undesirable, as it allows structural attacks on the hashing algorithms, such as arbitrary insertion of carefully selected bytes to produce a desired output hash. We investigated continuous hashes, and developed a new fuzzy hashing technique called *histo-hash*.

Histogram-based hashing is designed with the goal of creating a simple hash function that has known continuity properties. Thus we avoid the use of pseudo-random hashing functions such as cryptographic hashing; we also avoid the use of context-triggering, and other complicating rules of sampling. We posit that the simplest strategy for down-sampling binary sequence may be to analyze byte-histograms of the binary divided into a sub-division of consecutive subsequences, ordered first by length and then by offset.

Histogram-based hashing takes as input a binary-file and a parameter indicating how many times to subdivide the file; after computing lengths and offsets for a subdivision of the binary-file, counts of bytes are tabulated for each division and a discrete representation of normalized counts are printed to output as the *histo-hash*. Because the output of histogram based hashing is dependent on the observed frequency of bytes, we can precisely bound the variation of the *histo-hash* outputs when one input differs from another by a small number of bytes. Because that type of variation is common in malware families that have configuration blocks, *histo-hash* may be particularly well suited for applications for a quick assessment of similarity.

2.7 Publications and Presentations

This work was presented at the 2011 Malware Technical Exchange Meeting in McLean, VA.

2.8 Bibliography

[Aquilina 2008]
Aquilina, C. M.. *Malware Forensics: Investigating and Analyzing Malicious Code*. Syngress. 2008.

[Brin 1995]
Brin, D. G.-M. “Copy Detection Mechanisms for Digital Documents,” 398-409. Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data. San Jose, CA. ACM, 1995.

[Cohen 1997]
Cohen. “Recursive Hashing Functions for N-Grams.” *ACM Transactions in Information Systems 15* (3). 1997.

[Cross 2008]
Cross, S. *Scene of the Cybercrime*. Syngress. 2008.

[DigitalNinja 2007]
DigitalNinja. (2007, Apr 05). *Fuzzy Clarity: Using Fuzzy Hashing Techniques to Identify Malicious Code*. Retrieved July 24, 2010, from ShadowServer:
http://www.shadowserver.org/wiki/uploads/Information/FuzzyHashing.pdf

[Elkan 2003]
Elkan. “Using the Triangle Inequality to Accelerate k-Means.” *Proceedings of the Twentieth International Conference on Machine Learning*. aaai.org, 2003.

[Hispasec 2010]
HispasecSistemas. (2010). Analysis. Retrieved July 29, 2010, from Virus Total:
http://www.virustotal.com

[Hurlbut 2009]
Hurlbut, D. (2009, January 9). *Fuzzy Hashing for Digital Forensics Investigators*. Retrieved July 21, 2010, from Access Data:
http://www.accessdata.com/downloads/media/fuzzy_hashing_for_investigators.pdf

[Karp 1987]
Karp. “Efficient Randomized Pattern-Matching Algorithms.” *IBM Journal of Research and Development [0018-8646]31*, 2 (1987): 249.

[Kornblum 2006]
Kornblum, J. “Identifying Almost Identical Files Using Context Triggered Piecewise Hashing.” *Elseview Digital Investigation* (2006):S91-S97.

[Kornblum 2010]
Kornblum, J. (2010, May 6). ssdeep. Retrieved July 29, 2010, from Source Forge:
http://ssdeep.sourceforge.net

[Mandiant 2006]
Mandiant. (2006). The State of Incident Response. Retrieved 2010, from Black Hat:
http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Mandia.pdf

[NIST 2010]
National Institute of Standards and Technology. (2010, February 24). ssdeep Datasets. Retrieved July 29, 2010, from National Software Reference Library: http://www.nsrl.nist.gov/ssdeep.htm

[Roussev 2007]
Roussev, R. M. “Multi-Resolution Similarity Hashing.” *Digital Investigation* (2007): S105-S113

[Tridgell 2002]
Tridgell. (2002). Spamsum README. Retrieved July 29, 2010, from Tridge's Junk Code: http://samba.org/ftp/unpacked/junkcode/spamsum/README

[Walenstein 2007]
Walenstein, V. H. “Exploiting Similarity Between Variants to Defeat Malware,” 12. *Black Hat DC*. Black Hat, 2007.

[Wise 1996]
Wise. “YAP3: Improved Detection of Similarities in Computer Program and Other Texts,” 130-134. *ACM SIGCSE 28*, 1 (March 1996): 130-134.

[Zeltser 2010]
Zeltser, L. (2010). Lenny Zeltser. Retrieved July 26, 2010, from What to Include in a Malware Analysis Report: http://zeltser.com/reverse-malware/malware-analysis-report.html

3 Safe Resource Optimization of Mixed-Criticality Cyber-Physical Systems

Dionisio de Niz

In some key industries, such as defense, automobiles, medical devices, and the smart grid, the bulk of the innovations focus on cyber-physical systems. A key characteristic of cyber-physical systems is the close interaction of software components with physical processes, which impose stringent safety and time/space performance requirements on the systems.

Cyber-physical systems are often safety-critical since violations of the requirements, such as missed deadlines or component failures, may have life-threatening consequences. For example, when the safety system in a car detects a crash, the airbag must inflate in less than 20 milliseconds to avoid severe injuries to the driver. Industry competitiveness and the urgency of fielding cyber-physical system to meet Department of Defense (DoD) mission needs is increasingly pressuring manufacturers to implement cost and system performance optimizations that can compromise their safety. Indications of these compromises in the commercial world can be seen in recent automotive recalls, delays in the delivery of new airplanes, and airplane accidents.

3.1 Purpose

Although optimizing cyber-physical systems is hard, cost-reduction market pressures and small form factors, e.g., small, remotely piloted aircraft (RPA), often demand optimizations. An additional challenge faced by DoD cyber-physical systems is the scheduling of real-time tasks operating in environments where the amount of computation to perform is not fixed but depends on the environment. For instance, the computation time of collision avoidance algorithms in RPAs systems often varies in proportion to the objects the RPA finds in its path. This variation is hard to accommodate in traditional real-time scheduling theory, which assumes a fixed worst-case execution time. Nonetheless, scheduling is essential for RPAs and other autonomous systems that must function effectively in dynamic environments with limited human intervention.

As part of our research, we investigated a safe double-booking of processing time between safety-critical and non-safety-critical tasks that can tolerate occasional timing failures (deadline misses). This double-booking approach helps reduce the over-allocation of processing resources needed to ensure the timing behavior of safety-critical tasks. Timing assurance is possible in conventional real-time systems by reserving sufficient processing time for tasks to execute for their worst-case execution time. The typical execution time of these tasks, however, is often less than the worst-case execution time, which occurs very rarely in practice. The difference between the worst-case and typical execution time of these tasks is thus considered an over-allocation.

Our approach takes advantage of over-allocation by packing safety-critical and non-safety critical tasks together, letting the latter use the processing time that was over-allocated (but not used) to the former. This approach essentially “double-books” processing time to both the safety- and non-safety critical tasks. To assure the timing of the safety-critical tasks, however, whenever these tasks need to run for their worst-case execution time, we stop non-critical tasks. We identify this approach as an asymmetric protection scheme since it protects critical tasks from non-critical ones, but does not protect non-critical tasks from critical ones.

3.2 Background

Multiple papers have been published related to overload scheduling. For example, Mejia, Melhem, and Mosse—and Buttazzo, Spuri, and Sensini—use a form of criticality together with a value assigned to job completions [Mejia 2000, Buttazzo 1995]. Their approach is then to maximize the accrued value. In our case, we combine the accrued value of job completion with criticality which is not included in their approaches. In Shih, Ganti, and Sha, the authors describe an approach to map the semantic importance of tasks to quality of service (QoS) service classes to improve resource utilization [Shih 2004]. They ensure that the resources allocated to a high-criticality task are never less than the allocation to a lower-criticality one. In our case, the use of Zero-Slack Rate-Monotonic (ZSRM) scheduling in our approach supports criticality-based graceful degradation while also maximizing total utility.

The elastic task model proposed in Buttazzo, Lipari, and Abeni provides a scheme for overload management [Buttazzo 1998]. In this scheme, tasks with higher elasticity are allowed to run at higher rates when required, whereas tasks with lesser elasticity are restricted to a more steady rate. In our scheme, we also change rates (periods) but they are pre-defined and only changed at the zero-slack instant. We call this *period degradation* and the degradation is carried out in decreasing order of marginal utility, leading to a minimal loss of utility.

Another related work is the earliest deadline zero laxity (EDZL) algorithm [Cho 2002, Cireni 2007]. In EDZL, tasks are scheduled based on the earliest deadline first (EDF) policy until some task reaches zero laxity (or zero slack), in which case its priority is elevated to the highest level. While the notion of zero slack is used in our solution, the existing EDZL results do not consider the notion of task criticalities and are not directly applicable to the mixed-criticality scheduling problem. In addition, no utility maximization is proposed in their approach.

In Baruha, Li, and Leen 2010, the authors propose the Own Criticality-Based Priority (OCBP) schedulability test to find the appropriate priority ordering for mixed-criticality schedulability [Baruha 2010]. Such a scheme is aimed at the certification requirement. In contrast, we are focused on an overbooking approach to improve the total utility of the system.

In this chapter, we present a utility-based resource overbooking scheme that uses marginal utilities to decide which task should get more resources when a task overruns its NCET. When combined with ZSRM, this model allows us to maximize the utility obtained from mission-critical tasks without compromising safety-critical tasks. We call this approach *ZS-QRAM* (Zero-Slack Q-RAM).

3.3 Approach

Real-time systems have relied on scheduling policies that guarantee task response time bounds based on their worst-case execution times (WCET). Unfortunately, two factors make the WCET increasingly pessimistic. On the one hand, processor technologies to improve average-case execution time (ACET), such as cache memory and out-of-order execution, increase the difference between ACET and WCET. On the other hand, the use of complex algorithms whose execution time depends on environmental conditions makes the WCET difficult to estimate. For example, the execution time of a vision-based collision avoidance algorithm depends heavily on the number of objects within view.

The increasing pessimism of the WCET has motivated alternative schemes that can improve the schedulable utilization of the processors without compromising critical guarantees in the system. The Zero-Slack Rate-Monotonic scheduling (ZSRM) is one of these approaches [de Niz 2009]. ZSRM is a general fixed-priority preemptive scheduling policy that is defined for a uniprocessor system. In ZSRM, a *criticality* value is associated with each task to reflect the task's important to the cyber-physical system's mission. Thus, ZSRM is specifically designed for mixed-criticality systems where tasks have different criticality levels and in the case of overloads, more critical tasks must execute to completion even at the complete expense of less-critical tasks. In addition, ZSRM uses a task model with two execution times per task, one considered the WCET during a nominal mode of operation, called *Nominal-Case Execution Time* (NCET), and another named *Overloaded-Case Execution Time* (OCET) that is considered the WCET during an overloaded situation (e.g. when the number of objects to avoid is unusually large). These two execution times are used to schedule the system to ensure that all tasks meet their deadlines if they run for their nominal execution time. However, when some of the tasks run for their overloaded execution times, a criticality ranking is used to ensure that tasks miss their deadlines in *reverse* order of criticality.

ZSRM “overbooks” time, by allocating execution durations for potential use by tasks of different levels of criticality. If a more-critical task needs to execute for its OCET, this task uses those cycles. Otherwise, a lower-criticality task will avail of those cycles. ZSRM therefore supports an asymmetric protection scheme that (a) prevents lower-criticality tasks from interfering with the execution time of higher-criticality tasks, and (b) allows the latter to steal cycles from the former when these higher-critical tasks overrun their NCET.

In avionics systems, safety-critical tasks are primarily concerned with the safety of the flight. The objective of these tasks is to avoid damage. As a result, criticality is a good match to the criticality of the potential damage. In contrast, mission-critical tasks pertain to higher level mission objectives like surveillance or path planning. In this case, the value of the mission objectives is accrued over time and can saturate (e.g., once we detect a certain number of objects of interest in a surveillance mission, it may become more valuable to keep the frames-per-second of a video stream beyond, say 15 frames per second). Hence, in this paper we present a new scheme that uses the utility of the tasks. Specifically, when a task overruns its NCET, the overbooked CPU cycles are given to the task that derives the most utility from the use of these cycles.

Our utility-based resource overbooking combines mechanisms from ZSRM and the Quality-of-Service (QoS) Resource Allocation Model (Q-RAM) [Rajkumar 1997]. Q-RAM uses utility functions that describe the different QoS levels that tasks can obtain along with the resources (e.g., processor time) they consume and the utility the user derives from each QoS level. In simple configurations, Q-RAM primarily takes advantage of the fact that as applications (e.g. video streaming) increase their QoS level, the incremental utility to the user keeps decreasing. This is known as *diminishing returns*. In other words, the utility we get from moving from a QoS level *i* to *i+1* is larger than moving from level *i+1* to *i+2*. For instance, in a video streaming application increasing the frames per second from 15 to 20 gives the user higher utility (i.e., perceived quality) than increasing from 20 to 25 frames per second.

Q-RAM uses the utility functions to perform a near-optimal allocation of resources to different tasks exploiting the diminishing returns property. In particular, the diminishing returns property manifests itself in these functions as a monotonically-decreasing utility-to-resource ratio. This ratio is known as *marginal utility*. Q-RAM uses the marginal utility to perform the allocation one increment at a time starting with the increment that derives the largest utility for the smallest allocation of resources (largest marginal utility). In each of the subsequent steps, it selects the next largest marginal utility increment until the entire resource (e.g. CPU utilization) has been allocated. In the ideal case, the marginal utility of the last allocation (QoS level) of all the tasks is the same.

ZS-QRAM is designed for tasks whose different QoS levels are implemented using different task periods. Task periods are mapped to allocation points in the utility functions where the resource consumption (or utilization) of the task is calculated by diving its execution time (either NCET or OCET) by its period. ZS-QRAM first considers NCET utility functions, and utilizes Q-RAM to do an initial allocation where each increment in the allocation is represented by an increasingly shorter period. If a task overloads at runtime, an overload management mechanism is used to degrade tasks (by selecting a longer period) to keep the task set schedulable. This mechanism uses task utility functions based on their OCET to select the tasks that render the least utility per unit of CPU utilization (marginal utility).

3.4 Collaborations

Collaborating on this effort were Lutz Wrage, Gabriel Moreno, Jeffrey Hansen, Peter Feiler, Mark Klein, and Sagar Chaki of the SEI, Anthony Rowe, Nathaniel Storer, and Ragunathan (Raj) Rajkumar, of the Carnegie Mellon Electrical and Computer Engineering department, and John P. Lehoczky of the Carnegie Mellon University Statistics department.

3.5 Evaluation Criteria

In order to measure the benefit of ZS-QRAM, we developed a metric called *utility degradation resilience* (UDR). UDR measures the capacity of the resource allocator and overload management mechanisms to preserve the total utility of the system as tasks run beyond their NCET and trigger a load-shedding mechanism that degrades the utility of the system.

UDR is measured in a similar fashion to ductility in ZSRM [Lakshmanan 2010]. It is defined as a matrix that evaluates all possible overloading conditions and the resulting consequences over the deadlines of the different tasks. However, instead of accruing only unit values when a task meets a deadline, we accrue the utility of the task. This is formally defined as:

$$\begin{matrix}\langle O_n & \dots & O_1\rangle \\ \langle O_n & \dots & O_1\rangle \\ \langle O_n & \dots & O_1\rangle\end{matrix}\begin{pmatrix} D_nU_n & \dots & D_1U_1 \\ \dots & \dots & \dots \\ D_nU_n & \dots & D_1U_1\end{pmatrix}$$

where:

O_i is a zero-or-one variable that indicates whether the task with the *i* highest utility (larger *i* means higher utility) overruns,

D_i is a zero-or-one variable indicating whether the task mode with the *i* highest utility meets its deadline, and

U_i is the utility of the *i* highest utility task mode.

As an example consider the task set presented in Table 2.

Table 2: Sample Task Set

Task	Period 1	Period 2	C	C^o	U1	U2
τ_1	9	4	2	2	5	6
τ_2	8		2.5	5	8	8

In this task set there are two modes (for the two periods) for task τ_1 and one mode for task τ_2. For this task set, its UDR matrix is as follows.

$$\begin{array}{cc} \begin{array}{c} \langle 1 \quad 1\rangle \\ \langle 1 \quad 0\rangle \\ \langle 0 \quad 1\rangle \\ \langle 0 \quad 0\rangle \end{array} & \begin{pmatrix} 1*8 & 0*6 & 1*5 \\ 1*8 & 0*6 & 1*5 \\ 1*8 & 1*6 & 0*5 \\ 1*8 & 1*6 & 0*5 \end{pmatrix} \end{array}$$

The overloading vectors correspond to tasks $\langle \tau_2 \quad \tau_1 \rangle$ and we enumerate all possible overloadings of the modes $(\tau_{2,1}, \tau_{1,2}, \tau_{1,1})$. That is, the first row is when both τ_2 and τ_1 overload, the second one when only τ_2 overloads, the third when only τ_1 overload, and the fourth when no task overloads. The consequences of the overloads when using ZS-QRAM are. For the first overloading row, $\tau_{2,1}$ meets its deadline, but task τ_1 is degraded from mode $\tau_{1,2}$ to mode $\tau_{1,1}$, which meets its deadline. The second overloading row also degrades task to τ_1 in the same fashion as in the previous row. In the third overloading row, ZS-QRAM allows $\tau_{1,2}$ to overrun meeting its deadline because τ_2 did not overrun. Finally, in the fourth overloading row nobody overloads and, hence, both $\tau_{2,1}$ and $\tau_{1,2}$ meet their deadlines. It is worth noting that only one of the modes of a task is counted as meeting its deadline.

We project the UDR matrix into an UDR scalar by simply adding the resulting utility of the tasks. That is, we add up all the cells in the matrix. In the case of the Matrix presented above the total utility resilience obtained is *54*.

3.6 Results

Using the UDR metric we compare our scheme against the Rate-Monotonic Scheduler (RMS) and a scheme called “Criticality-As Priority Assignment” (CAPA) that uses the criticality as the priority. Our experiments showed we can recover up to 88 percent of the ideal utility that we could get if we could fully reclaim the unused time left by the critical functions if we had perfect knowledge of exactly how much time each function needed to finish executing. In addition, we observed our double-booking scheme can achieve up to three times the UDR that RMS provides.

We implemented a design-time algorithm to evaluate the UDR of a system and generated the scheduling parameters for our runtime scheduler that performs the conflict resolutions of our overbooking scheme (deciding which function gets the overbooked CPU time). This scheduler was implemented in the Linux operating system as a proof-of-concept to evaluate the practicality of our mechanisms. To evaluate our scheme in a real-world setting, we used our scheduler in a surveillance UAV application using the Parrot *A.R. Drone* quadricopter with safety-critical functions (flight control) and two non-critical functions (a video streaming and a vision-based object detection functions).

Our results confirmed that we can recover more CPU cycles for non-critical tasks with our scheduler than with the fixed-priority scheduler (using rate-monotonic priorities) without causing problems to the critical tasks. For example, we avoided instability in the flight controller that can lead to the quadricopter turning upside down. In addition, the overbooking between the non-critical tasks performed by our algorithm, allowed us to adapt automatically to peaks in the number of objects to detect (and hence execution time of the object detection function) by reducing the frames per second processed by the video streaming function during these peaks.

In future work we are extending our investigation to multi-core scheduling where we plan to apply our scheme to hardware resources (such as caches) shared across cores.

3.7 References

Baruha 2010
Baruha, Sanjoy, Li, Haohan, & Leen, Stougie. “Towards the Design of Certifiable Mixed-Criticality Systems,” 13-22. *Proceedings of Real-Time Technology and Applications Symposium*. IEEE, 2010.

Buttazzo 1998
Buttazzo, Giorgio, Lipari, Giuseppe, & Abeni, Luca. “Elastic Task Model for Adaptive Rate Control,” 286-295. *Proceedings of Real-Time Systems Symposium*. IEEE, 1998.

Buttazzo 1995
Buttazzo, Giorgio, Spuri, Marco, & Sensini, Fabrizio. “Value vs. Deadline Scheduling on Overload Conditions,” 90-99. *Proceedings of Real-Time Systems Symposium*. IEEE, 1995.

Cho 2002
Cho, Lee S-K, Han, A, & Lin, K-J. “Efficient Real-Time Scheduling Algorithms for Multiprocessor Systems.” *IEICE Transactions Communications 85*, 2002: 2859-2867.

Cirinei 2007
Cirinei, M. & Baker, Theodore. “EDZL Scheduling Analysis,” 9-18. *19th Euromicro Conference on Real-Time Systems*. IEEE, 2007.

deNiz 2009
de Niz, Dionisio, Lakshmanan, Karthik, & Rajkumar, Ragunathan (Raj). “On the Scheduling of Mixed-Criticality Real-Time Tasksets,” 291- 300. *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*. Washington, D.C. IEEE, 2009.

Lakshmanan 2010

Lakshmanan, Karthik, de Niz, Dionisio, Rajkumar, Ragunathan (Raj), & Moreno, Gabriel. “Resource Allocation in Distributed Mixed-Criticality Cyber-Physical Systems,” 169-178. *International Conference on Distributed Computing Systems*. IEEE, 2010.

Mejia 2000

Mejia, Pedro, Melhem, Rami, & Mosse, Daniel. “An Incremental Approach to Scheduling During Overloads in Real-Time Systems,” 283-293. *Proceedings of the 21st IEEE Conference on Real-Time Systems Symposium*. IEEE, 2000.

Rajkumar 1997

Rajkumar, Ragunathan (Raj), Lee, Chen, Lehoczky, John P., & Siewiorek, Daniel. “A Resource Allocation Model for QoS Management,” 298-307. *Proceedings of the Real-Time Systems Symposium*. IEEE, 1997.

Shih 2004

Shih, Chi-Sheng, Ganti, Phanindra, & Sha, Lui R. “Schedulability and Fairness for Computation Tasks in Surveillance Radar Systems.” *Real-Time and Embedded Computing Systems and Applications*. IEEE, 2004.

4 Measuring the Impact of Explicit Architecture Descriptions

Rick Kazman, Len Bass, William Nichols, Ipek Ozkaya, Peppo Valetto

Many large government software acquisition programs require contractors to create extensive documentation, but this seldom includes sufficient architecture documentation for technical reasoning about quality attributes and their tradeoffs. This choice saves money in the short term but potentially costs more in the long term.

The benefits of architectural documentation have never been quantified and empirically validated, so it is difficult to justify requiring architecture documentation. Our research objective was to attempt to determine the value of architectural documentation: when it was used, by whom, and to what end.

4.1 Purpose

Provide a sound empirical basis for the following:

- determining the value of architectural documentation in a project
- providing government organizations a data point addressing the “how much is enough” documentation question, to support making early architectural tradeoff decisions, and to support evolution

4.2 Background

The Hadoop project is one of the Apache Foundation’s projects. Hadoop is widely used by many major companies such as Yahoo!, eBay, Facebook, and others.[17] The lowest level of the Hadoop stack is the Hadoop Distributed File System (HDFS) [Shvachko 2010]. This is a file system modeled on the Google File System that is designed for high volume and highly reliable storage [Ghemawat 2003]. Clusters of 3,000 servers and over four petabytes of storage are not uncommon with the HDFS user community.

