UNIVERSITY OF CALIFORNIA
SANTA CRUZ

SOFTWARE SIGNATURE DERIVATION FROM SEQUENTIAL DIGITAL FORENSIC ANALYSIS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER SCIENCE

by

Alexander J. Nelson

June 2016

The Dissertation of Alexander J. Nelson
is approved:

Table of Contents

List of Figures

List of Tables

Abstract

Software signature derivation from sequential digital forensic analysis

by

Alexander J. Nelson

Hierarchical storage system namespaces are notorious for their immense size, which is a significant hindrance for any computer inspection. File systems for computers start with tens of thousands of files, and the Registries of Windows computers start with hundreds of thousands of cells. An analysis of a storage system, whether for digital forensics or locating old data, depends on being able to reduce the namespaces down to the features of interest. Typically, having such large volumes to analyze is seen as a challenge to identifying relevant content. However, if the origins of files can be identified—particularly dividing between software and human origins—large counts of files become a boon to profiling how a computer has been used. It becomes possible to identify software that has influenced the computer's state, which gives an important overview of storage system contents not available to date.

In this work, I apply document search to observed changes in a class of forensic artifact, cell names of the Windows Registry, to identify effects of software on storage systems. Using the search model, a system's Registry becomes a query for matching software signatures. To derive signatures, file system differential analysis is extended from between two storage system states to many sequences of states. The workflow that creates these signatures is an example of analytics on data lineage, from branching data histories. The signatures independently indicate past presence or usage of software, based on consistent creation of measurably distinct artifacts. A signature search engine is demonstrated against a machine with a selected set of applications installed and executed. The optimal search engine according to that machine is then turned against a separate corpus of machines with a set of present applications identified by several non-Registry forensic artifact sources, including the file systems, memory, and network captures. The signature search engine corroborates those findings, using only the Windows Registry.

To my parents,

Harold and Young Shin.

Acknowledgments

Applying to graduate studies was one of the most difficult challenges I undertook. The strain was nonsensical, versus my gut having told me for years that I would be going. My friends gave me emotional support to go. It was two doctors that helped most with my soul-searching as I wrote my application essay: my father, Harold Nelson, Jr. (who I will now disambiguate with great pleasure as "Dr. Nelson, Jr."), and Jason Donev. Thank you, my home friends and family, for helping me get here.

At the other end of graduate studies, I am grateful to my committee, Darrell, Ethan, Simson, and Golden: Thank you for believing in me, and helping me work through my challenges.

From home, through California, to Maryland, I have many other kind people to whom I am grateful. In Santa Cruz, my professors and labmates in the Storage Systems Research Center gave me years of camaraderie. I enjoyed philosophy-shaping walks with Tom Kroeger and Ahmed Amer that helped me form my writing strategies. The forensics research community, particularly the frequenters of the Digital Forensics Research Workshop (DFRWS), has made me feel welcome.

My colleagues at NIST have been wonderfully supportive. Thanks to the NSRL team, especially Doug White, Michael Ogata, Mary Laamanen, and John Tebbutt, for helping facilitate the Diskprint research. Megan Dower provided an invaluable review of an early version of this thesis text. Thanks to Barbara Guttmann, Lee Badger, and Matt Scholl for keeping the way forward walkable. The chances to discuss this research outside of my everyday groups have also been a boon, and I am grateful to Alden Dima, Jim Filliben, and David Flater for the experimental design discussion. Thanks to Ian Soboroff for the discussion on search evaluation and model construction.

My collaborations since arriving in Maryland have been a pleasure. Thanks to Jim Jones, Kathryn Laskey, and Tahir Khan for their analysis discussions, and to Tom Austin for helping me explore programming languages.

And, Sarah. Thank you for being with me.

Chapter 1

Introduction

There was only one student in the room, who was bending over a distant table absorbed in his work. At the sound of our steps he glanced round and sprang to his feet with a cry of pleasure. "I've found it! I've found it," he shouted to my companion, running towards us with a test-tube in his hand. "I have found a re-agent which is precipitated by hæmoglobin, and by nothing else." Had he discovered a gold mine, greater delight could not have shone upon his features.

Sir Arthur Conan Doyle, *A Study in Scarlet* [14]

1.1 Background of the Problem

Analysis of storage systems is challenged by growing data volume. File systems continue to grow, in counts of bytes, files, and directories. Volume and namespace complexity obscure files from their owners once the files leave recent memory, though there is a chance an owner can recall the purpose of a file on seeing it. In digital forensic analysis of a file system, where an analyst is inspecting files instead of the owner, the analyst has no such luxury of familiarity, and no relent from subject media growth.

Table 1.1: Select caseload processing statistics from the FBI Regional Computer Forensics Laboratory Program's annual reports. The "Data processed" column is measured in terabytes, not tebibytes.