The amount and extent of architectural documentation that should be produced for any given project is a matter of contention [Clements 2010]. There are undeniable costs associated with the production of architectural documentations and undeniable benefits. The open source community tends to emphasize the costs and downplay the benefits. As evidence of this claim, there is no substantive architectural documentation for the vast majority of open source projects, even the very largest ones.

4.3 Approach

When writing architectural documentation it is necessary to have an overview of what the system components are and how they interact. When there is a single architect for the system, the easiest route is to simply talk to this person. Most open source projects, however, do not have a single

[17] See http://wiki.apache.org/hadoop/PoweredBy for a list of Hadoop users.

identifiable architect—the architecture is typically the shared responsibility of the group of committers.

The first step of our documentation process was to gain this overview. Subsequent steps include elaborating the documentation and validating and refining it. To do this we needed to turn first to published sources.

4.3.1 Gaining the Overview

HDFS is based on the Google File System and there are papers describing each of these systems [Ghemawat 2003, Shvachko 2010]. Both of these papers cover more or less the same territory. They describe the main run-time components and the algorithms used to manage the availability functions. The main components in HDFS are the NameNode that manages the HDFS namespace and a collection of DataNodes that store the actual data in HDFS files. Availability is managed by maintaining multiple replicas of each block in an HDFS file, recognizing failure in a DataNode or corruption of a block, and having mechanisms to replace a failed DataNode or a corrupt block.

In addition to these two papers, there is an eight-page *Architectural Documentation* segment on the Apache Hadoop website [Apache 2011]. This segment provides somewhat more detail than the two academic papers about the concepts used in HDFS and provides an architectural diagram, as shown in Figure 9.

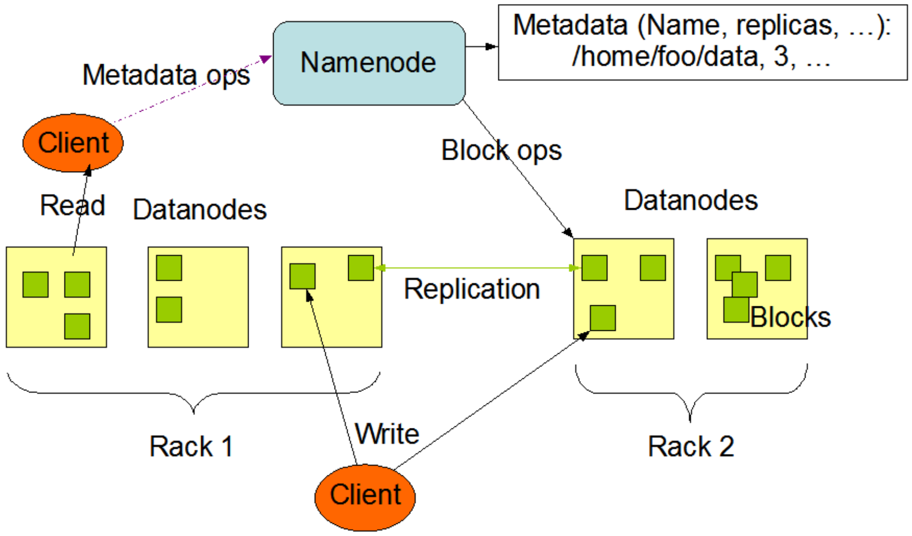


Figure 9: HDFS Architecture Diagram From Hadoop Website

Code level documentation (JavaDoc) is also available on the HDFS website. What currently exists, then, are descriptions of the major concepts and algorithms used in HDFS as well as code-level JavaDoc API documentation.

What is missing from the existing documentation can be seen by considering how architectural documentation is used. Architectural documentation serves three purposes: 1) a means of introducing new project members to the system, 2) a vehicle for communication among stakeholders, and 3) the basis for system analysis and construction [Clements 2010]. These uses of architectural documentation include descriptions of the concepts and, where important, the algorithms. But architectural documentation, to be truly useful, must also connect the concepts to the code. This connection is currently missing in the HDFS documentation. A person who desires to become a contributor or committer needs to know which modules to modify and which are affected by a modification. Communication among stakeholders over a particular contribution or restructuring is also going to be couched in terms of the relation of the proposed contributions to various code units. Finally, for system construction, maintenance, and evolution to proceed, the code units and their responsibilities must be unambiguously identified. Lack of such focused architecture documentation can assist contributors become committers faster. It could also assist addressing many current open major issues.

Architectural documentation occupies the middle ground between concepts and code and it connects the two. Creating this explicit connection is what we saw as our most important task in producing the architectural documentation for HDFS.

4.3.2 Expert Interview

Early in the process of gaining an overall understanding of HDFS, we interviewed Dhruba Borthakur of Facebook, a committer of the HDFS project and also the author of the existing architectural documentation posted on the HDFS website [Apache 2011]. He was also one of the people who suggested that we develop more detailed architectural documentation for HDFS. We conducted a three hour face-to-face interview where we explored the technical, historical, and political aspects of HDFS. Understanding the history and politics of a project is important because when writing any document you need to know who your intended audience is to describe views that are most relevant to their purposes [Clements 2010].

In the interview, we elicited and documented a module description of HDFS as well as a description of the interactions among the main modules. The discussion helped us to link the pre-existing architectural concepts—exemplified by Figure 9—to the various code modules. The interview also gave us an overview of the evolutionary path that HDFS is following. This was useful to us since determining the anticipated difficulty of projected changes provides a good test of the utility, and driver for the focus, of the documentation. Figure 10 shows a snippet from our interview and board discussions where Dhruba Borthakur described to us the three DataNode replicas in relationship to the NameNode.

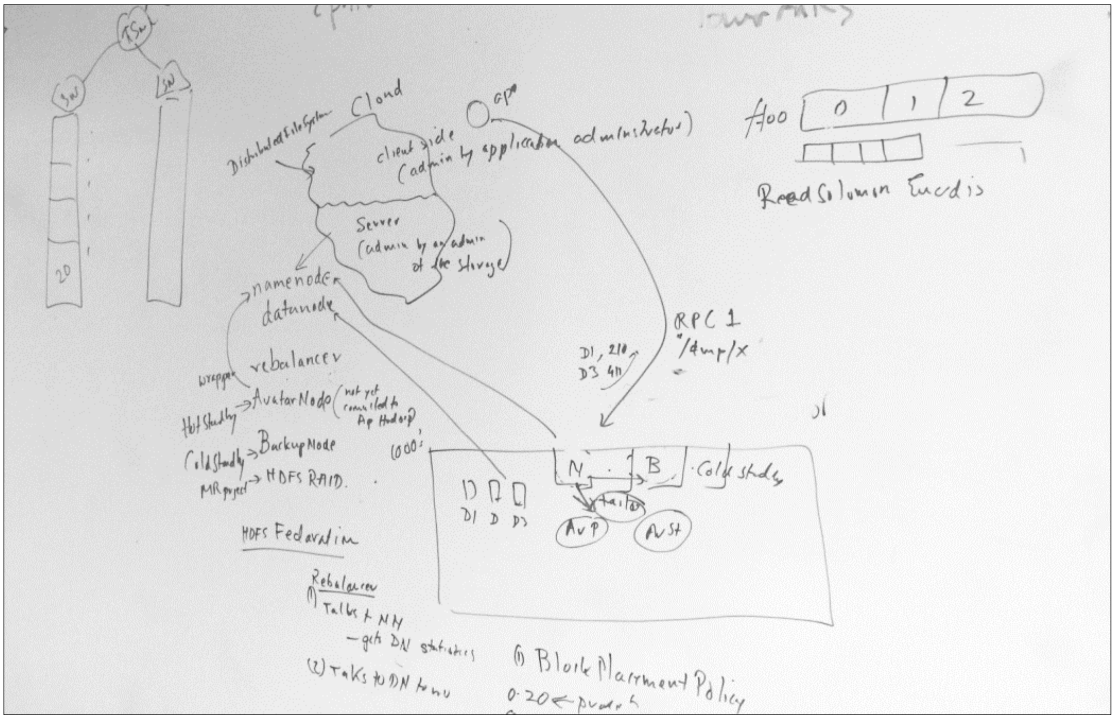


Figure 10: Elicitation of Architectural Information

4.3.3 Directory Structure

A final item that proved very helpful is the directory structure of HDFS. The code is divided cleanly into the following pieces:

- the library used by the client to communicate with the Namenode and the Datanodes
- the protocols used for the client communication
- the Namenode code
- the Datanode code
- the protocols used for communication between the Namenode and the Datanodes

In addition, there are a few other important directories containing functionality that the HDFS code uses, such as Hadoop Common.

4.3.4 Tool Support

An obvious first step in attempting to create the architectural documentation was to apply automated reverse engineering tools. We employed SonarJ and Lattix, both of which purport to automatically create architectural representations of a software product by relying on static analysis of the code dependencies [SonarJ 2011, Lattix 2011]. However, neither of these tools provided useful representations–although they did reveal the complexity of the dependencies between Hadoop elements. For example, Figure 11 shows an extracted view of the most important code modules of HDFS, along with their relationships, produced by SonarJ.

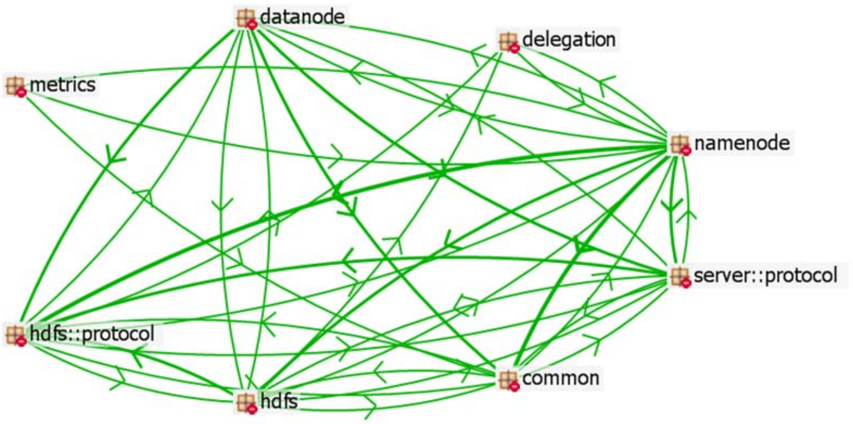


Figure 11: Module Relationships in HDFS

What are we to make of this representation? It appears to be close to a fully connected graph. Is the code a “big ball of mud?” The answer lies in the purpose and goals of the architecture. The main quality attribute foci of HDFS are performance and availability. These concerns dominate the architectural decisions and the discussions amongst the project’s committers. Of significant, but decidedly lesser concern, are qualities such as modifiability and portability. The process Hadoop follows in handling modification is a planned evolutionary processes where a committer suggests an alternative design, it is vetted among the key committers, and then planned for an agreed upon future release cycle. The goals of the project should be aligned with the focus of the architecture. Since performance and availability were the top goals of HDFS, it is not surprising that these concerns shaped the architectural decisions. Since modifiability and portability were of lesser concern, it is also not surprising that these qualities were not strongly reflected in the architectural structures chosen.

The reverse engineering tools SonarJ and Lattix are primarily focused on these latter concerns—modifiability and portability. They aid the reverse engineer in determining the modular and layered structures in the architecture by allowing the definition of design rules to detect violations for such architectural structures. We thus see a mismatch between the goals of the tools and the goals of HDFS. For this reason, the structures that these tools were able to automatically extract were not particularly interesting ones, since they did not match the goals of the project and the important structures in the architecture. HDFS does not have any interesting layering, for example, since its portability concerns are, by and large, addressed by the technique of “implement in Java.” The governing architectural pattern in HDFS is a master-slave style, which is a run-time structure. And modifiability, while important, has been addressed simply by keeping the code base at a relatively modest size and by having a significant number of committers spending considerable time learning and mastering this code base.

The modest code size, along with the existing publications on the availability and performance strategies of HDFS, allowed us to document the architecture by tracing the key use cases through the code. While this is not an easily repeatable process for larger open source projects, it proved to be the most accurate and fit for purpose strategy for creating the architecture documentation of HDFS.

This lack of attention to specific architectural structures aimed at managing modifiability is a potential risk for the project as it grows, since it makes it difficult to add new committers—the learning curve is currently quite steep. Our architectural documentation is one step in addressing this risk. Another step that the HDFS committers could take is to simplify the "big ball of mud."

4.3.5 Validation

The final phase of the initial creation of the architectural documentation was to validate it. We invited external review and have received comments on our documentation from three of the key HDFS committers/architects.

Based on their extensive comments we have corrected, modified, and extended the architectural documentation.

4.3.6 Publication

The validated documentation has been published at a publicly accessible location–http://kazman.shidler.hawaii.edu/ArchDoc.html–and we then linked to this location from the Hadoop wiki: http://wiki.apache.org/hadoop/. We also advertised the publication of the architecture documentation via Hadoop's Jira.

In advance of the publication, we created baseline metrics that we can track and attempt to correlate to the introduction of the architecture documentation such as number of committers and contributors, and turnaround time for Jira items.

4.4 Collaborations

The original SEI team working on this research included Rick Kazman, Len Bass, William Nichols, and Ipek Ozkaya. Later Dennis Goldenson was recruited to help with survey design. In addition Peppo Valetto, from Drexel University, was an active collaborator throughout.

4.5 Evaluation Criteria

The evaluation of this research project focuses on tracking the usefulness of the architectural documentation. We do this in a number of ways:

- by tracking how often it is downloaded and how often it is mentioned in discussion groups
- by tracking project health measures, such as the growth of the committer group, and the time lag between someone's appearance as a contributor and their acceptance as a committer and other measures
- by tracking product health measures, such as the number of bugs per unit time and Jira issue resolution time

Furthermore, we are attempting to understand whether any changes have occurred in the HDFS social network that may be attributable to the introduction of the architecture documentation.

4.6 Results

The usage of the documentation is being tracked using Google Analytics (daily snapshots). As of the time of writing we have had more than 2,300 visits from visitors in 60 different countries. We have averaged just over10 visits per day, with about 45 percent of the page views from return visitors. Also, roughly 12 percent of visitors have visited the site nine or more times, and 67 have visited 50 or more times. This implies that some people around the world are finding the architecture documentation, and are finding it useful.

All project data—code, Jira issues, and social network data (emails, blog posts, Wiki posts, etc.)—have been baselined, from project inception to the present. This will be used to determine the *effects* of the architecture documentation, allowing us to answer the question of whether this documentation changes project activity or product quality.

As the changes in project structure and product quality take time to manifest themselves, the process of collecting and evaluating the data is ongoing.

4.6.1 Repository Data Collection

To facilitate the ongoing collection of data from the project repositories, we have developed a series of reusable tools built upon the Taverna Workflow Management System (www.taverna.org.uk). These tools, known as *workflows*, target a specific data repository for the HDFS project and collect all data from that repository within a specified date range.

There are currently three repositories that are targets for our data collection efforts: the HDFS Jira bug tracking system, developer mailing list archives, and the project's SVN code repository.

From the Jira bug tracker, we are able to gather high-level information about HDFS releases as well as details about individual Jira "issues" (development tasks) within those releases. This data includes all of the properties related to an issue (such as the reporter, assignee, date opened, description, severity, etc.), the issue's entire change history, comments within it, and contributed patches attempting to close the issue.

The developer mailing list archive gives us access to details about communication trends within a time period. We are able to store all details about individual messages and message threads, including message subject, contents, sender address, recipients, date, and reply targets.

The final data source that we use is the SVN code repository for HDFS. Here we are able to see which files are changed within a commit, the details of how those files have been changed, who does the committing, and when the commits happen.

We are able to tie these disparate sources together via user aliases, names, and email addresses to merge the actors across all three data sources. We can also use clues within the data that we collect to bridge the different repositories and provide traceability among them. For example, the Jira issues can be tied to SVN commits by examining the revision number supplied when an issue is resolved. Additionally, mailing list messages can be tied to Jira issues by examining the contents of the subject and message body. They can also be loosely linked to releases by examining the dates that communication threads span.

By linking our data, we gain an accurate map of the users and their interactions as these relate to activity within and across releases of HDFS.

All of the workflows, operating as outlined above, store their results in a relational database. Currently this database contains all project data since June 2009, when HDFS became a project independent from the main Hadoop project within the Apache open source ecosystem.

4.6.2 Analyses

We have also created a series of workflows to perform a number of different analyses on the project data. Our current workflows focus on three primary areas: 1) approximating effort as it pertains to resolving individual issues and releases, 2) mapping participation activity, and 3) determining work and performance trends over time.

4.6.2.1 Effort Analysis

We are able to approximate effort within an issue by analyzing the patch proposals and SVN commits attached to the issue. Each patch or commit contains a set of files that have been changed as well as a diff containing the individual line changes. We use a metric based on lines of code = changed to approximate how much effort a single issue takes to resolve. We then aggregate the effort of all of the issues within a release to calculate the effort per release.

This effort data can then be used to assign a cost for both individual issues and releases within the project. As time goes on, we can track how architectural changes within HDFS impact the cost to implement new features or perform maintenance on the code.

4.6.2.2 Participation Analysis

In addition to effort data, we are also able to map how user participation trends change over time. We have a few different workflows that focus on this kind of analysis.

We have developed an analysis that examines the roles of the project's active users and how a user gets promoted from one role to another. We have defined three primary roles within the project: commenters, contributors, and committers. Commenters refer to those users who have posted a message to an issue or the mailing list. Contributors refer to users who submit patches to attempt to resolve issues. Finally, committers are the users with the final responsibility of committing changes to the source code repository. We look at the dates when a user first comments on an issue, first submits a patch, and first commits a change to determine the promotion time between those roles. We also examine the number of comments between a user's first comment date and first contribution date as well as the number of issues commented on between those dates. We perform a similar analysis between contributions and commits, looking at the number of contributions between a user's first contribution date and first commit date, as well as the number of issues contributed to within that time period. This analysis aims to determine whether there are any trends in terms of the time elapsed between promotions to different roles, controlling for the amount of work carried out while in each role.

Another workflow involves social network analysis. We create a directed, weighted graph structure around the communications recorded within Jira and in the mailing list. The nodes of the graph represent individual users. The edges of the graph represent a message from one user to another, with the weight of the edges corresponding to the number of communications between those users. From this graph, we can look at network statistics like the centrality of users in different roles, to determine which users are central to guiding the conversations within the project community, and whether their central position and involvement changes over time.

4.6.2.3 Project Work Analysis

The final area of analysis that we focused on in this project examines how work statistics change over time. The first workflow that we developed in this area examines issues that remain open at specific intervals in the project. For each issue and at each interval, we are able to calculate how long that issue has been open. Additionally, we are able to look at aggregate statistics related to the number of open issues within a period. These statistics are useful to see how well the project is able to cope with a backlog of issues and reduce it, and how quickly new issues are accumulated.

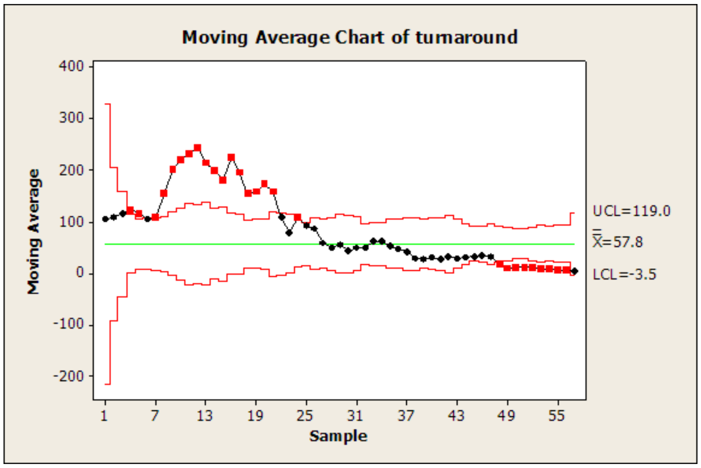


Figure 12: Jira Issue Turnaround Time Statistics

Figure 12 shows the turnaround time statistics trend in between June 2009 and August 2011.

The other workflow we have developed analyzes an issue’s change history to determine the turnaround time per issue. It looks at all of the individual periods in which the issue has been open and subsequently closed to provide an accurate measure of the amount of time that issue has been worked on. From this analysis we can derive aggregate statistics of how the pace of issue resolution changes over time.

4.6.3 Result Conclusions

In summary, we use the following five analysis calculations to monitor the changing state of the HDFS project:

- Effort approximation for issues and releases
- Social network analysis statistics derived from the mailing list and issue comments
- Use role promotion statistics
- Open issue statistics
- Issue turnaround statistics

This data and collection of analyses will be used to determine the *effects* of the architecture documentation, allowing us to answer the question of whether this documentation changes project activity or product quality.

As the changes in project structure and product quality take time to manifest themselves, the process of collecting and evaluating the data is ongoing.

4.7 Publications and Presentations

A paper on our initial architecture reconstruction process and early results was presented at OSS 2011 (the 7th International Conference on Open Source Systems), in Salvador, Brazil [Bass 2011].

4.8 References

[Apache 2011]
Apache Hadoop. "*HDFS Architecture*."
http://hadoop.apache.org/common/docs/r0.19.2/hdfs_design.html (Retrieved Apr. 7, 2011).

[Bass 2011]
Bass L., Kazman R., & Ozkaya O. "Developing Architectural Documentation for the Hadoop Distributed File System," 50-61. *Proceedings of the 7th Conference on Open Source Systems (OSS2011),* Salvador, Brazil, October 2011. Springer, 2011.

[Clements 2010]
Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Merson P., Nord R., & Stafford J. *Documenting Software Architectures: Views and Beyond, 2nd ed.* Addison-Wesley, 2010.

[Ghemawat 2003]
Ghemawat S., Gobioff, H., & Leung, S-T. "The Google File System." *ACM SIGOPS Operating Systems Review 37*, 5 (December 2003): 23-43.

[Lattix 2011]
Lattix. http://www.lattix.com (Retrieved Apr. 9, 2011).

[Shvachko 2010]
Shvachko, K., Kuang, H., Radia, S., & Chansler, R. "The Hadoop Distributed File System," 1-10. *IEEE 26th Symposium on Mass Storage Systems and Technologies*, Incline Village, NV, May 2010. IEEE, 2010.

[SonarJ 2011]
SonarJ. http://www.hello2morrow.com/products/sonarj (Retrieved Apr. 9, 2011).

5 Regression Verification of Embedded Software

Sagar Chaki, Arie Gurfinkel, Ofer Strichman

This chapter summarizes the FY11 SEI Line-Funded Exploratory New Start (LENS) project on regression verification of embedded software.

5.1 Purpose

Technological innovation is the hallmark of the computer hardware industry. Keeping pace with this innovation is a major challenge for software engineering: new hardware makes new resources available, but to take advantage of them the software must be migrated (or ported) to the new hardware platform. This is not trivial. An idealistic approach—developing and validating new software from scratch for every (significant) improvement in hardware—is impractical in view of deadlines and budgetary constraints. For example, it is infeasible to rewrite all of the existing sequential software to take full advantage of the new multi-core CPUs. At the same time, it is dangerous to directly reuse software written for one platform on another. A new platform changes the underlying assumptions and requires the software to be revalidated, at great cost. This is particularly a problem for entities, such as the DoD, that use large complicated software with lifetimes spanning decades and several major technological changes.

Automated program verification is an important tool for ameliorating this re-validation problem. By analyzing the source code of the program statically, it is possible to infer what happens in all possible run-time executions without ever executing a single line of code. Thus, a single program analysis pass reveals how specific changes in platform assumptions affect the execution of the software. However, the main challenges in applying automated verification for validation (or violation) of architectural assumptions is its limited scalability and (often) the need for formal specification of desired behavior.

One technique, regression verification [Strichman 2005, Godline 2009, Strichman 2009], stands out as a way to address those challenges. Regression verification is the problem of deciding the *behavioral equivalence* of two, closely related programs. First, it does not require formal specification. Second, there are various opportunities for abstraction and decomposition that only apply when establishing an equivalence of similar programs. In the best possible scenario, the effort required for regression verification is proportional to only the difference between programs being compared, and not to their absolute sizes. This makes the approach tractable in practice. Although the original definition of regression verification applies to proving equivalence of similar software, we believe that it extends naturally to the comparison of the same software on two different platforms.

While the challenge of migrating software between platforms occurs in many contexts, we are interested in studying the particular case of migration of real-time embedded systems from single-core to a multi-core platforms. There are several reasons for concentrating on this domain in particular. First, those systems are often safety-critical and their validation is necessary and costly. Second, DoD is one of the largest consumer of such systems. Third, the real-time embedded platforms are often highly restricted (a prerequisite for predictable schedulability) which may lead to more scalable analysis. Finally, the SEI has both significant expertise in this domain, as well customers willing to engage in collaborate research on this project. Both factors increase the chances of success of our project.

At a first glance, it may seem that from the safety and security perspective, porting of embedded software to multi-core platforms is trivial. After all, the software has been extensively validated on single-core platforms, and is not being modified. It must, therefore, be functionally unchanged, and remain as safe and secure on multi-core hardware. Unfortunately, this is not so. The key insight is that software's behavior depends not just on itself, but also on the underlying hardware (i.e., on the architectural assumptions provided by the hardware). Catastrophic failures have occurred when legacy software has been used without due consideration to changes in the environment in which the software is being run as exemplified by the infamous Ariane 5 disaster [Ariane 5].

In the context of multi-core platforms, the crucial change in assumption is the presence of *real concurrency* as opposed to *virtual concurrency*. Real concurrency means that multiple threads may run in parallel on multiple cores. Virtual concurrency means that multiple threads run one-at-a-time on a single-core. Hence, in virtual concurrency only one thread has access to the hardware at any point in time.

Virtual concurrency is exploited in the widely used *priority ceiling* mutual exclusion protocols that ensure exclusive access to a shared resource (e.g., memory, and peripheral devices). The essential idea of such protocols is to allow highest-priority threads to access shared resources at will. The platform ensures that no other thread is able to preempt (and execute) while the highest-priority thread is accessing the resource. Priority ceiling is widely used in embedded software owing to its simplicity. It requires no special mutual-exclusion primitives (e.g., locks, semaphores, monitors etc.), and reduces chances of concurrency-related errors like races and deadlocks.

However, priority-ceiling breaks down on multi-core platforms because they have real concurrency. Here, having the highest priority does not guarantee exclusive access to the hardware. This leads to more possible thread interleavings, and, therefore, to races and deadlocks. Therefore, software that runs safely and securely on a single-core platform can misbehave on a multi-core hardware. A major challenge for architecture migration is to detect such problems as early as possible (and take preventive measures).

In this project we take first steps in this direction by extending regression verification from sequential to multi-threaded programs. In particular, we make the following contributions:

- We extend a concept of *partial equivalence* to non-deterministic programs.
- Assuming a bijective correspondence mapping between the functions and global variables of two programs, we develop two proof rules for verifying partial equivalence. The premises of the rules only require verification of sequential programs, at the granularity of individual functions.
- We evaluate the feasibility of the rules on several examples.

5.2 Background

The proposed LENS addresses some of the most crucial problems facing the nation, and the DoD. For example, in its February 2005 report [PITAC Report] titled *Cyber Security: A Crisis of Prioritization*, the President's Information Technology Advisory Committee (PITAC) enumerates several cyber security research priorities that are crucial for maintaining the nation’s technological lead, and its safety and security. The research proposed in this LENS is of direct relevance to several of these research areas, notably:

1. Portable or reusable code that remains secure when deployed in different environments (page 39)
2. Verification and validation technologies to ensure that documented requirements and specifications have been implemented (page 39)
3. Building secure systems from trusted and untrusted components, and integrating new systems with legacy components (page 40)

Limitations of current solutions. A number of different approaches have been investigated to solve our target problem, including testing and static analysis. All have inherent limitations. Testing is non-exhaustive. Critical errors have escaped detection even after years of rigorous state-of-the-art testing effort. This problem is even more acute for concurrent programs where testing is able to explore only a minute fraction of the enormous number of inter-thread interactions. Exhaustive approaches, like static analysis and model checking, do not scale well. More importantly, they require a target specification to verify. Writing down appropriate specifications is known to be an extremely difficult and time-consuming task. In the absence of good specifications, exhaustive approaches provide only limited guarantees. We believe that there is no “silver bullet” for this problem, and novel solutions must be explored to complement and aid existing ones. In particular, we propose to explore the applicability of program equivalence techniques in this context.

Regression Verification. The problem of proving the equivalence of two successive, closely related programs P_{old} and P_{new} is called regression verification [Strichman 2005,Godlin 2009, Strichman 2009]. It is potentially easier in practice than applying functional verification to P_{new} against a user-defined, high-level specification. There are two reasons for this claim. First, it circumvents the complex and error-prone problem of crafting specifications. In some sense, regression verification uses P_{old} as the specification of P_{new}. Second, there are various opportunities for abstraction and decomposition that are only relevant to the problem of proving equivalence between similar programs, and these techniques reduce the computational burden of regression verification [Godlin 2009]. Specifically, the computational effort is proportional to the change, rather than to the size of the original program. This is in stark contrast to testing and functional verification: in testing, a change in the program requires the user to rerun the whole (system) test suite; in formal verification, depending on the exact system being used, reusing parts of the previous proof may be possible, but it is far from being simple and in general not automated.

Both functional verification and program equivalence of general programs are undecidable problems. Coping with the former was declared in 2003 by Tony Hoare as a “grand challenge” to the computer science community [Hoare 2003]. Program equivalence can be thought of as a grand challenge in its own right, but there are reasons to believe, as indicated above, that it is a “lower hanging fruit.” The observation that equivalence is easier to establish than functional correctness is supported by past experience with two prominent technologies: i) regression testing – the most popular automated testing technique for software, and ii) equivalence checking – the most popular formal verification technique for hardware. In both cases the reference is a previous version of the system.

From a theoretical computational complexity point-of-view, equivalence checking is also easier than functional verification (e.g., via model checking), at least under our assumption that P_{old} and P_{new} are mostly similar. One may argue, however, that the notion of correctness guaranteed by equivalence checking is weaker: rather than proving that P_{new} is “correct”, we prove that it is “as correct” as P_{old}. However, equivalence checking is still able to expose functional errors since failing to comply with the equivalence specification indicates that something is wrong with the assumptions of the user. In practice, this is of tremendous benefit. In addition, due to its lower complexity, equivalence checking is often feasible in cases where the alternative of complete functional verification is not.

Regression verification is generally useful wherever regression testing is useful, and in particular for guaranteeing backward compatibility. This statement holds even when the programs are not equivalent. In particular, it is possible [Godlin 2009] to define an “equivalence specification,” in which the compared values (e.g., the outputs) are checked only if a user-defined condition is met. For example, if a new feature—activated by a flag—is added to the program, and we wish to verify that all previous features are unaffected, we condition the equivalence requirement with this flag being turned off. Backward compatibility is useful when introducing new performance optimizations or applying refactoring.

5.3 Approach

The key challenge in this project is that all of the research in regression verification to-date has focused on checking equivalence between syntactically different *sequential* (i.e., one thread of control) programs. In contrast, our goal is to define and check equivalence between two syntactically similar *concurrent* (i.e., multiple threads) programs. Unfortunately, standard notions of regression verification for sequential programs do not extended directly to multi-threaded programs. Thus, our approach was to extended definition of partial equivalence to multi-threaded (or non-deterministic) programs and construct proof rules to verify equivalence under new definition.

Two sequential programs *P* and *P’* are *partially equivalent* if for any input *x* whenever *P(x)* terminates and outputs *y* then *P’(x)* terminates and outputs *y* as well and vice versa. That is, two sequential programs are partially equivalent if for an input that they both terminate, they produce the same result. Note that the behavior of *P* and *P’* on non-terminating computations is ignored (hence the equivalence is *partial*).

Multi-threaded programs are inherently non-deterministic: different runs of a multi-threaded program can produce different outputs even if the inputs are the same, based on the scheduling decisions. Thus, sequential definition of partial equivalence, as given above, does not apply. In particular, under that definition a multi-threaded program *P* is *not sequentially partially equivalent* to itself. Thus, we clearly need a better definition of equivalence.

Let *P* be a multi-threaded program. *P* defines a relation between inputs and outputs, which we denote by *R(P)*. Let *Π(P)* denote the set of terminating computations of *P*.

Then:

$$R(P) = \{(\boldsymbol{in}, \boldsymbol{out}) | \exists \pi \in \Pi(P). \pi \text{ begins in } \boldsymbol{in} \text{ and ends in } \boldsymbol{out}\}$$

We define two non-deterministic programs *P* and *P'* to be *partially equivalent* if and only if *R(P)=R(P')*. Note that the definition refers to whole programs and that by this definition every program is equivalent to itself. The definition generalizes the notion of sequential partial equivalence. That is, if two sequential programs *P* and *P'* are sequentially partially equivalent, then they are partially equivalent with respect to the above definition as well.

If programs are finite (i.e., loops and recursion are bounded) then detecting partial equivalence is decidable. However, we are interested in the unbounded case. Our approach is to construct proof rules that decompose the verification problem to the granularity of threads and functions. Note that because of the undecidability of the problem, the rules are necessarily incomplete. That is, they may fail to establish equivalence of two partially equivalent programs.

5.4 Collaborations

This LENS was a collaborative effort between members of the SEI's Research, Technology and System Solutions program, Dr. Sagar Chaki and Dr. Arie Gurfinkel, and a visiting scientist, Prof. Ofer Strichman, who is on the faculty of the Technion, Israel Institute of Technology.

5.5 Evaluation Criteria

The deliverables for this project were

- a peer-reviewed publication presenting the formal foundations of regression verification for multi-threaded software
- preliminary experiments to evaluate the feasibility of the approach

All deliverables were completed successfully. In addition, the SEI has gained the expertise in the state-of-the-art research on regression verification.

5.6 Results

As described above, we have extended the notion of partial equivalence to multi-threaded (or more generally, non-deterministic) programs. Our main results are two proof rules to reduce the problem of deciding partial equivalence to sequential partial equivalence of functions. The rules themselves are too complex to present here. Full details are available in Chaki 2012.[18] In here, we briefly summarize the key insights of the rules.

In both rules, we assume that there are two programs *P* and *P'*, and a mapping between functions of *P* and *P'* such that function *f* of *P* is equivalent to *f'* of *P'*, and a mapping between global variables such that a variable *g* of *P* is equivalent to variable *g'* of *P'*.

[18] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. "Regression Verification for Multi-Threaded Programs," to appear in 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12), Philadelphia, PA, USA.

The first rule is based on an observation that two multi-threaded programs are equivalent if their corresponding functions are equivalent in *arbitrary* environment. The rule is stated formally as follows:

$$\frac{\forall i \in [1..n] \,.\, \delta(f_i)}{p.e.(P, P')}$$

Here $\delta(\,f\,)$ means that the function f in program P is equivalent to the corresponding function f' in the program P' when executed under arbitrary but equal environments. The notion of *environment* is extremely important since in multi-threaded scenario each thread provides an environment, by modifying global variables, to each other thread. Thus, as a function *f* of one thread is executing, the other threads can modify global variables and influence the outcome of *f*.

To reduce this to sequential equivalence, we had to explicate the model of the environment. To check that two functions *f* and *f'* are partially equivalent in arbitrary environment, we "execute" *f* and record all environment observations of *f*. Then, we "execute" *f'* and force the environment observations to be consistent with those seen by *f*. The two functions are equivalent under arbitrary environments if they agree in all such "executions". Note that "execution" is in quotes since we do not actually execute the non-deterministic functions, but rather reason about all of their executions symbolically.

The limitation of the first proof rule is that it is very strong. It requires the functions of both programs to behave identically under any environment. However, in practice, only the environment provided by other threads is important. To address this limitation we have developed a second proof rule that takes the actual environment generated by other threads into account. This rule is stated formally as follows:

$$\frac{\forall i \in [1..n] \,.\, \Delta(f_i)}{p.e.(P, P')}$$

Here $\Delta(\,f\,)$ means that the function f in program P is equivalent to the corresponding function f' in the program P' when executed under arbitrary but equal environments that are consistent with the behavior of the remaining threads in the programs. The main challenge here is that environment produced by other threads is unbounded. Our solution is to show that for reasoning about loop-free recursive function, only a bounded part of that environment is necessary. Thus, we are able to use a finite abstraction of the environment without losing precision. The details of the abstraction and second proof rule can be found in Chaki 2012.[19]

5.7 Bibliography/References

[Ariane 5]

Ariane 5 Flight 501. http://en.wikipedia.org/wiki/Ariane_5_Flight_501.

[19] Sagar Chaki, Arie Gurfinkel, and Ofer Strichman. "Regression Verification for Multi-Threaded Programs," to appear in 13th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12), Philadelphia, PA, USA.

[Godlin 2009]
Godlin, Benny & Strichman Ofer. "Regression Rerification," 466-471. *Proceedings of the 46th Design Automation Conference*, DAC 2009, San Francisco, CA. ACM, 2009.

[Hoare 2003]
Hoare, C. A. R. "The Verifying Compiler: A Grand Challenge for Computing Research." *Journal of the ACM 50*, 1 (2003): 63-69.

[PITAC Report]
President's Information Technology Advisory Committee (PITAC). *Report to the President on Cyber Security: A Crisis of Prioritization* (February 2005), http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf.

[Strichman 2005]
Strichman, Ofer & Godlin, Benny. "Regression Verification—A Practical Way to Verify Programs," 496-501. In *Proceedings of First IFIP TC 2/WG 2.3 Conference on Verified Software: Theories, Tools, Experiments*, VSTTE 2005, Zurich, Switzerland. Springer, 2005.

[Strichman 2009]
Strichman, Ofer. "Regression Verification: Proving the Equivalence of Similar Programs," 63. In *Proceedings of 21st International Conference on Computer Aided Verification*, CAV 2009, Grenoble, France. Springer, 2009.

6 Sparse Representation Modeling for Software Corpora

William Casey, Jeffrey Havrilla, Charles Hines, Leigh Metcalf, Aaron Shelmire

6.1 Purpose: Accurate, Reliable, and Efficient Malware Discovery

Modern software systems are complex, measured in millions of lines of code for operating systems; further, they include a vast assortment of third-party applications and web services. Measuring the security risk from adversarial users who exploit vulnerabilities in these information systems in order to install malicious software (also known as *malware*) is now an engineering imperative [Anderson 2008]. To improve security measures the research community has focused on a practical process of identifying software artifacts producing anomalous behavior, triaging these artifacts for deep analysis to determine design and intent, and using conclusions to update working assumptions of baseline, benign, and malicious system behavior.

The outcome of malware triage and analysis is dramatically improved if a provenance of software artifacts can be identified, in particular if specific attributes of suspected malware can be used to identify similarities to a set of known malware artifacts. Discovered links to known artifacts increases awareness of potential threats. To succeed, two things are needed: a rich collection of malware artifacts (e.g., a reference corpus) comprising known, previously analyzed, or previously identified samples; and efficient methods to query the reference data to determine similarities. These two ingredients provide a general method for checking any unknown software artifact against a reference corpus for possible links, a process called “discovery.” The discovery process applied to a reference corpus of malware artifacts may be instrumented to identify threats.

Currently search/retrieval, let alone similarity tools, for malware reference data are struggling to keep pace with the rapid growth of malware reference data sets. The success of a malware analysis and triage program depends on several factors including the quality of the reference data (coverage and specificity), but also on the critically important question of how quickly the discovery process can be applied. While assessing the quality of malware reference data is a vast data curation challenge, the question of how quickly and precisely we can exhaust malware reference corpora for provenance studies is an important problem of scale with measurable outcomes.

Specifically we address the problem of improving search/retrieval and similarity measures for malware artifacts where the outstanding challenges are to improve

- sensitivity: results correctly identify provenance and do not introduce any miss-identifications (i.e., false positives)
- completeness: leaving no possibility of unawareness (i.e., false negatives)
- robustness: capable of processing corrupted text and working in the presence of errors in malware corpus data
- scalability: application to large data sets

In this chapter we discuss existing malware similarity techniques as well as notions of software provenance. We then introduce our approach exploiting sparse representation data structures to organize malware reference data for optimized search/retrieval, efficient identification of code-clones, string-scales, and the Longest Common Substring (LCS) similarity measure. Finally we conclude this chapter with an examination of outstanding challenges in the area.

6.2 Background

To defend computer networks, the research community has adapted an approach including the collection of forensic data associated with historical compromises/attacks, analyzing these events for patterns of threat. When malware is involved the collection and curation of malware reference data creates an important type of forensic tool. Malware reference data includes the collection of malicious code, behavioral attributes of code [Bayer 2006, Bayer 2009, Morales 2011], and static attributes of code either codified in the file format (e.g., section offset-map [Casey 2010a]) or interpreted from binary (e.g., position independent code analysis [Shabtai2011] and function extraction [Bayer 2009, Sayre 2009, Pleszkoch 2008]).

Malware reference data can be an effective tool for discovery of threats, both known and derived, when combined with exact and approximate matching. An unknown artifact which meets similarity thresholds when compared to reference data may be triaged for deeper analysis to investigate potential threats. While there are several efficient concepts of data matching including fuzzy-hashing [Roussev 2007, Roussev 2011], PIC-matching [Shabtai 2011], call graph matching [Hu 2009], bloom filters [Song 2009, Roussev 2011, Jang 2010] and Bertillonage [Godfrey 2011], the tendency is that efficiency is the tradeoff of completeness or specificity.

One particularly appropriate concept of matching, used to determine provenance in the area of software engineering, having both exact and approximate notions is the problem of identifying code-clones [Cordy 2011]. A code-clone is the result of a copy-and-paste operation taking source code from one file and putting it into another with possibly small but functionally unimportant modifications. For malware, the source code is rarely available, and this match concept may be applied to binary artifacts as well. The challenges for code-clone identification in malware may also be compounded by deception techniques such as obfuscation [Rhee 2011]. Enhancing the malware reference data to include as many attributes for each artifact as possible, including de-obfuscation procedures and runtime analysis, may defeat deception attempts but leads to a larger problem of identifying critical links in a larger and more general set of data associations. The problem of provenance must then rely on identifying key links amongst a vast possible set of noisy, uninformative links.

Size and redundancy of data in malware and software corpora affects discovery tools. For large and dynamic data sets in computational biology, advanced methods to support efficient information retrieval operations have been developed [Bieganski1994, Gusfield1997, Homann 2009, Gurtowski 2010] where the term *homology* describes similarity in DNA and protein sequences. One related solution for homology studies in bio-informatics is based on the development of suffix trees [Ukkonen 1992, Ukkonen 1995, Gusfield 1997] and this is a particularly important direction for our work relating to malware corpora.

The overall goal of determining provenance in malware relies heavily on quality of malware reference data. Reference data quality (for malware) is difficult to measure due to problems associated with sampling hidden populations but is suggested by normative notions of good coverage, specificity, and annotation from expert analysts and reverse engineers. Toward the goal of enhancing malware reference data there are efforts to increase automation and develop a common language for repeatability of studies [MITRE 2010].

Diverse malware reference data combined with quicker-performing search/retrieval and similarity tools improve malware provenance and increase the feasibility of real time capabilities for threat discovery.

6.3 Approach

Our approach to improving the search/retrieval and similarity tools for malware provenance is to design methods to exploit the underlying sparsity of raw data for efficient resolution of string matching. We instrument suffix tree data structures and algorithms found in bio-informatics studies [Bieganski 1994, Gusfield 1997, Homann 2009, Gurtowski 2010] and augment these for application to malware reference data sets. The suffix tree may be constructed online and in linear time (linear in the input size) [Ukkonen1992, Ukkonen 1995]; further it may be used to invert the indexing of a string (file) and identify all string-matches or [Gusfield 1997]. Our contribution was to augment an externalized version of the suffix tree to invert the entire corpus (indexing of content) and summarize rank statistics for all exact matches which satisfy criteria based on string length, content (e.g., entropy), coverage subset (files in the corpus have the particular string as a substring) and string locations (offset positions within files).

Specifically we have augmented data-structures and algorithms to support these applications:

- Given a query string (from any artifact), locate all occurrences in a software corpus in time proportional to the length of query string and number of match occurrences, but constant in the total size of the corpus.
- Given an artifact, query the file against all malware reference data and determine code-clones in the reference data in time linear in the artifact size but constant in the number of artifacts in the reference data set.
- Given a set of artifacts, resolve the top ranked long common substrings (code-clones) within the set in time proportional to the sum length of artifacts. Ranking is measured in terms of an objective function in variables: string length, content (e.g., entropy), coverage subset, and positional offsets within artifacts.
- Design a measure expressing code similarity as the total number (or length) of long common sub-strings. This can be applied to poorly defined families to view sub groups and improve family identifications.

In order to efficiently invert a corpus of data we implemented linear-time construction algorithms for suffix trees [Ukkonen 1992, Ukkonen1995]. Suffix trees are further able to determine LCS in linear time (linear in the input size) [Gusfield 1997, Ukkonen 1995]. We augmented the suffix-tree data structure by adding an index table for multiple files [Gusfield 1997]. In order to scale to large data sets (larger than system memory) we designed the data structure and algorithm to include I/O aspects so data structures could be stored either statically on disk (external) or in memory and recalled by methods that seek to minimize input and output (see Arge for a description of IO bound or external-memory algorithms and static analysis) [Arge 1996]. To investigate external memory management in general and how they may be tailored to specific data sets, we created a general memory model that can use a variety of memory-management strategies including Least Recently Used (LRU) as a default.

To incorporate measures of coverage (which artifacts contain a particular string) we sought to enhance a traversal process for the suffix tree data structure to include lightweight set encodings based on a technique called bit-arrays but designed for sparse arrays (implemented with a red-black tree data structure) which works advantageously over sparse cover sets. The enhancement of the data structure to include support for cover sets led to new query capabilities providing authoritative answers to questions such as:

- "What is the longest common substring in 50 percent of a given artifact set?"