Fiscal Year	Data processed (TB)	Examinations conducted	Sources
2003	82	987	[46, 49, 50]
2004	229	1,304	[47, 49, 50]
2005	457	2,977	[48, 49, 50]
2006	916	3,633	[49, 50]
2007	1,288	4,634	[50]
2008	1,756	4,524	[51]
2009	2,334	6,016	[52]
2010	3,393	6,564	[53]
2011	4,687	7,629	[45, 54, 55]
2012	6,582	8,566	[54, 55]
2013	6,567	7,273	[55]

For digital forensic analysts, forensic caseload and ingested data are accelerating. The US FBI's Regional Computer Forensics Laboratory program releases case workload statistics annually, with some counts excerpted in Table 1.1. If one makes the simple division estimate of average data processed per examination, this ten year window saw a growth from 83 GB processed per case in 2003, to 903 GB in 2013.

Table 1.2: Rough average files and directories (arithmetic means) of computer systems, according to the Microsoft series of metadata studies. These are summary statistics of computers used at a corporate campus for business, administration, and technical development purposes.

Year	Files	Directories	Source
1998	13k	–	[13, Table 2]
2000	30k	2.4k	[1, Section 3.1]
2004	90k	8.9k	[1, Section 3.1]
2009	225k	36k	[38, Section 4.3]

Alongside the problem forensic analysts have with caseload, file system namespace size is exploding. The FBI does not release reports of metadata from their cases, but measurements from Microsoft's campus systems show the metadata growth directions. The Microsoft series of metadata studies show that, in a business setting, the rough average number of files per system has grown from 13k to 225k between 1998 and 2009 (Table 1.2).

In forensic analysis of storage systems, a common early step is to triage files for inspection. Many files can be recognized in the file systems by comparing content hashes to hash lists [15]. The most common recognized-file scenarios are using hashes to look for known contraband data [28], or to suppress known files bundled with software [71, 34]. Intuition suggests that the software association list would recognize most of file systems' software content. However, a measurement of a substantial software-files list showed only a 32.7% recognition rate among 36 million files from over two thousand used hard drives [61]. If the software files were all recognized in the hash list, then 67.3% of millions of files would have to have been created by people, which is unlikely; or they would have been created as software-generated artifacts of user activity. Alternatively, the premise that all software files were recognized can be false, and it is when considering the definition of a "software file" on a system. In that study, the hash list is the National Software Reference Library's (NSRL) Reference Data Set (RDS), a collection of content hashes of files derived from installation media [71]. One suspected reason behind the low recognition rate is the semantic gap between the files the NSRL lists, and the files put on systems when software is installed and used [61, Section 7].

1.2 Statement of the Problem

To date, file system growth has been an ever-increasing hindrance in analyzing computer storage. Considering where those files come from creates a new opportunity to understand a computer. It should be possible to discover which programs have run on a system, by associating files with their likely creating program. A manifest of software used on a computer simultaneously describes its purpose up to the time of inspection, and also indicates types of data one may expect to find when inspecting its contents. Thus, the millions of artifacts on machines can be turned to investigators' advantage.

The general problem addressed in this dissertation is taking the forensic features of a subject machine, and recognizing what software was ever run on that machine based on the present features. This model can only work if the features have been recognized before, and more importantly, if the features have some notion of discriminatory strength—effectively, a measure of distinctness. The Registry's feature count quickly moves this problem outside the ability of unassisted manual analysis. When taking preliminary measurements for this document, installing and removing Adobe Reader produced 54,000 total changes in the Registry, counting changes between four disk snapshots. The Registry entries added by a program have the further challenge of being potentially deceptive if happened upon; for example, consider this Registry key from a Windows XP machine:

- `\$$$PROTO.HIV\Software\Netscape\Netscape Navigator\Suffixes\application/msexcel`

The product name "Netscape Navigator" appears in the path. An investigator who only descends into the Registry to that middle key may conclude that Netscape was on the system. However, the key was created by installing Microsoft Office, Professional Edition 2003, and no Netscape product was placed on the system. These inter-product references could confuse signatures for software products made without the context of signatures for many other products.

Document search presents a model for this task, fitting constellations of features to the cause of their instantiation. This thesis shows how to apply document search to identify software presence. A subproblem is assigning many relevancy scores to artifacts—*e.g.*, how relevant a file is to any program, or whether it means nothing for any software presence.

1.3 Purpose of the Study

There are many resources a computer has to identify the programs on it. For instance, Microsoft Windows includes an Installed Programs feature, for listing and removing software. However, such resources can be incomplete, or deliberately evaded if a user or program is intent on covering their tracks. It is worthwhile understanding what the forensic footprint of conspicuous software is—but to learn that, we must necessarily learn what normal activity is as well. Hence, a side effect of this research is coming to understand the forensic footprint of benign background noise, to be sifted from software-indicative data.

1.4 Theoretical Framework

A computer will have a set of programs, P, that have affected it. Each artifact a is associated with none, one, or many of the programs $p \in P$. With A as the set of artifacts, this implies an $|A| \times |P|$ artifact-to-program association matrix, where entry (a_i, p_j) is some score of a_i's relevance to p_j. This dissertation applies information retrieval problem design to file system analysis, in order to populate that matrix.