To prevent uninteresting results (e.g., byte padding sequences or null pads which are long stretches of zeros) we further enhanced the suffix-tree data structure to include a computation of Shannon entropy (a rough indication of how interesting a string is) [Shannon 1948]. Together these considerations allows for authoritative answers to questions such as:

- "What is the longest common substring exceeding a threshold of entropy (1.2 log base 10) and found in 50 percent of a given artifact set?"

We generalized the modifications of the traversal to include a language allowing us to select and aggregate rank statistics for strings matching selection criteria comprised of logical conjunctions over the following primitives:

- string length; for example, select strings that exceed a given length
- corpus coverage, number or percent of files containing a string; for example, select strings found in 80 percent or more of the malware reference corpus
- string selection based on specific corpus coverage; for example, select strings found in reference file A and B but not in file C
- string selection based on its specific location within reference files; for example, select strings by requiring that they start in the first 50 bytes of a file
- string selection based on statistical measures; for example, select a string if its Shannon entropy exceeds a given value

Because malware reference data may contain errors such as malware family identity, we provide an example scenario showing how some of the primitives above may be combined to create robust queries, and overcome some inherent problems of malware data sets and artifacts.

Scenario: We are focused on a specific malicious file that is alleged to be related to a recent virus family with poor family identification (e.g., anti-virus researchers have conflicting signatures of defining terms). While we can assemble a reference corpus of all candidate artifacts, and despite the low certainty of member identity, we can still envision a search concept that may lead to precise findings: we wish to match not against every file but against some percentage of the candidate files as follows:

- Identify all strings of minimum length 200 bytes, having a Shannon entropy of 1.2 (log base 10) or higher and found both in our specific malicious file and in at least 50 percent of the candidate artifact set.

The above search criteria defines a robust technique because even if a minority of files in the candidate set are misclassified the result may still identify code-clones found to be common to the specific malicious file and a majority of samples indicating the underlying malware family. The entropy threshold limits the number of non-interesting results such as long stretches of byte padding.

Technically in the worst case the queries require traversal of the suffix tree (linear in the size of the corpus) to compute selection criteria primitives. Thus we are able to construct and query the malware reference data in worst-case linear time in the size of the malware reference data set.

Collectively we refer to our approach including the data structures indicated above as *sparse representation* method. Using sparse representation we have been able to implement efficient search/retrieval operations for malware-reference data, an ability to identify shared substrings, developed a concept for similarity to known malware families, developed a method for extracting string-scales (largest set of strings common to a set) and develop efficient methods to measure file-to-file similarity and cluster files based on longest common substrings.

The remainder of this section illustrates several applications for the sparse-representation data-structures.

6.3.1 Query an Artifact Against a Malware Reference Data Set

This application addresses the following basic question: given an unknown artifact what parts (if any) have been identified in a previously known malware family? Results may then be triaged into a focused question: are these snippets of code substantial evidence to support or reject a hypothesis of shared provenance?

Below we demonstrate the application of comparing against a malware reference data set comprised of a sample of a malware-grouping called *Aliser*, (determined to be a file-infector [Casey 2010a]). We created an *Aliser* reference data set from a sample of 80 files sampled randomly from a set of files which anti-virus researchers identified by terms “Aliser.” Given another artifact selected from the anti-virus data set but not from the sample, we used this method to identify code-invariance from artifact-to-samples.

File-infectors affect header, entry-point, and append small pieces of executable code to infected-files, therefore (since the overall modifications to the files are small) characterizing these files using inexact standard techniques of code comparison is non-informative to misleading [Casey 2010a]. File infectors provides a class of examples where the exactness (specificity) and completeness of the suffix tree provides better discovery capabilities than inexact methods such as fuzz-hashing or file structure match methods. What is needed in this case is an exact string match capability that can scale to large data sets and pinpoint exactly which strings are common to all artifacts.

To resolve this problem we consider each offset (or equivalently the suffix starting from each offset) of the query artifact and use the suffix tree to resolve two separate concepts of match: how deep does this suffix (starting at each offset) match to 1) any of the samples and 2) all of the samples. After suffix tree construction for the sample (linear in the total number of bytes in sample) we may resolve this problem in time linear to the query artifact size—no matter how large the sample size (reference data) nor how much matching the query presents.

In Figure 13 we plot (as a function of the offset in the query artifact) the length of the longest common string match to 1) any of the *Aliser* samples and 2) all of the *Aliser* samples; the plot is windowed into a specific region of interest showing a long string which matches query artifact and all *Aliser* samples in the reference data set.

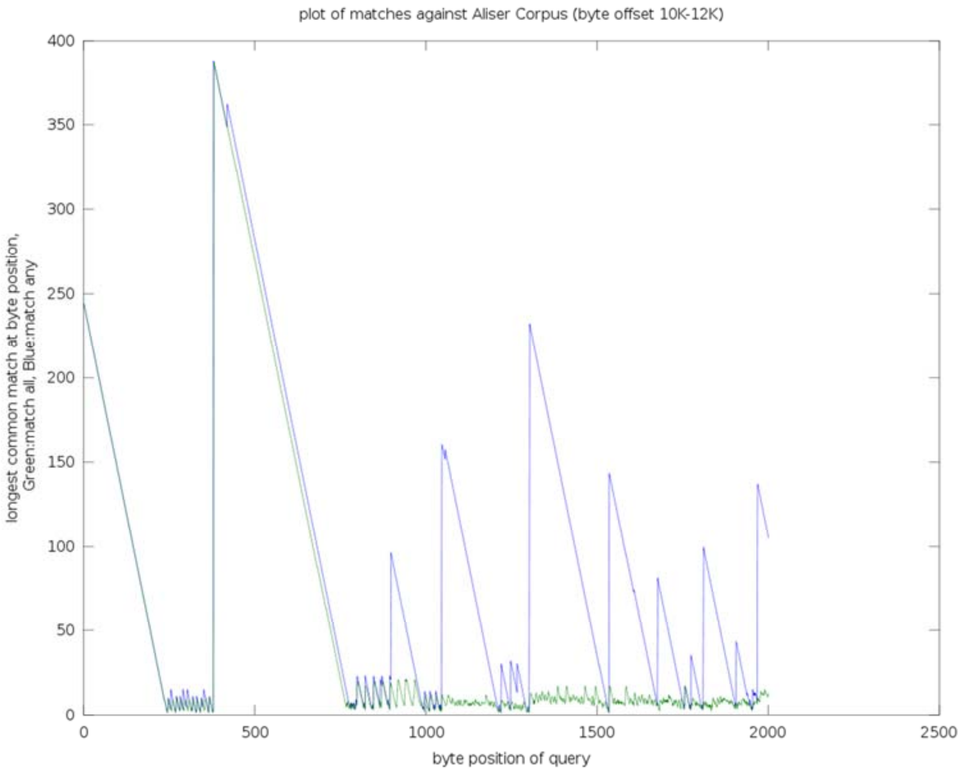


Figure 13: Plot of Matches Against Aliser Corpus

This application is a powerful tool to identify code-sharing when applied to known malware data and could assist greatly in the understanding of code flow across malware families. We plan to develop reference data sets on a per-family basis by leveraging known families found in the CERT Artifact Catalog and instrument the above application to identify family assignment of unknown malware.

6.3.2 Search and Retrieval of Long Common Substrings to Investigate Alleged Code Sharing

This application addresses one of the most common questions arising in malware analysis: Given some preliminary evidence that two artifacts are related can we identify good evidence of a relation? In this section we show how this problem can be resolved with the evidence of code-clones or shared code. For families of artifacts with alleged relations can the specific shared content in the different families be identified quickly in a discovery process that could triage items for further investigation?

There are two immediate benefits to determining relationships between separate artifacts. A more thorough understanding of threat actors may be gained from the interrelations of artifacts. This will help defenders. The defenders will be able to better understand the motives behind the attacks, and determine possible next steps the attacker may take. Knowing these next steps can provide the defenders with indicators such as related IP addresses, domain names, MD5 hashes of artifacts, or more soft indicators such as network traffic patterns. Additionally, by relating custom artifacts to one another, law enforcement cases may be amended to include further illegal acts by the same actors. This may allow for a more coordinated effort among the investigators.

Below we demonstrate this application for four files from two malware families that are allegedly related. These artifacts are from the *Stuxnet* [Falliere 2011] and *Duqu* [Symantec 2011] families whose alleged relation is outlined in Falliere. The *Stuxnet* artifacts are extremely complex pieces of standalone software designed to disrupt or sabotage a specific industrial control system setup. The *Duqu* artifact appears to be designed to give an attacker an initial foothold in the victim organization, allowing further download of additional executables. Even with these completely different purposes both artifacts have similar techniques and procedures.

Initially, 30 files were analyzed in the artifacts. From this initial set four artifacts have been identified as a test set. These were identified by eliminating packed files and finding a balanced set with equal numbers of artifacts from both families. This sample comprises four files (two *Duqu* artifacts and two *Stuxnet* artifacts) totaling 106,696 bytes of data. The longest string that matches all four files is 172 bytes in length. There are additional string matches which, while not the longest, were also shown to be of interest as they pointed to code recovered in IDA-Pro disassembly. Currently, this analysis is ongoing but includes the identification of about 105 distinct code-clones for triage to analysis resources.

More specifically there are 27 code-clones shared among all four artifacts which have a length of 50 bytes or more and entropy 1.2 (log base 10) or more. These 27 code-clones comprise 2087 bytes of distinct content representing 9080 bytes from the four artifacts. Thus the amount of data (content) common to all four artifact amounts to 8.51% of all data. That this important sub-set of data can be determined in a matter of minutes using sparse representation methodology is a significant step toward improving our ability to reason about artifact provenance.

By relaxing the criteria that code-clones have to be found in all artifacts to a less stringent criteria of being found in two or more we are then able to identify 105 code-clones providing context for 44,175 bytes of distinct content in a total of 106,696 bytes for four artifacts. This results show that 41.4 % of all data can be explained by 105 code-clones found in two or more artifacts.

Index	**File**
1	stuxnet/1e17d81979271cfa44d471430fe123a5
2	stuxnet/f8153747bae8b4ae48837ee17172151e
3	duqu/0eecd17c6c215b358b7b872b74bfd800
4	duqu/4541e850a228eb69fd0f0e924624b245

The longest common substring (code-clone) in all four files is 172 bytes in length. Further, while this matches all files, there are longer extended sequence matches in smaller subsets: a 280-length string stemming from the same location is found in the *Stuxnet* files and a length 744 sequence is found in the *Duqu* files. The strings were identified by focusing on the longest match in all four files exceeding an entropy of 1.2 (log base 10).

Tree (id, parent)	Index set	Entropy	Match Length	Position
(0,)	1 2 3 4	3.091597	172 bytes	NA
(1,0)	1 2	3.240296	280 bytes	NA
(2,1)	1	NA	NA	stuxnet/1e17...23a5 @ 0x492a
(3,1)	2	NA	NA	stuxnet/f815...151e @ 0x4416
(4,0)	3 4	3.328524	744 bytes	NA
(5,4)	3	NA	NA	duqu/0eec...d800 @ 0x56e4
(6,4)	4	NA	NA	duqu/4541...b245 @ 0x56e4

The above result provides a clue into the visibility that the sparse representation data structures provide. Although the longest string in common to all four files is 172 bytes it can be extended to longer features in either the *Duqu* or *Stuxnet* family. This type of characterization of strings by set appears to describe the structure of code families in a rich and meaningful way. We plan to develop a descriptive language for code features (such as the one above) which may have descriptive properties analogous to the way we describe geographical features (such as the way *ridge* describes a linear highpoint separating two lower lying areas).

The longest match exceeding entropy of 1.2 (log base 10) in all four files is the following string of 172 bytes (representing hex encoding):

```
0000290352746C44656C657465456C65      6D656E7447656E657269635461626C65
0000B4014B6547657443757272656E74      5468726561640000830352746C4C6F6F
6B7570456C656D656E7447656E657269      635461626C6500006A0352746C496E69
7469616C697A6547656E657269635461      626C65006E0352746C496E7365727445
6C656D656E7447656E65726963546162      6C650000CC0352746C55706361736555
6E69636F6465436861720000
```

Focused investigations involve building suffix trees on a corpus of artifacts with reference sets numbering hundreds or less. Traversing them with selection criteria, and performing retrievals of interesting strings can all be done in minutes to hours (e.g., real time) on a laptop computer.

To put the longest common match in context of all multi files matches, we visualize below all matches to the criteria that they exceed 50 bytes of length, entropy greater than 1.2 (log base 10) and having a coverage size of two or greater distinct files. In Figure 14 below binaries are drawn as the outer ring of the circle, with sections (file format) identified by colors. Matches are illustrated as arcs through the interior of the circle and are colored green to red to indicate the entropy value of the string.

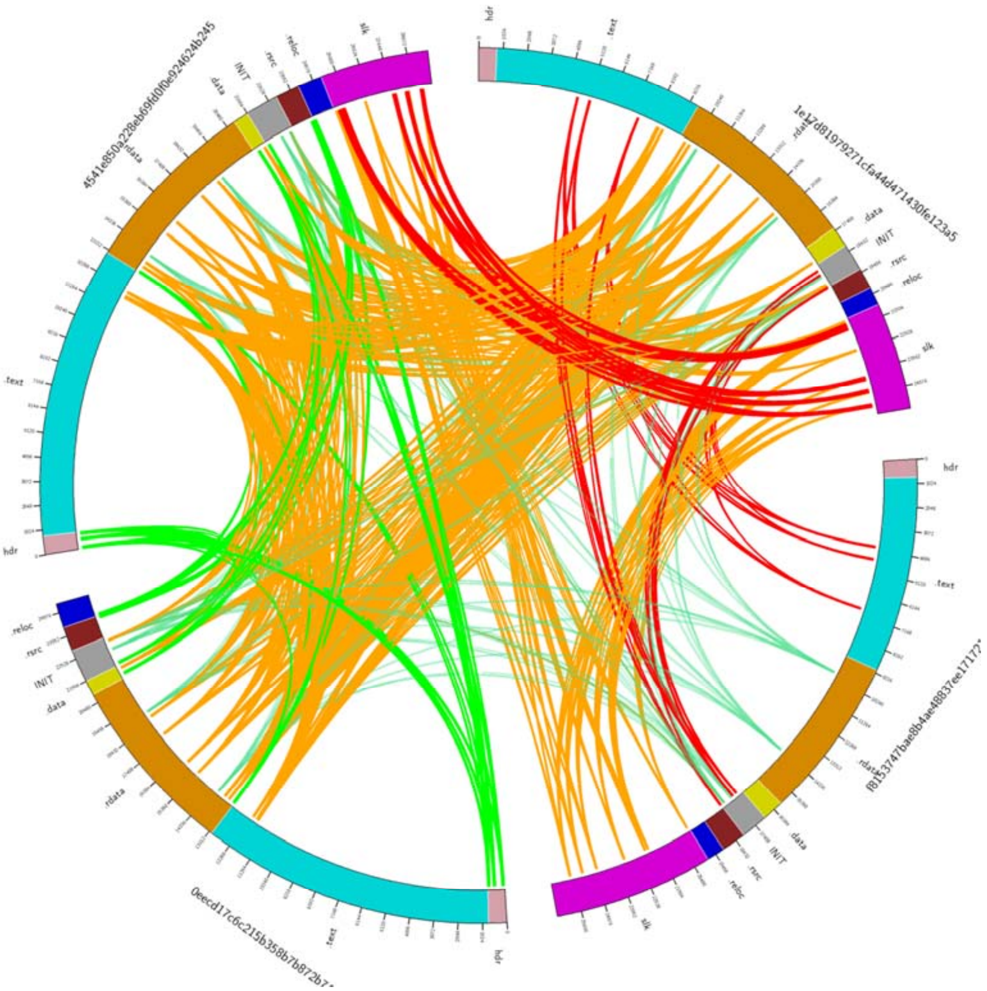


Figure 14: A Visualization of File Matches

The above application validates the ability of the sparse representation data structures to, at the very least, assist in triage for provenance studies. By identifying code-clones among the artifacts can be comprehended at a glance and this can assist in how the analysis may proceed in fundamental and practical ways.

6.3.3 LCS Application to Malware Family Clustering

This application addresses a fundamental question that is routinely addressed in malware analysis: Given a group of malware samples that are allegedly related, for example a set of files assigned to the *Poison Ivy* family by anti-virus researchers, how confident can we be that they are in fact related? We would like to validate or reject the hypothesis of common identity by identifying shared content (code-clones) in the artifacts or identifying particular content elements that may be distinctive to individual artifacts and use these as critical features to improve the assumptions of family assignments. This activity improves what is known about the artifacts, as many are assigned family identity by malware researchers working on a large volume with limited resources and using a variety of heuristic methods (not to mention that malware has a tendency to be deceptive). The ideal ground truth that the community works toward is a high confidence inference of the code identity, provenance, authorship, and development history.

In the following we illustrate an application of sparse representation to vectorize malware for cluster analysis. Here we show a method that could be used to assess a confidence level or suggest improved assignments, thus leading to overall improvements in the quality of malware reference data. This clustering method is based entirely on LCS features in the corpus which can be identified in time linear in the corpus size. This idea of using the suffix tree data structure to assess family assignments is worthy of further consideration as it may lead to a more systematic and formal concept analysis framework such as that developed in Ferre [Ferre 2007].

Below we are able to report that clustering based on LCS determined by our sparse representation approach was consistent with five years of analyst curation and efforts to assign individual *Poison Ivy* files into versions. Further the clustering provided additional visibility into subgroups of the *Poison Ivy* family. In addition the method was able to resolve the identity of a set of corrupted files for which standard techniques failed, thus indicating the LCS measure to have robustness properties.

In this study on *Poison Ivy* we take a sample of 143 files from a malware set that had been extensively studied and assigned to the *Poison Ivy* family by analysts. Further, analysts have attributed each artifact with a version tag based on a static signature. Using this data we create the suffix tree for the set. We then traverse the tree to identify all strings satisfying the following criteria: found in three or more distinct files, have a length that exceeds 600 bytes, and have Shannon entropy exceeding 1.5 (log base 10). These strings then become a set of features for which we can derive a binary attribute matrix linking files to features. In the image below, Figure 15, the attribute matrix is viewed by collapsing features into feature classes that share the same file inclusion pattern (i.e., each column [class] is a distinct cover set of 143 files). The image demonstrates how features and files are related and generally shows that most features occur densely across the sample indicating substantial elements of shared code adding confidence to the family assignment in general.

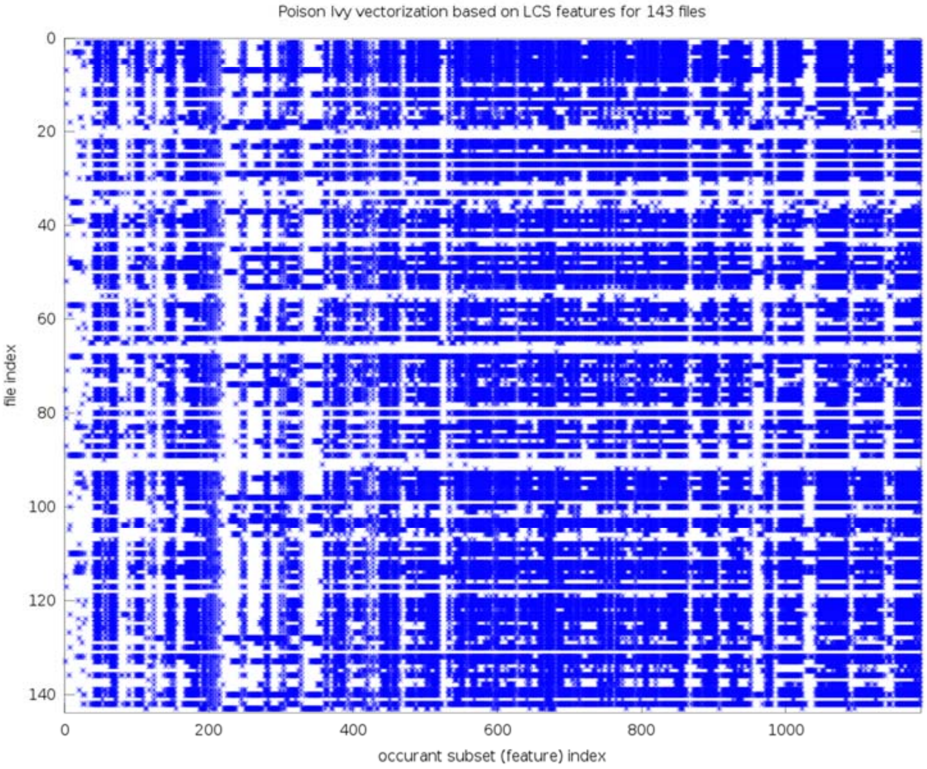


Figure 15: Poison Ivy Vectorization Based on LCS Features

We are able to identify 1,200 distinct cover sets (of the 143 files) which share at least one long interesting string in common. This vectorization process provides a means to investigate the malware family assignment and determine structures within the family based on long common substrings (LCS) alone.

The problem of clustering the rows of the binary attribute matrix is a well-known problem within bio-science with heuristic solutions including hierarchical cluster analysis, maximum parsimony, maximum likelihood [Felsenstein 1981], clustal-W [Thompson 1994], and X-clustering [Dress 2004]. These techniques, as well as understanding their limitations, are in common use in bioinformatics, where they are used to detect similar expression of gene patterns from hybridization arrays and organize taxonomies of species into phylogenetic trees.

Using hierarchical clustering we are able to obtain the following dendogram (Figure 16). Leaves of the tree are annotated with file-id and family tag (obtained from applying analyst developed static signatures compiled over a multi-year period).

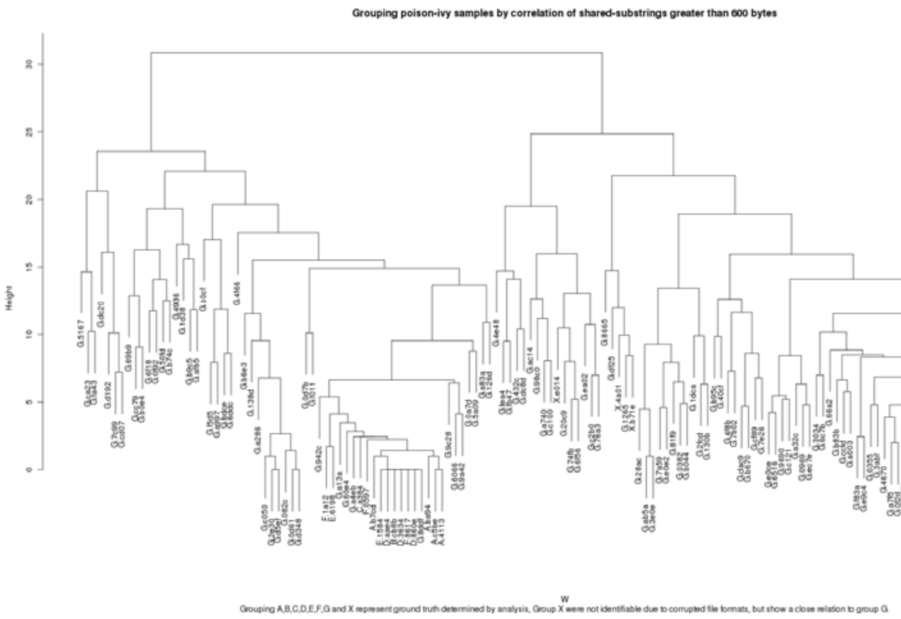


Figure 16: Dendogram Using Hierarchical Clustering

The clustering based on long common substrings is consistent with several of the categories of known types within the *Poison Ivy* family. The hierarchical clustering is able to resolve sub-groupings within the categories already known. In addition several of the corrupted files (group X) were identified as belonging to group G. We were also able to detect a possible misclassified data point.

To indicate the performance of this type of analysis, consider that on a workstation it may take a few hours to complete all of this work including construction of suffix tree, traversal of suffix tree, selection of features, feature reduction, vectorization of malware, computing pairwise correlation, and computing hierarchical clustering. Given the short amount of time this analysis requires (with the use of sparse representation data structures) to corroborate five years of analysis tagging work, we feel that it may have immediate impacts on the problem of malware classification by assessing confidence in existing malware classification.

Assessing the quality of a family assignment may seem like a secondary concern given the more immediate concerns of malware threats; however it is a necessary problem for improving malware reference data sets, and is shown above to be practical. At present what is known about malware families often requires validation, as preliminary knowledge is often derived from automated tools or heuristic methods, and confidence can be improved by these tools designed to assist the human curation process.

6.4 Challenges, Open Problems, and Future Work

In this chapter we have highlighted several outstanding problem and techniques necessary to identify provenance or similarities during the malware triage and analysis. We discussed our approach via sparse representation data structures and how we use them to tailor applications to analyze malware problems. We have shown three distinct applications on actual malware data and how they may impact the analysis and our current state of knowledge and confidence therein.

In this section we identify several of the open and challenging problems having both technical and analytic aspects. For each problem we discuss what our research has found and where it may lead in the future.

Data structures and algorithms

The largest outstanding technical problem is the problem of scale. While we were able to externalize the suffix tree algorithm and explore vastly larger data sets than are available to a standard RAM model algorithm, we would like to scale our inputs to Peta-byte size range in future work. We are currently able to construct suffix trees for thousands to tens of thousands of malware artifacts on a single desktop. However, at our maximum sizes we must deal with large constant factors (associated with the linear runtime) arising with I/O and this slows down the construction algorithm in a noticeable way. In order to incorporate millions of files we have determined several methods worthy of investigation. The first and most basic is focused on optimization, the second on distribution, and third approach is based on input normalization.

Optimizations can be made to incorporate considerations about hardware, with basic steps to optimize code for footprint size. Further we may investigate memory management strategies with the aim to minimize disk reads and writes. For suffix trees the optimal memory management strategy appears to be a very deep and open problem; our work has suggested that determining the optimal strategy has elements of data-dependence. We have developed a technique that could be used to study memory management strategies on data by recording a construction-page-call-sequence which can be analyzed for quantities such as call *frequency*, *inter-arrival-times*, and even tree-topology considerations. Further, this can be done as a separate offline process to determine how best to organize the external-memory model. It may even be possible to switch or dynamically update the memory management strategy to react to data. One step we have already made in this direction is that we have externalized the sub-word tree construction technique into three separate components associated with dramatically different memory usage patterns. Optimal memory management techniques for these data structures remain an open problem.

The possibility of adding parallelism to the tree construction is another; at the very least we envision the possibility of establishing separate threads for each component of externalized data and possibly factoring and distribution of the tree. Another technique implements distributed and parallel solutions by implementing builds and searches independently [Gurtowski 2010], while another possibility cited as generalized suffix trees is to build in parallel independent trees and merge trees as a finishing process. Toward this end we have codified a suffix-tree merge procedure and plan to implement this in future work.

Another possible approach to high throughput performance is *input normalization*. This may involve string homomorphism (working on larger alphabets such as k-bytes) or reductive mappings (removing redundant strings before processing) by identifying redundancy either with alternate data structures or the tree itself. Another avenue of exploration could be based on alteration of inputs by compression [Navarro 2008].

Further another technical idea worthy of exploration and which may improve the runtime characteristics associated with the tree traversal is to further augment the suffix-tree data structure to accelerate computations by using memorization. This could either be done by increasing the tree data-structure footprint, or possibly by additional improvements to data representations—for example, exploring the use of zero-suppressed binary decision diagrams (ZDDs) to encode cover sets compactly for the suffix tree [Minato 1993].

Improving the runtime constants associated with construction and search remains a critical challenge for which we believe there are both fundamental and simple improvements that could be made. Because these methods (which are inherently linear time methods) have greatest impact for large scale data sets (involving millions of files) the challenge of exploring the limits of performance is a worthy problem with practical applications and impact to the area of malware analysis.

Analysis

While we have presented three distinct applications for the sparse representation methods, there are many more potential applications worthy of exploration. We indicate a few additional applications that could potentially provide additional visibility into data, allowing for both exploration and precise measures for provenance and relation discovery:

Code Flow: In the first application which searches an artifact for any code-clones in a known data set, we mentioned our plans to build and develop reference data sets by leveraging existing CERT data. Using these reference data sets we envision a procedure that given an unknown would check for code-clones in the standard set of families, thus indicating code flow from family to family—a very interesting possibility that has been alleged in several known families such as the relation between *Zbot* and *Ispy*. We plan to construct these tools to identify and quantify code flow in existing malware data sets as future work.

Code Normalization: By creating a data set of system libraries and a potentially large set of third-party software we may re-implement the first application (artifact vs. reference data set) to identify linked code or code that is cloned from system or third-party software and thus winnowing less interesting code from the triage queue and promoting the rapid identity of specific malcode.

Statistical Baseline: An important fundamental problem (worthy of consideration) arose while studying the relations between *Stuxnet* and *Duqu*, and would resolve questions of how many code-clones would we expect for non-related malware families vs. how many would we expect for related families. We anticipate that the question will be resolved quickly by statistics but produce non-standard models. Currently our maps of string identity can be used by an analyst to determine relations but we envision a possibility that a statistical baseline models may allow machine processing of these questions to produce meaningful results.

Formal concept analysis of artifacts: To further the descriptive language idea presented in our second application a worthy possibility is to address these problems using formal concept analysis (FCA) techniques [Ferre 2007] which may provide effective means to reason and summarize code relations in sets—and may be particularly applicable in malware given the murky family identities of code and possible heavy cut-and-paste code flows. These methods could also apply to the clustering problem presented in application three.

String similarity: In addition to the LCS measure presented in the third application there is previous work with developing suffix trees for application to general approximate string matching, with the most significant example for code-clones being search for p-strings [Baker 1993]. P-strings are parameterized strings that although having identical aspects are noisy at various locations (think of the way multiple function calls to a C function may look in source code). Effective tools to match P-strings may have significant impact on understanding the static binary. Another concept of string similarity are the affine alignment algorithms from biology. While these techniques are clearly useful for binaries (for variable offsets in section or headers, shuffle obfuscation, and variable data-frame size) the idea that they could be performed on sparse representation structures including the suffix tree is an open problem worthy of consideration.

6.5 Collaborations

This project involved SEI members Will Casey as principal investigator as well as Jeffrey Havrilla and Charles Hines. Aaron Shelmire and Leigh Metcalf helped in the development of analytics in fundamental and significant ways. In addition David French helped us to identify data sets for study and cross validation of methods.

In addition we worked with external collaborators Ravi Sachidanandam and James Gurtowski at Mount Sinai School of Medicine, New York City, who helped us get started with suffix-tree construction and usage. They utilize suffix-tree searching in parallel to handle large number of simultaneous searches on deep-sequence bio-informatic databases, and we benchmarked their applications and confirmed the runtime characterizations of scale for both bio-informatic data and software artifacts.

6.6 References

[Anderson 2008]
Anderson, Ross. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Press, 2008.

[Arge 1996]
Arge, Lars. “The Buffer Tree: A New Technique for Optimal I/O Algorithms.” *Basic Research in Computer Science* RS-96-28 (August 1996).

[Baker 1993]
Baker, Brenda. “A Theory of Parameterized Pattern Matching: Algorithms and Applications,” 71-80. *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*. 1993.

[Bayer 2006]
Bayer, U., Moser, A., Kruegel, C., & Kirda, E. “Dynamic Analysis of Malicious Code.” *Journal in Computer Virology 2*, 1 (2006): 67–77.

[Bayer 2009]
Bayer, U., Habibi, I., Balzarotti, D., Kirda, E., & Kruegel, C., “A View on Current Malware Behaviours,” 8. *Proceedings of the 2nd USENIX Conference on Large-scale Exploits and Emergent Threats: Botnets, Spyware, Worms, and More*. USENIX Association, 2009.

[Bayer 2009a]
Bayer, U., Comparetti, P., Hlauschek, C., Kruegel, C., & Kirda, E. “Scalable, Behavior-Based Malware Clustering,” In *Proceedings of Symposium on Network and Distributed System Security (NDSS)*. 2009.

[Bieganski 1994]

Bieganski P., Riedl J., Carlis J., & Retzel E. “Generalized Suffix Trees for Biological Sequence Data: Applications and Implementation,” 35-44. *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*. 1994.

[Casey 2010]

Casey, W., Cohen, C., French, D., Hines, C., Havrilla, J., & Kinder, R. *Diversity Characteristics in the Zeus Family of Malware* (CMU/SEI-2010-SR-030). Software Engineering Institute, Carnegie Mellon University, 2010.

[Casey 2010a]

Casey, W., Cohen, C., French, D., Hines, C., & Havrilla, J. *Application of Code Comparison Techniques Characterizing the Aliser Malware Family* (CMU/SEI-2010-SR-031). Software Engineering Institute, Carnegie Mellon University, 2010.

[Cordy 2011]

Cordy, James R. and Roy, Chanchal K. “Efficient Checking for Open Source Code Clones in Software Systems,” 217-218 *Proceedings of IEEE 19th International Conference on Program Comprehension (ICPC)*. 2011.

[Denning 1986]

Denning, D. “An Intrusion Detection Model.” *Proceedings of the Seventh IEEE Symposium on Security and Privacy*. 1986.

[Dress 2004]

Dress, A.W.M. and Huson, D.H. “Constructing Splits Graphs.” *IEEE/ACM Transactions on Computational Biology and Bioinformatics 1*, 3 (July-September 2004): 109-115.

[Falliere 2011]

Falliere, N., Murchu, L.O., & Chien, E. *W32.Stuxnet Dossier*. Symantec Security Response, February 2011.
http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf

[Felsenstein 1981]

Felsenstein, J. “Evolutionary Trees From DNA Sequences: A Maximum Likelihood Approach.” *Journal of Molecular Evolution 17*, 6 (1981): 368–376.

[Ferre 2007]

Ferre, S., “The Efficient Computation of Complete and Concise Substring Scales with Suffix Trees,” 98-113. *Proceedings of the 5th International Conference on Formal Concept Analysis.* 2007.

[Godfrey 2011]

Godfrey M., German D., Davies J., & Hindle A. “Determining the Provenance of Software Artifacts,” 65-66. *Proceedings of the 5th International Workshop on Software Clones IWSC.* ACM, 2011.

[Gurtowski 2010]

Gurtowski, J, Cancio, A, Shah, H., Levovitz, C., Homann, George R., & Sachidanandam, R. "Geoseq: a Tool for Dissecting Deep-Sequencing Datasets," *BMC Bioinformatics 2010, 11*: 506.

[Gusfield 1997]

Gusfield, Dan. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997

[Homann 2009]

Homann, R., Fleer, D., Giegerich, R., & Rehmsmeier, M. "mkESA: Enhanced Suffix Array Construction Tool." *Bioinformatics 25,* 8 (2009):1084-1085.

[Hu 2009]

Hu, X., Chiueh, T., & Shin, K. "Large-Scale Malware Indexing Using Function-Call Graphs," 611-620. *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 2009.

[Jang 2010]

Jang, Jiyong, Brumley, David, & Venkataraman, Shobha. *BitShred: Fast, Scalable Malware Triage* (CMU-CyLab-10-022). Carnegie Mellon University, November 5, 2010.

[Minato 1993]

Minato, Shin-ichi. "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems," 272-277. *Proceedings of the 30th International IEEE Design Automation Conference*. ACM, 1993

[MITRE 2010]

Science of Cyber-Security (JSR-10-102). MITRE, November 2010

[Morales 2011]

Morales, Jose Andre, Main, Michael, Lou, Weiliang, Xu, Shouhuai, & Sandhu Ravi. **"**Building Malware Infection Trees," 50-57. *Proceedings of the 6th IEEE International Conference on Malicious and Unwanted Software (Malware 2011),* Fajardo, Puerto Rico, October 18-19 2011.

[Navarro 2008]

Navarro, G. and Mäkinen, V. "Dynamic Entropy-Compressed Sequences and Full-Text Indexes." *ACM Transactions on Algorithms 4,* 3 (2008): article 32.

[Pleszkoch 2008]

Pleszkoch, Mark G., Linger, Richard C., & Hevner, Alan R. "Introducing Function Extraction into Software Testing." *The Data Base for Advances in Information Systems: Special Issue on Software Systems Testing*. ACM, 2008.

[Rhee 2011]

Rhee, J., Lin, Z., Xu, D. "Characterizing Kernel Malware Behavior With Kernel Data Access Patterns," 207-216. *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (ASIACCS '11)*. ACM 2011.

[Roussev 2007]
Roussev, V, Richard, G., & Marziale, L. "Multi-Resolution Similarity Hashing." *Digital Investigations 4* (August 2007):S105-S113

[Roussev 2011]
Roussev, V. "An Evaluation of Forensic Similarity Hashes." *Digital Investigations 8* (August 2011):S34-S41

[Shabtai 2011]
Shabtai, Asaf, Menahem, Eitan, & Elovici,Yuval. "F-Sign: Automatic, Function-Based Signature Generation for Malware." *IEEE Transactions on Systems, Man, and Cybernetics—Part C: Applications and Reviews 41*, 4 (July 2011): 494-508.

[Shannon 1948]
Shannon, Claude E. "A Mathematical Theory of Communication." *Bell System Technical Journal 27* (1948): 379–423, 623–656

[Song 2008]
Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M.G., Liang, Z., Newsome, J., Poosankam, P., & Saxena. P. "BitBlaze: A New Approach to Computer Security Via Binary Analysis." *Proceedings of the 4th International Conference on Information Systems Security*, Hyderabad, India, Dec. 2008. Springer-Verlag Berlin Heidelberg 2008.

[Symantec 2011]
W32.Duqu The Precursor to the Next Stuxnet. Symantec Security Response, October 2011 http://www.symantec.com/connect/w32_duqu_precursor_next_stuxnet

[Thompson 1994]
Thompson, J.D., Higgins, H.G., & Gibson, T.J. "CLUSTAL W: Improving the Sensitivity of Progressive Multiple Sequence Alignment Through Sequence Weighting, Positions-Specific Gap Penalties and Weight Matrix Choice." *Nucleic Acids Research 22* (1994):4673-4680.

[Ukkonen 1992]
Ukkonen, E. "Constructing Suffix Trees On-Line in Linear Time," 484-492. *Proceedings of Information Processing 92*, 1, IFIP Transactions A-12. Elsevier, 1992

[Ukkonen 1995]
Ukkonen, E. "On-Line Constructing of Suffix Trees," 249-260. *Algorithmica 14.* 1995

[Willems 2007]
Willems, C., Holz, T., & Freiling, F. "CWSandbox: Towards Automated Dynamic Binary Analysis." *IEEE Security and Privacy 5*, 2 (March 2007): 32-39.

7 Learning a Portfolio-Based Checker for Provenance-Similarity of Binaries

Sagar Chaki, Cory Cohen, Arie Gurfinkel

7.1 Purpose

Software is an integral part of our technology driven lives. In many situations, the software we use is obtained ultimately from unknown and untrusted sources. Therefore, tools and techniques that enable the understanding and prediction of a program's provenance are of vital importance. This is particularly true for the Department of Defense (DoD), whose cyber infrastructure is not only complex and developed in a decentralized manner, but also the target of malicious adversaries. The DoD is acutely aware of this problem and looking for effective solutions. For example, DARPA's recent Cyber Genome Program (DARPA-BAA-10-36)[20] which specifically requested new research that would "identify the lineage and provenance of digital artifacts" demonstrates the Department of Defense's interest in the binary similarity and provenance in particular.

Binary Similarity. While program analysis tools come in many flavors, those that are able to handle software in its binary form are particularly valuable for at least two reasons. First, binaries represent the "ground truth" of any software. Therefore, binary analysis provides results and insights that reflect the software's true nature. Second, the vast majority of commodity software is only available in binary format. The main topic of this LENS is "binary similarity", a form of binary analysis that involves understanding, measuring, and leveraging the degree to which two binaries are (dis)similar in terms of their syntax and structure.

Binary similarity is an active area of research with practical applications ranging from code clone detection [Quinlan 2009] and software birthmarking [Choi 2007] to malware detection [Cohen 2009] and software virology [Walenstein 2007]. Binary similarity capabilities enabling the association of intruder activity from multiple classified intrusions has continued to be a topic of interest amongst our intelligence sponsors, the DHS, and the DoD. Automated observations about code reuse, whole program duplication, and shared exploit techniques can assist in attribution which is often a key objective for national defense and law enforcement.

Binary similarity is an inherently complex problem, and several solutions have been proposed. However, none is known to be, nor likely to be, precise (i.e., having low false positives and negatives) and efficient across a wide range of programs. We believe that a more effective approach is to combine multiple such solutions into a "portfolio-based similarity checker."

Provenance-Similarity. An appropriate notion of "similarity" is critical in scoping the binary similarity problem to be both practically relevant and effectively solvable. For this LENS, we proposed and used the notion of "provenance-similarity" of binaries. The word "provenance" broadly means "the history of ownership of a valued object".[21] Thus, the provenance of an object

[20] https://www.fbo.gov/index?s=opportunity&mode=form&id=c34caee99a41eb14d4ca81949d4f2fde

[21] http://www.merriam-webster.com/dictionary/provenance

includes its original source, as well as the processes that it undergoes in its lifetime. The term was originally mostly used for works of art, but is now used in similar senses in a wide range of fields, including science and computing.[22]

Specifically, the U.S. Department of Homeland Security, Science and Technology Directorate (DHS S&T) [DHS09] states: *"Provenance" refers to the chain of successive custody—including sources and operations of computer-related resources such as hardware, software, documents, databases, data, and other entities. Provenance includes "pedigree," which relates to the total directed graph of historical dependencies. It also includes "tracking" which refers to the maintenance of distribution and usage information that enables determination of where resources went and how they may have been used.*

For the purposes of this LENS, "provenance-similarity" between two binaries implies similarity in not only the source code from which they were derived, but also the compilers (including code obfuscators and other transformers) used to generate them. Specifically, we define two binaries B_1 and B_2 to be "provenance-similar" if they have been compiled from the same (or very similar) source code with the same (or similar) compilers. Note that we exclude situations where the source code of B_1 and B_2 are very different, such as when B_1 is derived from B_2 via a major patch, or when B_1 and B_2 are radically different implementations of the same functionality, e.g., merge-sort and bubble-sort. We also exclude cases where the compilers involved are very different (e.g., Visual C++ and gcc). These problems are beyond the scope of this LENS.

Binary Similarity Applications. The ability to detect provenance-similarity has applications in several practically relevant domains. We discuss here two of the most notable:

1. ***Software Virology:*** Many programs (notably, malware and different versions and builds of a software package) are simply minor variants of old ones compiled with a different compiler. In a recent paper, Walenstein et al. sum up the situation crisply as follows [Walenstein 2007]:

   "The majority of malicious software floating around are variations of already well-known malware. Consider the data from Microsoft's "Microsoft Security Intelligence Report: January – June 2006" [Braverman 2006]. According to their data, there were 97,924 variants of malware found within the first half of 2006. That is over 22 different varieties per hour. The report does not say precisely what Microsoft considers a variant, but clearly simple byte-level differences are not enough for them to label two files as distinct variants. For instance, of the 5,706 unique files that were classified as Win32/Rbot, only 3,320 distinct variants were recognized. The variants Microsoft list [sic] are not just trivially different files.

   Clearly, these 97,924 distinct bundles of malevolence cannot all *be completely different malware built lovingly from scratch. Instead, the vast majority are merely modifications of previous software—new versions of programs. According to Symantec [Symantec 2006] and Microsoft [Braverman 2006] typically only a few hundred families are typical in any half-year period. And the Microsoft numbers paint a striking picture. Figure 2.1 shows a pie chart of the numbers of variants found. It breaks down the distribution into the 7 most common families, the 8-25 ranked families, and then the rest. The top 7 families account for more than 50% of all variants found. The top 25 families account for over 75%. Thus it is a*

[22] http://en.wikipedia.org/wiki/Provenance

> *solid bet that any new malicious program found in the wild is a variation of some previous program. The lion's share of work in handling the flood of new programs would be done if one could recognize even only the topmost 25 families automatically."*

Independent research by Cory Cohen—one of this LENS's leads—and his group at the Software Engineering Institute's CERT program support these observations. Clearly, the ability to check binary similarity enables effective techniques for cataloging and understanding the inter-relationships between such binaries.

2. ***Functionality Detection:*** Another application is in detecting if a binary contains specific functionality. For example, it is important to detect if a binary contains fragments compiled from specific source code. This is true especially in two types of situations. In one case, the presence of a specific binary fragment constitutes a violation of intellectual property law. Several instances of such violations have been reported already.[23] In another case, a specific binary fragment indicates the possibility of malware. Typical examples are infected files or devices (such as hard drives[24], USB drives[25], cell phones[26], and mp3 players[27]). The ubiquity of such mobile devices makes them an ideal and potent vector[28] for the spread of malicious software.

Existing Approaches. A number of different approaches have been proposed to implement Binary Similarity Checkers (BSCs). The most notable ones are based on signatures, feature vectors, or mathematical descriptions of the semantic meaning of binaries. None of these approaches is known to be efficient and robust (i.e., having low false positives and negatives) across a wide range of binaries in their application contexts. Therefore, we believe that no single approach will be efficient and robust for provenance-similarity as well. It is also unclear to what extent these BSCs complement each other. For example, an appropriate combination of a feature-based BSC and a semantic BSC may be more efficient and robust than either BSC in isolation. In general, we believe that a BSC for provenance-similarity derived from a portfolio of techniques is more effective than any technique in isolation

Supervised Machine Learning. In this LENS, we demonstrated that supervised learning is a promising candidate for constructing our desired portfolio-based BSC. Supervised learning excels at solving problems where: 1) closed form analytical solutions are hard to develop – this is certainly true for binary similarity, and 2) a solver can be "learned" using a training set composed of positive and negative samples – creating such a training set for binary similarity is feasible, since there are many known instances of similar and dissimilar binaries.

[23] http://gpl-violations.org

[24] http://www.zdnet.com/blog/hardware/malware-found-on-new-hard-drives/928

[25] http://www.zdnet.com/blog/security/malware-infected-usb-drives-distributed-at-security-conference/1173

[26] http://www.sophos.com/blogs/gc/g/2010/06/02/samsung-wave-ships-malwareinfected-memory-card

[27] http://www.infoworld.com/d/security-central/worm-eats-its-way-mcdonalds-mp3-player-promotion-378

[28] http://newsinfo.inquirer.net/breakingnews/infotech/view/20080423-132157/Experts-warn-vs-malware-spreading-through-removable-drives

Goals of this LENS. This LENS achieved three broad objectives:

1. **Developed a portfolio-based binary provenance-similarity checker via supervised learning.** Developing such a portfolio is challenging from several perspectives. First, the relative strengths and weaknesses of the constituent BSCs are difficult to characterize and quantify. Moreover, the space of BSCs is in constant flux, as new similarity checkers emerge and existing ones are refined and improved. Finally, as new software is developed, the input space of the portfolio evolves. Therefore, a static portfolio-based BSC is almost certainly doomed to fail. Indeed, we believe that an effective portfolio must be able to "learn" and adapt to the changing space of available binaries and BSC technology. This made machine learning an attractive choice for developing such a portfolio.
2. **Developed an effective benchmark for provenance-similarity for use by the wider community.** We believe that publicly available benchmarks are critical for advancements in binary similarity technology. As part of this LENS, we created and disseminated such a benchmark.
3. **Advanced the state-of-the art in binary similarity detection.** We believe that the development of the concept of provenance-similarity, and efficient provenance-similarity checkers, will be a major breakthrough in the field of binary similarity. We contributed to this process by publishing the outcomes of our research at a top-level venue in machine learning and knowledge discovery [Chaki 2011].

In summary, we believe that this LENS improved the state-of-the-art in binary similarity detection, aided the SEI in its mission and vision of "leading the world to a software enriched society," improved the visibility of the SEI as a center of technical excellence among the wider research and practitioner community, and fostered collaborative research within our organization.

7.2 Background

This LENS emerged from the convergence of several compelling circumstances. The interest in the proposed line of work originated in the course of collaborative research and technical discussions between members of the SEI's Research, Technology, and System Solutions (RTSS) program and Networked Systems Survivability (NSS) program. The LENS was led by a core group of these members, and supported by the others. It strengthened rapport and working relationship between the two SEI programs. While we discussed several possible LENS topics, a number of factors led to this particular selection.

First, the proposed LENS addresses some of the most crucial problems facing the nation, and the DoD. For example, in its February 2005 report titled *Cyber Security: A Crisis of Prioritization*, the President's Information Technology Advisory Committee (PITAC) warns that the "IT infrastructure is highly vulnerable to premeditated attacks with potentially catastrophic effects," and thus "is a prime target for cyber terrorism as well as criminal acts [PITAC Report]." The research proposed in this LENS is of direct relevance to several of the cyber security research priorities enumerated by the PITAC report, notably:

1. Technologies to efficiently and economically verify that computer code does not contain exploitable features that are not documented or desired (page 39)
2. Securing a system that is co-operated and/or co-owned by an adversary (page 40)

3. Identifying the origin of cyber attacks, including traceback of network traffic (page 43)
4. Tools and protocols to search massive data stores for specific information and indicators, possibly while the data stores are in use (page 43)
5. Developing security metrics and benchmarks (page 45)
6. Tools to assess vulnerability, including source code scanning (page 45)

Second, as mentioned earlier, we believe that this research addresses problems that are of vital interest to some of the SEI's most important customers and stakeholders. This LENS proposal complements work being performed under contract for intelligence agency sponsors over the last several years. These sponsors continue to be very interested in advancing the state of the art capabilities for binary code comparison.

Third, a number of research groups are investigating various problems in the area of binary similarity [Quinlan 2009, Walenstein 2007]. Publications in this area (including very recent ones) indicate that there is interest in the academic community and their funding agencies. We believe that binary similarity is one of the crucial and emerging research areas where the SEI must establish core competence in order to continue to fulfill its mission and vision. We view this LENS as one of the most effective mechanism for moving toward this goal.

Finally, the participants of this LENS have prior and ongoing research experience in related areas, which increases our chances of success. Specifically, Dr. Sagar Chaki and Dr. Arie Gurfinkel are actively involved in foundational and applied research, as well as peer-reviewed publications in the areas of formal specification and verification of systems. In particular, their expertise in software model checking [Gurfinkel 2008], static analysis [Gurfinkel 2010] and learning algorithms [Chaki 2010] are of direct relevance to the proposed LENS. Cory Cohen leads the malicious code research and development team in the NSS program, and has been engaging the malware analysis community to elicit practical applications and requirements for the effective transition of the results of this LENS. His expertise and that of his team ensured that this work provides tangible benefits to malware analysts and is fully informed of the peculiarities of the malware problem domain. Cohen's team is uniquely suited to engage to broader malicious code analysis community to report our work and solicit feedback.

7.3 Approach

Recall that we define two binaries to be "provenance-similar" if they have been compiled from the same (or very similar) source code with the same (or similar) compilers. For example, B_1 is provenance-similar to B_2 if:

1. the source code of B_1 is derived from that of B_2 by a minor patch;
2. the source code of B_1 is derived from that of B_2 by renaming certain variables;
3. B_1 and B_2 are compiled from the same code by different members of the same family of compilers, e.g., Visual C++ 2005 and 2008;
4. B_1 and B_2 are compiled from the same code by different versions of the same compiler, e.g., `gcc 4.3` and `gcc 4.4`;
5. B_1 and B_2 are compiled from the same code by the same compiler with different optimization levels, e.g., `gcc` and `gcc -O2`;

At a high-level, a binary similarity checker (BSC) is a "black box" that accepts a pair of binaries B_1 and B_2 as input, and outputs "yes" if B_1 and B_2 are similar, and "no" otherwise. A number of different approaches have been proposed to implements BSCs, notably:

1. **Signature-Based:** There are several known ways of extracting signatures from binaries. For example, one approach works by: extracting a sequence of bytes that correspond to the instructions in the binary; and applying a strong cryptographic hash function, such as MD5, to the resulting byte sequence [Cohen 2009]. In fact, anti-virus tools work by extracting such signatures from binaries and comparing them against a corpus of signatures extracted from known viruses. Unfortunately, signature-based schemes have limited effectiveness, and are foiled by even minor syntactic alterations in binaries. This effect is heightened by the use of hashes, such as MD5, that are designed to be sensitive to syntactic differences. Consequently, signature-based schemes suffer from high rates of false positives, or false negatives, or both.
2. **Feature-Vector-Based:** This works by extracting and comparing feature vectors. For example, Walenstein et al. propose the use of *n-grams* to extract a vector of numbers from a target binary [Walenstein 2007]. Each element of this vector corresponds to the number of occurrences of a specific feature – e.g., the instruction sequence (`push,add,xor,pop`) – in the target binary. The similarity of two binaries is derived from the "angle" or "cosine-distance" between the feature vectors extracted from them. Feature-vector based approaches are less sensitive to syntactic differences that signature-based ones. However, their effectiveness depends crucially on the features used. Finding "good" features is non-trivial, and features derived purely from the syntactic structure of binaries suffer from the same limitations as signature-based schemes.
3. **Semantic:** This involves computing and comparing "mathematical" descriptions of the semantic meaning of binaries. For example, Gao et al. find semantics difference between binaries by a comparing the control-flow-graphs (at the function-level) and call-graphs (at the binary level) [Gao 2008]. Nodes of the control-flow graphs correspond to basic blocks (i.e., sequences with a single flow of control) of machine instructions. These nodes are compared in terms of the effects of their execution on the system state. This is the most precise approach, and ideally would be able to show the similarity between two implementations of the same functionality (e.g., merge-sort and bubble-sort). However, it is also the least scalable. It requires the most manual interaction and computationally expensive technology for symbolic and mathematical reasoning (e.g., term-rewriting and theorem proving), which reduces its practical effectiveness.

None of the above BSCs is known to be, nor likely to be, efficient and robust (i.e., having low false positives and negatives) across a wide range of binaries for the specific contexts in which they have been applied. We believe that, in isolation, they will be of limited effectiveness for provenance-similarity as well. Therefore, we developed a portfolio-based checker for provenance-similarity that combines the above (and possibly other) BSCs into one that is more robust and efficient that any individual BSC.

From an external perspective, our target portfolio-based checker is just another BSC. Internally, the portfolio computes the similarity of binaries B_1 and B_2 using each of its constituent BSC techniques $T_1, \ldots, T_n$, and then combines the results $R_1, \ldots, R_n$, in using some "special sauce" to obtain its final "yes/no" answer. Figure 17 shows this process pictorially.

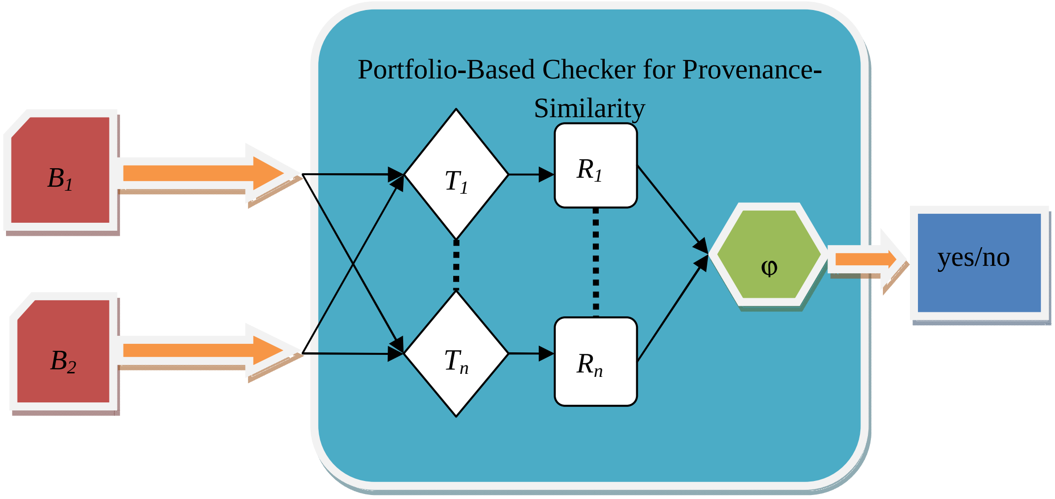


Figure 17: Overview of a Portfolio-Based Checker for Provenance Similarity

The key challenge is of course finding the special sauce, i.e., the right combination φ of $R_1, \ldots, R_n$. We constructed φ using supervised learning. Our approach consisted of the following major steps:

- Learn about the state-of-the-art in supervised learning techniques. Select candidate techniques for experimentation and evaluation. Our choices of learning algorithms were based on decision-trees, random trees, random forests, support vector machines (SVM), and neural networks (e.g., perceptrons).
- Learn about the state-of-the-art in supervised learning tools and platforms. Select candidate tools for experimentation and evaluation. Based on a careful study, our choice was the WEKA (http://www.cs.waikato.ac.nz/ml/weka) platform.
- Construct a benchmark for provenance-similarity using open-source programs. In prior work, we had created a benchmark consisting of provenance-similar "functions." We adapted and extended this benchmark for our needs. The benchmark and associated tools are available online (http://www.contrib.andrew.cmu.edu/~schaki/binsim/index.html).
- Implement attribute extraction schemes. Broadly, we experimented with two categories of attributes: syntactic and semantic. Attribute extraction schemes were implemented on top of the ROSE infrastructure.
- Construct checkers for provenance-similarity using the attribute schemed mentioned above, and the WEKA framework.
- Evaluate these similarity checkers on our benchmark, and identify the best.
- Document the research and release the tools and benchmarks to the research community.

7.4 Collaborations

As mentioned earlier, this LENS was a collaborative effort between the RTSS and NSS programs at the SEI. It was co-lead by Sagar Chaki and Arie Gurfinkel from RTSS, and Cory Cohen from the NSS/CERT. The project also relied on feedback from Cohen's malicious code research and development team at NSS. In particular, we would like to express our thanks to Will Casey, David French, Jeffrey Havrilla, and Charles Hines. We would also like to express our gratitude to the developers of the WEKA and ROSE infrastructures, both of which were critical to the success of this LENS.

7.5 Evaluation Criteria

The deliverables for this project were the following:

- a prototype portfolio-based binary provenance-similarity checker
- a benchmark for binary provenance-similarity
- a research report (leading to a peer-review publication) describing the findings of the work

All deliverables were completed successfully. In addition, the success of this LENS was based on the development of the concept of provenance-similarity, and efficient provenance-similarity checkers. This enriched the broad field of binary similarity, and the research and practitioner community engaged in this area. Finally, the SEI gained expertise in binary similarity and recent advances in machine learning techniques.

7.6 Results

We summarize briefly some of our key results. Details are available in our conference publication [Chaki 2011]. We first evaluated the set of classifiers available as part of WEKA to find out their effectiveness on the provenance-similarity problem. We found five of them to be effective, i.e., they perform better than random guessing. Figure 18 shows the relative accuracy (in terms of a standard metric called the F-measure) of these five effective classifiers. Clearly, RandomForest is the most accurate.

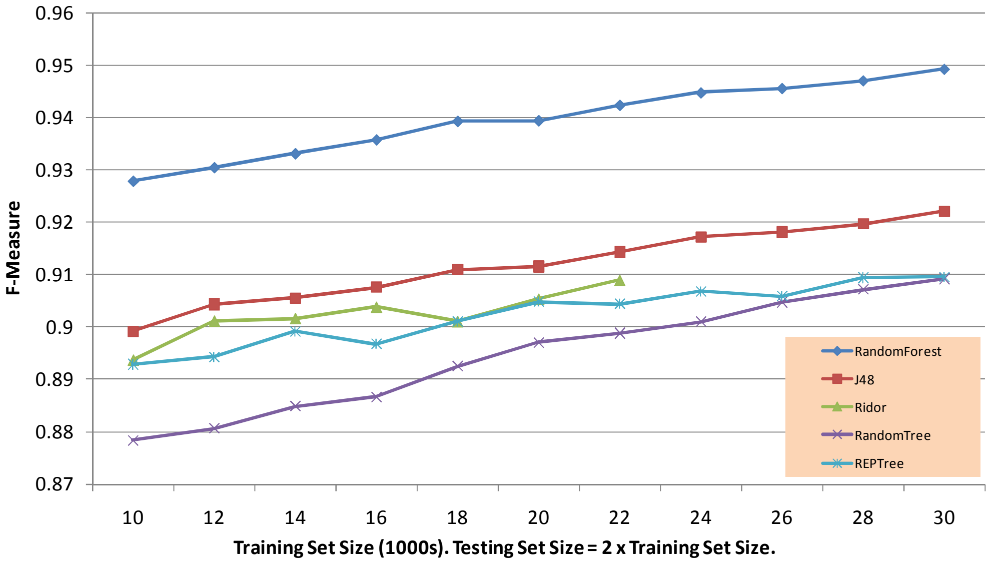


Figure 18. Comparison of Effective Classifiers

RandomForest is an ensemble classifier. It bases its prediction on the predictions made by a number of independently constructed decision trees. Clearly, the accuracy of RandomForest depends on the number of decision trees it uses. We next evaluated the accuracy of RandomForest by varying the number of decision trees. The results are summarized in Figure 19. The accuracy increases up to 40 trees, after which it tapers off.

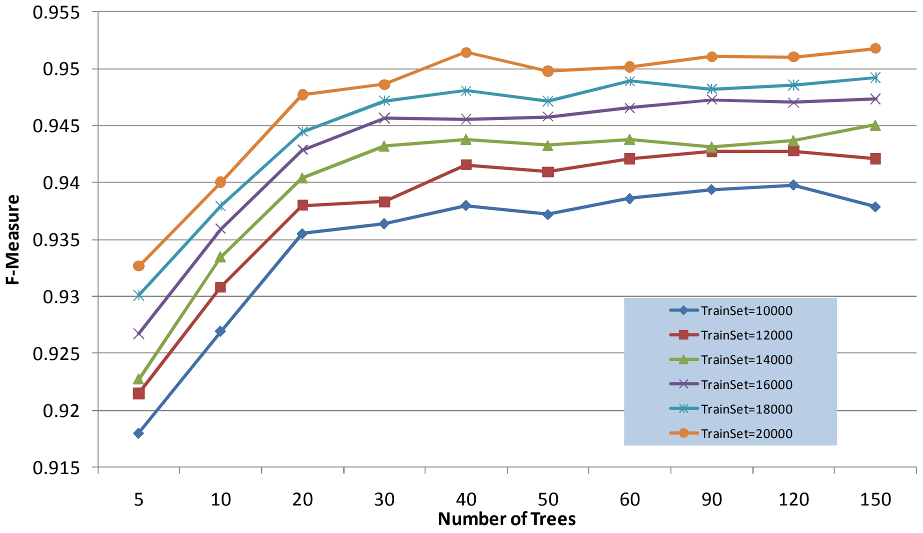


Figure 19. Performance of RandomForest with Varying Number of Trees

Finally, we evaluated semantic attributes against syntactic attributes (known as n-grams), and our approach of combining attributes from two functions into a single attribute vector with an alternate approach based on concatenation. The results are summarized in Figure 20.

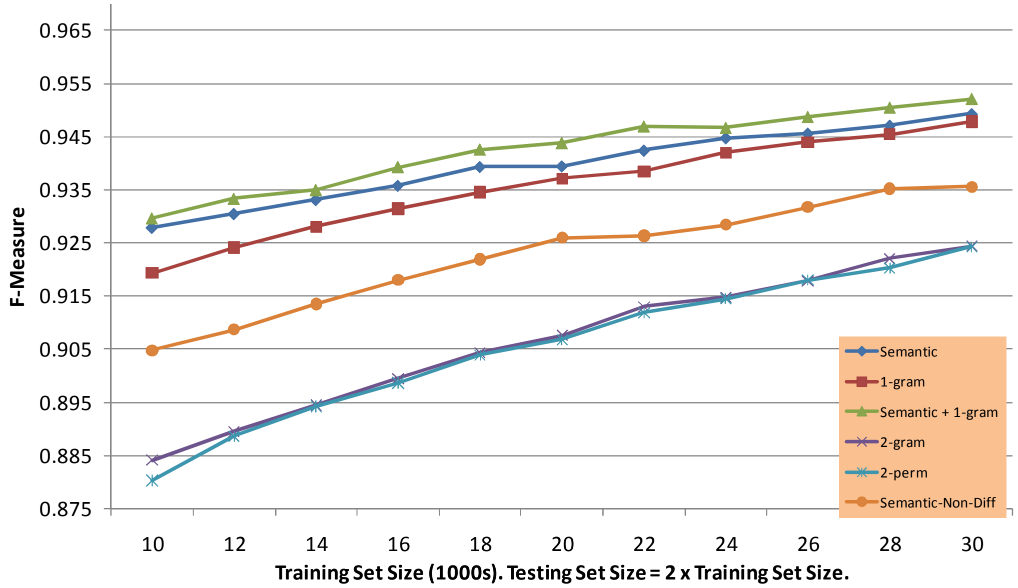


Figure 20. Comparison of Semantic and Syntactic Attributes

We observed that semantic attributes are more accurate than syntactic ones, and our approach for combining attribute vectors is superior to the one based on concatenation. Complete details of our research are available in our conference publication [Chaki 2011].

7.7 Bibliography/References

[Braverman 2006]

Braverman, M, Williams, J, & Mador, Z. *Microsoft Security Intelligence Report: January–June 2006*. http://microsoft.com/downloads/details.aspx?FamilyId=1C443104-5B3F-4C3A-868E-36A553FE2A02.

[Chaki 2010]

Chaki, Sagar & Gurfinkel, Arie. “Automated Assume-Guarantee Reasoning for Omega-Regular Systems and Specifications”, 57-66. *Proceedings of the 2nd NASA Formal Methods Symposium (NFM)*. Washington, D.C., April 13-15, 2010. NASA, 2010.

[Chaki 2011]

Chaki, Sagar, Cohen, Cory, & Gurfinkel, Arie. “Supervised Learning for Provenance-Similarity of Binaries,” 15-23. *Proceedings of the 17th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD)*. August 21–24, 2011, San Diego, California. ACM 2011.

[Choi 2007]

Choi, Seokwoo, Park, Heewan, Lim, Hyun-il, & Han, Taisook: “A Static Birthmark of Binary Executables Based on API Call Structure,” 2-16. *Proceedings of the 12th Asian Computing Science Conference on Advances in Computer Science: Computer and Network Security*. Springer-Verlag Berlin, 2007.

[Cohen 2009]

Cohen, C. & Havrilla, J. “Function Hashing for Malicious Code Analysis.” *CERT Annual Research Report 2009:* 26-29. Software Engineering Institute, Carnegie Mellon University, 2009.

[DHS 2009]

U.S. Department of Homeland Security, Science and Technology Directorate (DHS S&T). “Provenance.” *A Roadmap for Cybersecurity Research*:76. http://www.cyber.st.dhs.gov/docs/DHS-Cybersecurity-Roadmap.pdf.

[Gao 2008]

Gao, Debin, Reiter, Michael K., Xiaodong Song, Dawn. “BinHunt: Automatically Finding Semantic Differences in Binary Programs,” 238-255. *Proceedings of the 10th International Conference on Information and Communications Security*. Springer-Verlag Berlin, 2008.

[Gurfinkel 2008]

Arie Gurfinkel, Sagar Chaki: Combining Predicate and Numeric Abstraction for Software Model Checking. FMCAD 2008: 1-9.

[Gurfinkel 2010]

Gurfinkel, Arie & Chaki, Sagar. “BOXES: A Symbolic Abstract Domain of Boxes,” 287-303. *Proceedings of the 17th International Static Analysis Symposium* (SAS), September 14-16, 2010. Springer-Verlag Berlin, 2010.

[PITAC Report]

President’s Information Technology Advisory Committee (PITAC). *Report to the President on Cyber Security: A Crisis of Prioritization* (February 2005), http://www.nitrd.gov/pitac/reports/20050301_cybersecurity/cybersecurity.pdf.

[Quinlan 2009]

Sæbjørnsen, Andreas, Willcock, Jeremiah, Panas, Thomas, Quinlan, Daniel J., & Su, Zhendong. “Detecting Code Clones in Binary Executables,” 117-128. *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ACM, 2009.

[Symantec 2006]

Symantec. *Symantec Internet Security Threat Report: Trends for January 06 – June 06*. http://www.symantec.com/enterprise/threatreport/index.jsp.

[Walenstein 2007]

Walenstein, A., Venable, M., Hayes, M., Thompson, C., & Lakhotia, A. “Exploiting Similarity Between Variants to Defeat Malware: ‘Vilo’ Method for Comparing and Searching Binary Programs.” In Proceedings of BLACKHAT DC, 2007.

8 Edge-Enabled Tactical Systems: Edge Mission-Oriented Tactical App Generator

Soumya Simanta, Gene Cahill, Ed Morris, Brad Myers

8.1 Purpose

Many people today carry handheld computing devices to support their business, entertainment, and social needs in commercial networks. The Department of Defense (DoD) is increasingly interested in having soldiers carry handheld computing devices to support their mission needs in tactical and hostile environments. Not surprisingly, however, conventional handheld computing devices (such as iPhone or Android smartphones) for commercial networks differ in significant ways from handheld devices for tactical and hostile environments. For example, conventional devices and the software that runs on them do not provide the capabilities and security needed by military devices, nor are they configured to work over DoD tactical networks with severe bandwidth limitations and stringent transmission-security requirements. This article describes research we are conducting at the SEI to 1) create software that allows soldiers to access information on a handheld device and 2) allow soldiers to program the software easily and rapidly to tailor the information for a given mission or situation.

8.2 Background

Imagine a U.S. soldier on patrol, deployed abroad, and preparing to walk into an unfamiliar village that has been friendly toward U.S. soldiers in the past. Many pieces of information would be useful to that soldier in that situation. For example, it would be useful to know who the village elders are and to have pictures to identify them. A list of small gifts that the elders have appreciated in the past may be helpful, along with reports detailing the results of other soldier contact with the villagers. The names and residence locations of any friendly villagers who speak English may also be critical. Again, imagine the same soldier performing a different mission several days later. In this case, his mission goal is to set up a remote outpost to monitor activity along a remote road in the mountains. This time the soldier needs information that allows him to identify people and vehicles that pass, previous timings of the uses of the road, and any suspicious activity that has occurred.

Of course, in addition to having access to the right information for their needs, soldiers need a convenient way to capture new information on tactical handheld devices in a manner consistent with their mission. As these examples illustrate, the information needs of a soldier vary widely with their missions and with time. Soldiers we spoke to at multiple sites were able to identify many potential situations where their unique information needs could be met by a flexible capability on a handheld device that allowed them to control the information they receive and capture information in custom formats. Our goal was to create an app that provided this capability.

Developing the app presented challenges:

- **Developing applications that can support the full range of military missions.** In recent years, soldiers have provided humanitarian assistance to victims of natural disasters in Haiti and countries in Asia, patrolled our country's borders, protected global waterways from piracy, and performed many types of military operations in Iraq and Afghanistan. These missions are sufficiently diverse that a single software solution is not practical. For example, compare the different goals of clearing a route in a combat zone and of delivering humanitarian supplies in a relief effort, or the different information required to protect from improvised explosive device (IED) attacks and to treat a critically ill child. Not only is different information required, but also the rules for sharing it can vary. In a combat environment, security concerns require limiting access, while in a relief mission, the need to collaborate with non-governmental organizations requires that information be shared.
- **Processing large amounts of data available through the rapid computerization and internetworking of various military missions.** For example, the military employs hundreds of unmanned aerial vehicles (UAVs) that generate large amounts of data. There are also increases in the number of sensors, such as auditory, biological, chemical, and nuclear, that are network enabled. All the data generated from these devices makes it hard to pinpoint the right information for a given mission and situation.

Our goal was to ensure that the capabilities provided on tactical handheld computing devices are flexible enough to allow solders to control the amount and type of data that they receive and adaptive enough to meet the needs of particular missions. To achieve this goal, we explored end-user programming techniques that could be used on smartphones. These techniques are intended to enable soldiers to program software on tactical handheld devices without requiring them to be professional software developers. Filtering incoming information and displaying it in intuitive formats helps soldiers not to be inundated with too much data. We are currently developing software for Android devices, but the fundamental concepts are applicable to other mobile platforms as well.

8.3 Approach

Our app—called eMONTAGE (Edge Mission-Oriented Tactical App Generator)—allows a soldier to build customized interfaces that support the two basic paradigms that are common to smartphones: maps and lists. For example, a soldier could build an interface that allows them to construct a list of friendly community members including names, affiliations with specific groups, information about whether the person speaks English, and the names of the person's children. If the soldier also specifies a GPS location in the customized interface they construct, the location of the friendly community members could be plotted on a map. Likewise, the same soldier could build other customized interfaces that capture specific aspects of a threatening incident, or the names and capabilities of non-governmental organizations responding to a humanitarian crisis.

eMONTAGE is intended for soldiers who are well-versed in their jobs but are not programmers. After we developed our initial prototype, we asked several soldiers to provide feedback. Not surprisingly, we found that soldiers who are Android users and relatively young and very comfortable around computing devices (i.e., digital natives) quickly learned the app and could use it to build a new application onsite. Conversely, those less comfortable with computing devices had a harder time. Since our goal is to make our software accessible to every soldier, we are simplifying, revising, and improving the user interface.

eMONTAGE is constructed primarily in Java and operates on an Android smartphone platform. It is a native Android app–it is not running on a browser. Some of the basic characteristics of eMONTAGE include

- All warfighter programming is performed on the phone. This means that the warfighter can build a customized app interface on the phone even if they do not have access to a laptop or desktop computer. This strategy presented a harder problem than a strategy that allowed customization of the app interface on a laptop that was then downloaded to the phone–we had to live within the limitations of the phone's screen display. We believe it is a relatively simple step to allow development on a laptop, and there are many cases where this strategy may be more convenient.
- Warfighters write no code. Again, our goal was to provide a capability useful to warfighters that did not require any programming skills. We achieved that goal by allowing warfighters to use a form-based interface both to enter data and to search on data. This form-based interface allowed the user to create complex data types (e.g., records with multiple fields) and reference to other stored information (e.g., other records). Using the interface, warfighters can enter data, edit data, and search for specific information. Our no-code strategy precludes the use of programming-like vocabulary and representations. This provision proved to be a learning experience for our development team because we naturally fall into the use of terms in ways consistent with our engineering backgrounds.
- Our app supports the two most popular *display paradigms* on smartphones–lists and maps. A good example of a list is information retrieved based on a search for a particular type of person, place, or thing (e.g., coffee shop). The map-display paradigm typically takes information from lists and used geo-location information to display it on a map (e.g., coffee shops displayed on a map). Our initial investigation of publicly available app interfaces and several DoD-specific app interfaces led to an understanding that these two display paradigms were sufficient to cover the vast majority of app interfaces.
- User warfighter-constructed capabilities are sandboxed inside the single Android app that we constructed. This enables warfighter-constructed capabilities implemented in our app to benefit from improvements in Android security and emerging security strategies for Android smartphones within the DoD.
- e MONTAGE is a native Android app. It is not a web app executing within a browser. This presented several critical advantages, including a better user experience, better performance, and more complete access to libraries, devices, and the file system. We have since received feedback suggesting this was the correct decision from sources familiar with DoD efforts to port applications running on laptops and desktops to Android devices.
- For our initial version of eMONTAGE, we selected an object database (DB4.0) rather than SQLite that is provided as part of Android. This strategy offered several advantages, including flexibility to dynamically create new types and to extend existing types, as well as powerful application programming interfaces (APIs) that simplified our implementation.

- To speed searches of data, we used an external search capability rather than the native DB4.0 search capability.

Figure 21, Figure 22, and Figure 23 show how basic warfighter-specified types are created. In these examples, four types that reflect a disaster-response mission are created: missing persons in a disaster area, NGOs that are participating in the relief effort, information about volunteers, and tasks that assign volunteers to missing persons. Figure 21 reflects these four types. Figure 22 shows how the warfighter adds details to the type called *person*. Figure 23 shows the type *person* with several details added (e.g., name, age, picture). The details *lastseenlocation* and *marker* allow geo-location information about a person to be added for display on a map. *icon* specifies the display icon that should be used for the *lastseenlocation*.

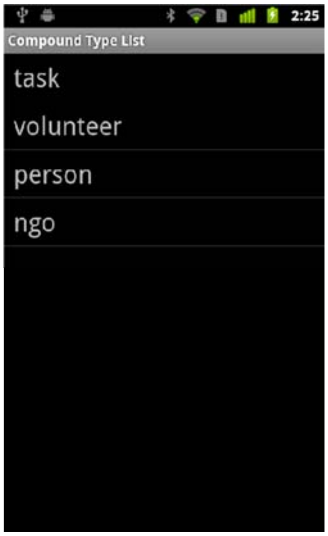


Figure 21: Creating Basic Types

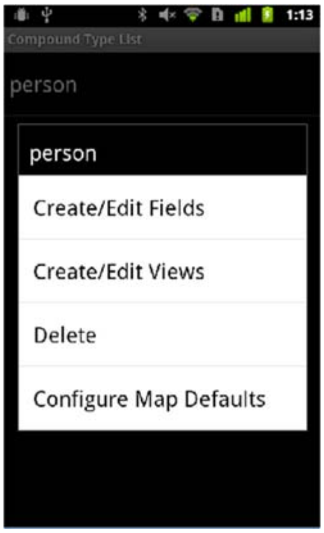
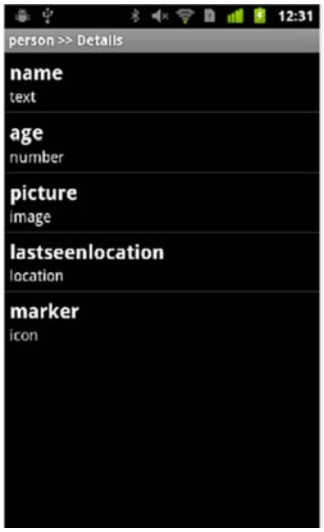


Figure 22: Adding Fields to a Type



person >> Details

name
text

age
number

picture
image

lastseenlocation
location

marker
icon

Figure 23: A Type (person) *With Fields Added*

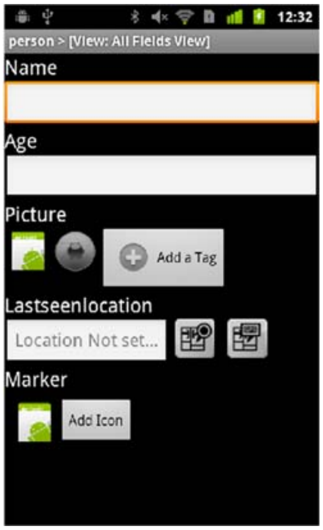
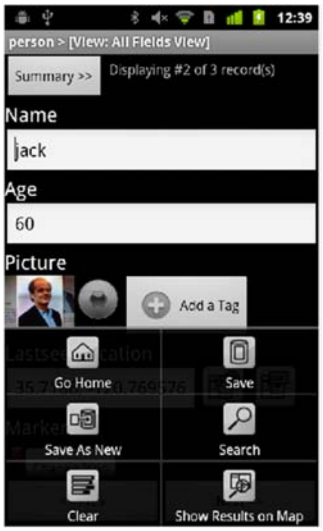


Figure 24: Input Form for Person

Figure 24 shows the resulting input form for a person that is generated from details warfighters enter for the person type. This form allows the warfighter to create records for a large number of persons.



person > [View: All Fields View]

Displaying #2 of 3 record(s)

Name

jack

Age

60

Picture

Figure 25: Information About a Specific Person

Figure 25 shows the phone displaying information about a specific person. The same screen is used to enter and display information, and to search for information that matches one or more fields in the database. Thus, if a warfighter has taken a picture of a person, they can search for all that is known about that person (e.g., name, age, *lastseenlocation*). Of course, this assumes that the warfighter has access to facial/image-recognition software on the handheld device, on a local platform such as a Humvee, or in an enterprise database. Providing local computation support that allows mobile handheld users access to sophisticated computing capabilities in tactical environments is the focus of another part of our work (see http://blog.sei.cmu.edu/post.cfm/cloud-computing-at-the-tactical-edge and http://blog.sei.cmu.edu/post.cfm/cloud-computing-for-the-battlefield)

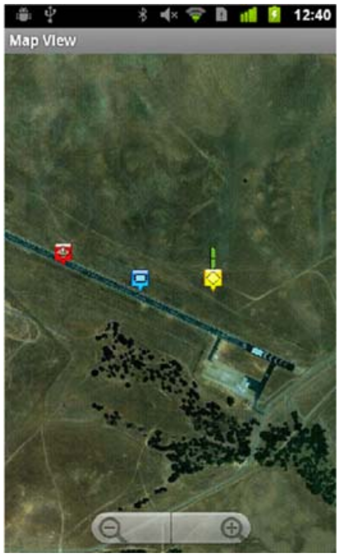


Figure 26: Information Displayed on a Map

Figure 26 shows how *lastseenlocation* of a person would appear displayed on a map (for example, the blue icon) along with other information.

8.4 Collaboration[29]

We are working with Dr. Brad Myers of the Carnegie Mellon University Human Computer Interaction Institute to identify and implement improved strategies for warfighter interaction with our Android smartphone. Dr. Myers is an expert on strategies for end-user programming and has worked to improve user interfaces on DoD programs such as the Command Post of the Future (CPoF).

Another major goal for 2012 is to work directly with DoD programs that are providing warfighter capabilities on smartphones. We will take advantage of existing relationships with programs such as DARPA Transformative Apps and FBCB2 to try out eMONTAGE. We continue to look for more DoD partners who want a simple way to build customized apps to provide new capabilities in support of warfighters.

8.5 Evaluation Criterion

Currently, eMONTAGE can handle the basic information types that are available on an Android phone, including images, audio, and data. Technologies such as fingerprint readers and chemical sensors are being miniaturized and will likely be incorporated into future handheld devices or easily connected to them. With each new technology, an appropriate basic type will have to be added to eMONTAGE. Fortunately, this is a relatively straightforward programming operation, but it does require engineering expertise. As a new type becomes available, professional engineers will add it to eMONTAGE, thereby making the type available to soldiers who may have little or no programming expertise.

Although it is intended to support warfighters in tactical environments, eMONTAGE is, by design, not DoD specific. This application can be used by other government organizations—or even non-government organizations— that want a user-customizable way to capture information about any variety of people, places, and things and share this information effectively in the enterprise. For example, it could be used by organizations such as the Department of Homeland Security to quickly build apps specific to a disaster-recovery situation. eMONTAGE is customizable to the DoD needs by providing access to DoD data sources, use of appropriate iconology, and development of interfaces unique to DoD missions and tasks.

One of the key unresolved problems with the use of mobile devices in a military setting is security. We are addressing this problem in several ways. First, we are porting our system to an Android kernel with improved security characteristics that is being developed by a DoD program. This allows our application to build on and benefit from DoD-specific security enhancements developed for the Android platform. Second, we are developing an approach to validate security and other quality characteristics of Android applications–both ours and those developed by others. This approach will employ a set of coding and runtime rules specifically designed to eliminate vulnerabilities, API violations, and runtime errors on Android platforms. We will develop a framework to check apps against this set of rules, using static analysis to check coding rules and directed random testing to check runtime rules. We will test the capability against the growing DoD Android code base and our own apps. We hope that this approach will be widely applied to

[29] The authors wish to thank Dan Plakosh for his foundational work at the Software Engineering Institute in the area of end user programming. We also wish to thank Joe Seibel for assisting on the usability study.

ensure that applications deployed on Android smartphones and similar devices exhibit appropriate security-related behaviors.

8.6 Results

Perhaps the greatest value of eMONTAGE will come as we integrate the warfighter-directed front end with DoD specific backend data sources (e.g., a database of events, photos of people). This capability will provide mashups that support soldiers by capturing multiple sources of information for display and manipulation based on mission requirements. Our initial approach employed a simple strategy requiring that a new app be generated for each backend data source. In effect, data sources were hard coded to the app. This strategy had the severe limitation of prohibiting simultaneous viewing of data from multiple sources. The new strategy we are now implementing is to allow mashups of data from various backend data sources. This allows simultaneous viewing of data from these sources. We are also developing an improved strategy to filter out irrelevant data, and to sort and arrange data for better display on handheld devices with limited screen space. Once these new capabilities are available in 2012, it will become much easier to build phone interfaces to new data sources and extend these interfaces with additional information.

We are also revising the smartphone interface during 2012 to make it easier to use and more capable. The new interface will provide a more flexible search capability that does not constrain the search to a particular type or source of information. For example, a soldier will be able to search multiple data sources for any information regarding a location. To make sure that the soldier is not overwhelmed with a huge volume of data from this sort of search, we will build a simple-to-use conditional capability that will provide operators such as *before* and *after* related to time, and *less than* and *greater than* related to amount. The new interface will also make it easier to construct interfaces and enter data by using heuristics to predict data types.

9 Communicating the Benefits of Architecting within Agile Development: Quantifying the Value of Architecting within Agile Software Development via Technical Debt Analysis

Robert L. Nord, Ipek Ozkaya, Raghvinder S. Sangwan, Nanette Brown

9.1 Purpose

The delivery of complex, large-scale systems poses the ongoing challenge of how best to balance rapid development with long-term value. Agile software development methods are increasing in popularity, in industry and DoD alike, with the hope that they can contribute to the solution.

A key challenge in Agile development is the ability to quantify the value of infrastructure and quality-related tasks which quite often are architectural. The current practice mostly assigns zero value to infrastructure management, future architecture planning, rearchitecting, and maintenance tasks in release planning. This is largely due to a lack of focus on the long-term impact of these infrastructure elements on customer-facing capabilities and overall system health as the system grows.

Agile methods conceptualize this trade-off between short- and long-term value with the technical debt metaphor where taking short-cuts to optimize delivery of capabilities in the short-term incurs debt, like financial debt, that needs to be paid off to optimize long-term success. Technical debt recasts a technical concept as an economic one. Cost, benefit, and value aspects of software development have begun to be addressed as a part of the value-driven software engineering agenda, but have not yet culminated in rigorous analysis models and research directions for large-scale projects. Research in the area of technical debt is only recently emerging, and mostly focused on defect management, which is retroactive, rather than monitoring trade-offs, which is a proactive mechanism to agile planning. Further research on developing quantitative models to appropriately manage technical debt is needed [Brown 2010a]. The goal of this project is creating a framework where the attributes of technical debt are understood and related to architecture decisions.

9.2 Background

Industry and government stakeholders continue to demand increasingly rapid innovation and the ability to adjust products and systems to emerging needs. Time frames for new feature releases continue to shorten, as exemplified by Z. Lemnios, Director of Defense Research and Engineering:

> “Get me an 80% solution NOW rather than a 100% solution two years from now and help me innovate in the field.” [Lemnios 2010]

Large-scale projects face the dilemma of balancing rapid deployment with long-term value. The term technical debt describes this trade-off between short- and long-term value and has already penetrated into practice. Popularized by agile software development techniques, an ongoing focus on management of technical debt is perceived as critical to the development of high quality software. Left unmanaged, such debt causes projects to face significant technical and financial problems leading to increased maintenance and evolution costs. Although agile practices of refactoring, test-driven development, and software craftsmanship are often sufficient to manage technical debt on small-scale projects, sound principles for managing debt on large-scale projects are lacking. Large-scale project owners drawn to agile practices by the allure of quicker time to market and improved responsiveness can find the promise of agility negated by increasing amounts of technical debt.

The technical debt metaphor was coined by Ward Cunnigham in his 1992 OOPSLA experience report [Cunningham 1992].

> *Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite... The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise.*

The metaphor highlights that like financial debt, technical debt incurs interest payments, which come in the form of the extra effort that we have to do in future development because of expedient design choices. Like financial debt, sometimes technical debt can be necessary. We can choose to continue paying the interest, or we can pay down the principal by re-architecting and refactoring. Although it costs to pay down the principal, we gain by reduced interest payments in the future.

While an appealing metaphor, theoretical foundations that provide key input to identifying and managing technical debt are lacking. In addition, while the term was originally coined in reference to coding practices, today the metaphor is applied more broadly across the project lifecycle and may include requirements, architectural, testing, or documentation debt.

There is a key difference between bad engineering principles employed which result in debt, and intentional strategic decisions with a key goal that requires incurring debt [McConnell 2006]. Martin Fowler expands this further describing technical debt using four dimensions [Fowler 2009] (see Figure 27).

	Reckless	**Prudent**
Deliberate	*We don’t have time for design.*	*We must ship now and deal with the consequences.*
Inadvertent	*What’s layering?*	*Now we know how we should have done it.*

Figure 27: Technical Debt [Fowler 2009]

The purpose of this LENS project was to focus on providing foundations for understanding and managing prudent, deliberate, and intentional debt. In the context of large-scale, long-term projects there is tension between code-level and architecture-level abstractions, especially when it comes to relating these to a global concept such as technical debt. This tension is also observed in related work in the following areas:

- *Agile software development techniques:* Although technical debt is coined in the context of agile software development, not much attention is given to how it is managed, especially in the context of architecture. In his book *Scaling Software Agility*, Leffingwell notes that none of the existing techniques pays much attention to software architecture, which at best will gradually emerge from a succession of refactorings [Leffingwell 2007].

- *Architecture-centric maintenance and evolution:* Technical debt resonates with maintenance activities when it needs to be repaid, especially with refactoring and re-architecting (see [Mens 2008] for a recent review of research and historical foundations of maintenance and evolution). Research work in this area has not addressed architecture-level evolution and maintenance which could be hindered or accelerated by taking on technical debt.
- *Code analysis:* Concepts like code smells and spaghetti code have been used to address code-level technical debt. Clone and defect detection, inferring change patterns are among concerns relevant to code-level technical debt [Kim 2009]. An existing debt visualization plug-in demonstrates how to monitor coding rules violations and providing measures using debt heuristics [Gaudin 2012].
- *Software economics and metrics:* Technical debt recasts a technical concept as an economic one. Cost, benefit, and value aspects of software development have been addressed under value-driven software engineering agenda in the broad but have not yet culminated in rigorous analysis models for managing debt [Biffl 2009].

9.3 Approach

Our hypothesis in this work is that a sound analytic model that assists with the detection, monitoring, and reduction of architectural technical debt will provide better input for managing short versus long-term design decisions and result in systems that better meet their goals.

Our approach has the following thrusts:

1. define and refine quantifiable properties of technical debt based on using module structure metrics to
   - gain insight into degrading architecture quality, and
   - identify the “tipping point” to trigger re-architecting decisions
2. monitor architecture quality by making architectural dependencies visible using dependency structure matrices (DSM)
3. model the impact of debt and pay-back by calculating the rework cost using the propagation cost metric based on DSM analysis

9.4 Collaborations

We established research collaborations with Dr. Philippe Kruchten, University of British Columbia and his Masters students Erin Lim and Marco Gonzales. Dr. Raghu Sangwan from Pennsylvania State University also worked with us closely. Manuel Pais, MSE student at the School of Computer Science at Carnegie Mellon University, conducted a study on tools for technical debt analysis for us. Don O’Connell from Boeing provided us with technical feedback. Nanette Brown from NoteWell Consulting and the Software Engineering Institute at the time was a member of the team during the study.

9.5 Evaluation Criteria

The ability to elicit the impact of technical debt early on and in an ongoing basis provides an advantage in managing prudent and deliberate debt. Therefore, the evaluation criteria for the project are focused on both the fidelity and the timing of the analysis results, and are the following:

- able to analyze problem earlier in the life cycle
- as good or better fidelity as current analysis
- improved ratio of capabilities delivered over total effort
- able to analyze problems of at least one order of magnitude in the number of iterations in the plan and number of modules up to 100

9.6 Results

Within any iterative and incremental development paradigm, there exists a tension between the desire to deliver value to the customer early and the desire to reduce cost by avoiding architectural refactoring in subsequent releases [Larman 2003]. The choice between these two competing interests is situational. In certain contexts, early delivery might be the correct choice, for example, to enable the release of critically needed capabilities or to gain market exposure and feedback. In other contexts, however, delayed release in the interest of reducing later rework might be the choice that better aligns with project and organizational drivers and concerns.

We conducted a study with the goal of identifying whether there are distinct return-on-investment outcomes of development paths if contrasting business goals are at stake and whether it is possible to monitor the outcomes at each delivery release. The two contrasting goals we studied are (1) maximizing value for the end user and (2) minimizing implementation cost due to rework. Then we considered a third path: a middle ground of integrated return on investment using both value and cost to guide each decision. We analyzed how propagation cost changed from iteration to iteration as we optimized for these different outcomes.

Architecture quality and visibility are closely related. Reasoning about quality with a quantifiable model requires that certain architectural properties be represented objectively and in a repeatable manner across systems for the model to work. It is for this reason that we look more closely into using dependency structure matrix (DSM) and domain modeling matrix (DMM) analysis to provide a representation with support for objective metrics generation. We discuss the propagation cost metric in this context.

The propagation cost measures the percentage of system elements that can be affected, on average, when a change is made to a randomly chosen element [MacCormack 2008]. Some existing approaches use DSM analysis to examine “direct” dependencies and provide metrics measuring complexity [Sangwan 2009] and decision volatility [Sethi 2009]. Other approaches use a propagation cost metric to take into account “indirect” dependencies and observe correlations between software coupling and the coordination requirements among developers [Amrit 2010]. These approaches take a snapshot of the current system state using a DSM, which then becomes the basis for this analysis. To calculate the cost of change, we use the system propagation cost metric that can be derived from the DSM of architecture elements. Propagation cost, Pc, is calculated as the density of a matrix M as represented by the ratio of the total number of filled cells due to direct or indirect dependencies among its elements ($\sum_{i=0}^{n} Mi$) to the size of the matrix (n^2):

$$Pc = Density\ (\textstyle\sum_{i=0}^{n} Mi\ ) / n^2 \qquad (1)$$

According to MacCormack and colleagues, this metric captures the percentage of system elements that can be affected, on average, when a change is made to a randomly chosen element [MacCormack 2008].

End-user value at each release is measured by adding the value of all customer requirements supported by that release. For the purposes of our study, value reflects the priority points of the customer requirements. Total cost of a release n, Tc_n, is the combination of the cost to implement the architectural elements selected to be added in that release, Ic_n, plus the cost to rework any preexisting architectural elements, Rc_n.

$$Tc_n = Ic_n + Rc_n \qquad (2)$$

Implementation cost, Ic_n, for release *n* is computed as follows:

- Sum the implementation cost of all architectural elements, AE_j, implemented in release n (and not present in an earlier release).
- The implementation cost is assumed to be given for all individual architectural elements (independent of dependencies).

Rework cost is incurred when new elements are added to the system during this release, and one or more pre-existing elements have to be modified to accommodate the new ones. This includes elements that can be identified with their direct dependencies on the new elements as well as those with indirect dependencies represented by the propagation cost. Rework cost, Rc_n, for release *n* is computed as follows:

- Compute the rework cost associated with each new architectural element, AE_k, implemented in release n. For each preexisting AE_j with dependencies on AE_k, multiply the number of dependencies that AE_j has on AE_k times the implementation cost of AE_j times the propagation cost of release (n - 1).
- Sum the rework costs associated with all new architectural elements AE_k implemented in the release.

The algorithm for rework is directional in nature and represents an initial effort to formalize the impact of architectural dependencies upon rework effort. The cost of each architectural element, the number of dependencies impacted by each architectural change, and the overall propagation cost of the system may all be seen as proxies for complexity, which is assumed to affect the cost of change. The relative weighting and relationship between these factors, however, is a subject of future research efforts. Therefore, within the context of our analysis, rework cost is interpreted as a relative rather than an absolute value and is used to compare alternative paths and to provide insight into the improvement or degradation of architectural quality across releases within a given path.

We picked the Management Station Lite (MSLite) system for the study, a system with which we have previous experience and access to the code, architecture, and project-planning artifacts [Sangwan 2006; Sangwan 2008].

The methodology used for the study consisted of first defining the system requirements and system structure using DSM and DMM analysis. Second, we modeled five strategies of development paths by which to realize the requirements and system structure:

- Paths 1 and 2: Create value- and cost-focused what-if scenarios to understand the space of decisions in iteration planning.
- Path 3: Create integrated return-on-investment what-if scenario to understand trade-offs of value and cost at key decision points in release planning.
- Planned Path: Apply proposed architecture quality analysis to the development path of MSLite as it was planned.
- Actual Path: Apply proposed architecture quality analysis to the development path of MSLite as it was implemented. We used dependency metrics provided in Lattix (2011) for this purpose.

The third and final step within our method was to analyze the patterns of value generation and cost incursion that result from each of the development paths.

We compare results along two dimensions: (1) comparing what-if development path scenarios to understand the space and boundaries of decisions with respect to value and cost, and (2) comparing Planned Path to Actual Path to understand the predictive nature of using the architecture earlier in the life cycle than code can be used.

We identify the set of DSM/DMM matrices that contain all the intra- and inter-domain dependencies relevant for our system analysis:

- Customer requirements DSM (DSM_{CR})
- Architectural elements DSM (DSM_{AE})
- DMM mapping customer requirements to architectural elements

The use of dependency management techniques such as DSM and DMM provides strong support in the path definition process, allowing us to identify the architectural elements to be implemented and the customer requirements to be delivered in each release for all development paths. Our goal is to see how these paths would differ when propagation cost is the basis for understanding the incurred rework costs. Each path consists of a set of software releases in which each release implements new architectural elements. In this study, all paths have the same defined end point and use the same catalog of architectural elements as building blocks to control for the differences in implementation and to highlight the influence of the orderings in the paths. Once defined, these paths serve as input for performing path analysis and deciding on the best one for the project, given specific customer needs. In other words, within a specific project context, we ask, “How willing is the customer to accept additional rework cost in exchange for the early delivery of valued functionality?”—in other words, paying back debt.

Path 1: Maximizing value for the end user

To define this development path, Table 4 is used to select user stories and system-level acceptance test cases with high end-user value for early delivery. First, the three most valuable user stories and/or system acceptance test cases are identified. Then, the customer requirements $\mathrm{DSM}_{\mathrm{CR}}$ are used to retrieve other user stories upon which the high-value stories depend. This defines the complete subset of user stories and/or system-level acceptance test cases to deliver.

Even though the selected user stories may have implied quality attribute requirements, they are not explicitly considered. For the system-level acceptance test cases, however, we look at the DMM to identify the architectural elements they depend on. Then we use $\mathrm{DSM}_{\mathrm{AE}}$ to retrieve any additional architectural elements upon which the initial set of architectural elements depends. The resulting subset of user stories, acceptance test cases, and architectural elements constitutes the first contemplated release.

The process repeats by picking the next three highest ranking customer requirements in terms of end-user value and following the same steps described until the next release is obtained. The final step of this iterative process will deal with the three (or fewer) customer requirements of lowest value to implement.

Path 2: Minimizing implementation cost

Cost is defined in terms of both the cost to implement an architectural element and the number of architectural elements that will be affected by a change to a single element in the system (rework cost). Therefore, to minimize rework cost the architectural elements with fewer dependencies on other elements would be delivered first as they are least likely to require modifications when new elements are introduced. We look at the DMM to identify requirements (if any) that are supported by the elements just implemented. This defines the features visible to the customer and the value delivered for the release.

Path 3: Integrated return on investment

This development path is focused on delivering high-value capabilities and pulling in the needed architectural elements on demand to support the selected capabilities. First, we identify the three most valuable user stories and/or system acceptance test cases. We then look into the customer requirements $\mathrm{DSM}_{\mathrm{CR}}$ to retrieve the stories upon which the high-value stories depend. After the complete subset of user stories and/or system-level acceptance test cases to deliver is defined, we look at the DMM to identify the architectural elements they depend on. Then we use $\mathrm{DSM}_{\mathrm{AE}}$ to retrieve any additional architectural elements upon which the initial set of architectural elements depends. The resulting subset of user stories, acceptance test cases, and architectural elements constitutes the first potential release. The development team then determines what is possible given the resources. If the three highest ranking stories are not all possible given the resources, the team may need to negotiate with the customer to deliver the stories ranked first or second and begin partial implementation of the others when possible or defer implementation to a later release. The process repeats by picking the next three highest ranking customer requirements in terms of end-user value and following the same steps described until the next release is obtained.

The final step of this iterative process will deal with the three (or fewer) customer requirements of lowest value to implement.

Table 3 lists the features to be implemented at each release of the three paths based on these three different approaches.

Table 3: Allocation of Stories to Release in Each Path

	Path 1	Path 2	Path 3
Release 1	US01: Visualize field object properties US02: Change field object properties US03: Add alarm condition	None	US01: Visualize field object properties US02: Change field object properties
Release 2	US04: Alarm notification & acknowledgement US06: Add logic condition and reaction US07: Alarms and logic/reaction pairs persistence	None	US03: Add alarm condition US04: Alarm notification & acknowledgement US06: Add logic condition and reaction
Release 3	US05: Ignore alarm notification ATC14: Connect similar field system ATC16: Connect field system providing new functionality	ATC14: Connect similar field system ATC15: Connect field system using different format and interfaces ATC16: Connect field system providing new functionality ATC21: Field object properties updated	US7: Alarms and logic/reaction pairs persistence US05: Ignore alarm notification ATC14: Connect similar field system
Release 4	ATC18: Alarm notification speed ATC19: No loss of alarm notifications ATC21: Field object properties updated US08: Secure access to system	US01: Visualize field object properties US02: Change field object properties US08: Secure access to system ATC17: Field object properties update speed ATC20: Field object properties data access	ATC16: Connect field system providing new functionality ATC18: Alarm notification speed ATC19: No loss of alarm notifications ATC21: Field object properties updated US08: Secure access to system
Release 5	ATC15: Connect field system using different format and interfaces ATC17: Field object properties update speed ATC20: Field object properties data access	US03: Add alarm condition US04: Alarm notification & acknowledgement US05: Ignore alarm notification US06: Add logic condition and reaction US07: Alarms and logic/reaction pairs persistence ATC18: Alarm notification speed ATC19: No loss of alarm notifications	ATC15: Connect field system using different format and interfaces ATC17: Field object properties update speed ATC20: Field object properties data access

Associated with the different goals of these paths were different heuristics for release definition. For instance, for Path 1, each release was conceptualized as an external deliverable to an end user. The release definition was therefore based upon the attainment of a cohesive set of end-user values. For Path 2, on the other hand, each release was conceptualized as a verifiable executable deliverable either internally for validation or externally for user adoption. For Path 3, similarly to Path 1, each release is conceptualized as an external deliverable to an end user, but it also keeps a cohesive set of end-user values focusing on underlying architectural elements.

Table 4 shows the summary of how we arrived at the cumulative cost for each release in the previous table. Implementation cost (I_c) is based on the cost associated with implementing all the architectural elements in a given release. Rework cost (R_c) is calculated using the algorithm described above and uses the propagation cost (P_c) as its basis. Total cost (T_c) is simply the sum of I_c and R_c. Cumulative cost is the running sum of T_c.

Table 4: Comparison of the Costs of the Three What-If Development Paths

	Path 1					Path 2					Path 3				
	Ic	**Pc**	**Rc**	**Tc**	**Cumulative Cost**	**Ic**	**Pc**	**Rc**	**Tc**	**Cumulative Cost**	**Ic**	**Pc**	**Rc**	**Tc**	**Cumulative Cost**
Release 5	12	0.35	10	22	108	16	0.30	0	16	74	12	0.30	8	20	85
Release 4	11	0.31	12	23	86	15	0.29	0	15	58	2	0.24	0	2	66
Release 3	17	0.46	12	29	63	10	0.29	0	10	43	11	0.26	0	11	64
Release 2	16	0.41	0	16	34	12	0.24	0	12	33	19	0.37	4	23	53
Release 1	18	0.56	0	18	18	21	0	0	21	21	30	0.32	0	30	30
Total	74					74					74				

We graphically show the value of capabilities delivered over the total effort for each of the three paths over five releases in Figure 28. The total implementation effort (cumulative cost) of the system independent of rework is depicted as 100 percent cost on the x-axis of the figure. The additional cost over 100 percent reflects the rework or expense to deal with the technical debt.

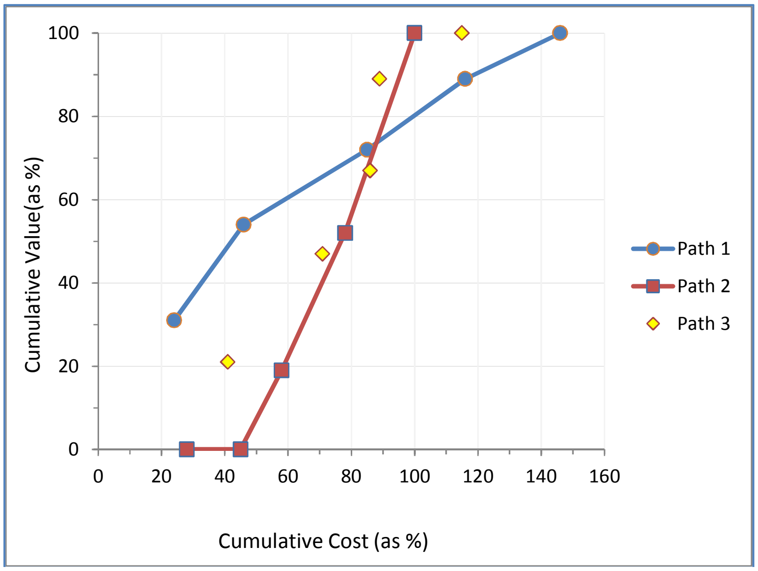


Figure 28: Value of Capabilities Delivered Over Total Effort for What-If Development Paths

Figure 28 shows iterations of uniform duration and reflects cadences of the development effort. In this case, there are 10 iterations spanning development, with each iteration representing 10 percent of the release cost. The product owner asks for the high-priority capabilities, and the developers say what is possible given the allotted resources. Developers plan at the granularity of tasks to develop elements that are needed to implement a capability.

Each path releases five increments of the product over the course of development, according to its own timeline. Releases reflect stakeholder values and so are not uniform in duration.

- Path 1 shows high value during the first two releases, but the delivery of value tapers off as subsequent releases take longer, an indication of the rework needed to deal with the growing complexity of dependencies.
- Path 2 shows that there is no value delivered to end users early on, as the team focuses on the architecture. Once the architecture is in place, the team settles into a rhythm of releasing high-value capabilities every two iterations.
- Path 3 shows that the combined emphasis on high-value capabilities and architecture to manage dependencies makes delivery more consistent over time.

The impact of the cost difference of the three paths is observed in the rework cost. For instance, the rework cost associated with Path 1, Release 3 is 12. This cost is incurred because there are elements that need to be reworked.

When we conducted an actual versus planned analysis of the MSLite code, using the architecture and planning artifacts, we observed the following. For the MSLite system in general, the code conformed to the architecture, as revealed by comparing the DSM constructed from the architecture and the DSM constructed from the code. The code generally followed the module structure and was implemented according to the planned iterations.

This analysis demonstrates that we can improve project monitoring by providing quantifiable quality models of the architecture during iteration planning to model technical debt. We are investigating the use of the propagation cost metric to model the impact of degrading architectural quality in order to understand when to invest in improving the architecture as well as to inform trade-off discussion involving architectural investment versus the delivery of end-user valued capabilities. Our goal is to provide an empirical basis on which to chart and adjust course. Now that we have a baseline, we plan to investigate incorporating uncertainty in the economic framework and enhancing the approach to model runtime dependencies.

We accounted for rework in the current approach using a simple cash flow model in which cost is incurred at the time of the rework. There are economic models that include rework cost that is predicted in future releases. This is essential in understanding the impact of payback strategies. These models become more complex since there are more choices for when to account for the future debt. The ability to quantify degrading architecture quality and the potential for future rework cost during iterative release planning as each release is being planned is a key aspect of managing rework strategically [Brown 2010b]. Managing strategic shortcuts, as captured by the technical debt metaphor, requires better characterization of the economics of architectural violations across a long-term road map rather than enforcing compliance for each release. Our approach facilitates reasoning about the economic implications and perhaps deliberately allowing the architecture quality to degrade in the short term to achieve some greater business goal (all the while continuing to monitor the quality of the architecture and looking for the opportune time to improve).

We accounted for module dependencies in the current approach to support analysis of modifiability. During this study, we were able to account for runtime dependencies indirectly because our model system allowed us to map the component and connector (C&C) view to the module-structure view one-to-one. As an extension of our approach, we are looking at directly modeling runtime dependencies so we can reason about the quality attributes (e.g., performance schedulability) that they support and provide better engineering information to the development teams early on and continuously by taking advantage of architecture artifacts.

9.7 Publications and Presentations

Brown, N., Nord, R., Ozkaya, I., & Pais M. *Analysis and Management of Architectural Dependencies in Iterative Release Planning*, WICSA 2011.

Brown, N. and Nord R. *Hands-on Session on Making Hard Choices about Technical Debt*, , Delivered at AGILE 2011.

Brown, N., Nord, R, & Ozkaya, I. *Tutorial on Strategic Management of Technical Debt*, Delivered at SATURN 2011, WICSA 2011.

Brown, N., Nord, R., & Ozkaya, I. *Enabling Agility Through Architecture*, CrossTalk, Nov/Dec 2010.

Nord, R. L., Ozkaya, I, Brown N., & Sangwan, R.S. *Modeling Architectural Dependencies to Support Software Release Planning*, 13th International Design Structure Matrix Conference, Sept 13-15, 2011 2011 MIT.

Ozkaya, I., Kruchten, P., Nord, R.L., & Brown, N. Second International Workshop on Managing Technical Debt: (MTD 2011). ICSE 2011: 1212-1213.

9.8 References

[Amrit 2010]
Amrit, C. and van Hillegersberg, J., 2010. “Coordination Implications of Software Coupling in Open Source Projects.” *Open Source Software: New Horizons, IFIP Advances in Information and Communication Technology 319* (2010): 314-321.

[Biffl 2009]
Biffl, Stefan, Aurum, Aybuke, Boehm, Barry, Erdogmus, Hakan, & Grunbacher, Paul. *Value-Based Software Engineering*. Springer Berlin Heidelberg, 2009.

[Brown 2010a]
Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N. “Managing Technical Debt in Software-Reliant Systems,” 47-52. *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*. ACM, 2010

[Brown 2010b]

Brown, N., Nord, R., Ozkaya, I. “Enabling Agility through Architecture.” *Crosstalk*, Nov/Dec 2010.

[Cunningham 1992]

Cunningham, W. *The WyCash Portfolio Management System OOPSLA Experience Report*, 1992.

[Fowler 2009]

Fowler, M. *Technical Debt Quadrant*. Bliki [Blog] 2009. Available from: http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html.

[Gaudin 2012]

Gaudin, O. *Evaluate Your Technical Debt With Sonar*. 2012. Available from: http://www.sonarsource.org/evaluate-your-technical-debt-with-sonar/

[Kim 2009]

Kim, M. and Notkin, D. “Discovering and Representing Systematic Code Changes,” 309-319. In *Proceedings of the 2009 IEEE 31st international Conference on Software Engineering*. IEEE, 2009

[Larman 2003]

Larman, C., Basili, V.R., 2003. “Iterative and Incremental Development: A Brief History.” *IEEE Computer 36* 6 (2003):47–56.

[Leffingwell 2007]

Leffingwell, D. *Scaling Software Agility*. Addison-Wesley, 2007.

[Lemnios 2010]

Lemnios, Z. *Statement of Testimony Before the United States House of Representatives Committee on Armed Services Subcommittee on Terrorism, Unconventional Threats and Capabilities*, March 23, 2010. http://www.dod.mil/ddre/Mar232010Lemnios.pdf

[MacCormack 2008]

MacCormack, A., Rusnak, J., Baldwin, C., 2008. *Exploring the Duality between Product and Organizational Architectures: A Test of the Mirroring Hypothesis (Version 3.0)*. Harvard Business School, 2008.

[McConnell 2006]

McConnell, S. *Software Estimation—Demystifying the Black Art*. Microsoft Press, 2006.

[Mens 2008]

Mens, T. Introduction. “ Introduction and Roadmap: History and Challenges of Software Evolution,” 1-11. *Software Evolution*, Springer, 2008.

[Sangwan 2006]

Sangwan, R., Bass, M., Mullick, N., Paulish, D.J., Kazmeier, J. *Global Software Development Handbook*. Auerbach Publications, 2006

[Sangwan 2008]
Sangwan, R.S., Neill, C., Bass, M., El Houda, Z. Integrating a Software Architecture-Centric Method Into Object-Oriented Analysis and Design. *Journal of Systems and Software 81* (2008): 727–746.

[Sethi 2009]
Sethi, K., Cai, Y., Wong, S., Garcia, A., Sant'Anna, C., 2009. “From Retrospect to Prospect: Assessing Modularity and Stability From Software Architecture,” 269–272. *Proceedings of the Joint 8th Working IEEE/IFIP Conference on Software Architecture and 3rd European Conference on Software Architecture (WICSA/ECSA), Working Session*, Cambridge, U.K., 2009.

REPORT DOCUMENTATION PAGE	*Form Approved* *OMB No. 0704-0188*

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. **AGENCY USE ONLY** (Leave Blank)	2. **REPORT DATE** August 2012	3. **REPORT TYPE AND DATES COVERED** Final

4. **TITLE AND SUBTITLE** Results of SEI Line-Funded Exploratory New Starts Projects	5. **FUNDING NUMBERS** FA8721-05-C-0003

6. **AUTHOR(S)**

Len Bass, Nanette Brown, Gene Cahill, William Casey, Sagar Chaki, Cory Cohen, Dionisio de Niz, David French, Arie Gurfinkel, Rick Kazman, Ed Morris, Brad Myers, William Nichols, Robert L. Nord, Ipek Ozkaya, Raghvinder S. Sangwan, Soumya Simanta, Ofer Strichman, Peppo Valetto

7. **PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)** Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	8. **PERFORMING ORGANIZATION REPORT NUMBER** CMU/SEI-2012-TR-004
9. **SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)** HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	10. **SPONSORING/MONITORING AGENCY REPORT NUMBER** ESC-TR-2012-004

11. **SUPPLEMENTARY NOTES**

12A **DISTRIBUTION/AVAILABILITY STATEMENT** Unclassified/Unlimited, DTIC, NTIS	12B **DISTRIBUTION CODE**

13. **ABSTRACT (MAXIMUM 200 WORDS)**

The Software Engineering Institute (SEI) annually undertakes several line-funded exploratory new starts (LENS) projects. These projects serve to (1) support feasibility studies investigating whether further work by the SEI would be of potential benefit and (2) support further exploratory work to determine whether there is sufficient value in eventually funding the feasibility study work as an SEI initiative. Projects are chosen based on their potential to mature and/or transition software engineering practices, develop information that will help in deciding whether further work is worth funding, and set new directions for SEI work. This report describes the LENS projects that were conducted during fiscal year 2011 (October 2010 through September 2011).

14. **SUBJECT TERMS** Line funded exploratory new starts, research	15. **NUMBER OF PAGES** 100

16. **PRICE CODE**

17. **SECURITY CLASSIFICATION OF REPORT** Unclassified	18. **SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	19. **SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	20. **LIMITATION OF ABSTRACT** UL