To assemble the sets of artifacts of a program p_j, this dissertation inspects virtual machines instrumented to have only that program used on it after a baseline operating system is installed. The inspection follows the design pattern of file system differential analysis [21], following an algorithm to enumerate changes between two states of a storage system. The virtual machine snapshots readily provide states for differencing, and also provide a way to relate all of the change sets together, using the snapshot histories as a relational data structure.

The change sets provide unwieldy numbers of artifacts, only some of which are likely related to a program. Document search takes the change sets and assigns scores of the association matrix. Simple counting is one type of search, but many other variants exist [74].

1.5 Thesis Statement

By analyzing differences between two snapshots of the Microsoft Windows Registry, it is possible to reliably infer the installation, use, or removal of a specific application, using signatures automatically generated from a training set of disk images.

1.6 Importance of the Study

The process that most immediately benefits from this research is forensic triage. Automated production of a manifest of software ever used on a system would be a boon to investigators, who today begin with little usage context on each system they see. With separated installation and execution signatures, software run without being installed can be flagged early in analysis. For instance, recognizing execution of a web browser with no signs of installation may suggest some other device like a USB thumb drive was used to house a portable browser instance. The investigator could then review whether they have such a thumb drive in their possession.

This research makes it possible to use a Windows system's Registry files to report on software present and executed on the system, by using software signatures made with document search. Those signatures can become a resource supplementing already-available known-file sets. Combined with the National Software Reference Library's (NSRL) Reference Data Set (RDS), this research closes the semantic gap between installation media files and files expected to be found on subject machines.

1.7 Scope of the Study

This dissertation assumes the forensic inspection pattern of *post-mortem* storage analysis. In post-mortem analysis, a subject machine is inspected after being powered off, and no further changes are expected to occur within the machine. Further, this dissertation only analyzes one class of artifacts: path names extracted from the Windows Registry. The analysis does not include the contents of Registry values.

The study can also be performed on the Registry as extracted from RAM, but that is left to future work. The search techniques can then further expand to generic appearances of forensic artifacts. However, significant tailoring of the same design pattern will be needed for each class of artifact.

This research does not inspect presence of software after attempted deletion. However, some of the data used in the study has been used for that purpose [29], though with a different class of artifact, file sector hashes.

Last, the design of this study focuses on optimizing binary classifier performance against fixed inputs, *i.e.* against two corpora of virtual and physical machine states. The data considered do not allow for statistical claims to be made, because the tested data are drawn only once from the random population of machine states. Hence, the results of this study are generalizable to the extent that these machines are representative of the total population of computer storage states of systems running Microsoft Windows operating systems since the Registry was first introduced in Windows 95 [26].

The subject machine created specifically for this study is only representative of machines insofar as it has a collection of applications installed, with some run once. Most desktop systems that see normal use will have more applications installed, which will be run many times with more of their functionality exercised by the end user.

However, the second data set studied—“M57-Patents” [72], described further in Section 3.2.3—is more representative of realistic computer use, as the machines were used over the course of a calendar month by real people, though they were enacting a fictional scenario. Consistency in evaluated performance between these two data shows that the methodology can be generalized. This study's scope does not include a statistically strong evaluation. It is left to future work to apply to larger set of real data, *e.g.* the Real Data Corpus [23].

1.8 Definition of Terms

This dissertation combines two topic areas: document search, and storage system differential analysis [6, 21]. This thesis introduces new terminology to describe the behaviors of the Windows Registry in Diskprint data inter-relationships.

1.8.1 Document search

The analyses of this work are based on a search engine that uses the Vector Space Model (VSM) of document search [10, 74], a mechanism to measure similarity of documents and queries by representing them as term vectors. There are at least millions of possible, applicable variants of the VSM, that permute term weights, document lengths, and other factors. However, Zobel and Moffat exhaustively evaluated over 100,000 effective variants on English document corpora with queries and relevancy provided, and found "no component or weighting scheme was shown to be consistently valuable across all of the experimental domains" [74, Conclusions]. Hence, we select a "Stock-standard" VSM variant [74]: term-frequency, inverse-document-frequency weights (TFIDF) with cosine similarity. Future work can include seeking an optimal weighting scheme among the parameter space of Zobel and Moffat.

This section introduces search nomenclature and constructions that will be translated in later chapters to search forensic feature sets. As Zobel and Moffat enumerated a significant parameter space of constructing vector space models, the formulas quoted here will use their symbols and definitions.

Table 1.3: Symbols of document and term statistics used to define vector space models. For nomenclature consistent with the literature, definitions are as in Zobel and Moffat [74], some verbatim.

Symbol	Definition
N	The number of documents.
t	A term.
d	A document.
q	A query.
$f_{d,t}$	Absolute term frequency—*i.e.* count—of a term t within a document d.
f_t	The number of documents containing term t.
$f_{q,t}$	Absolute term frequency—*i.e.* count—of a term t within a query q.
$\mathcal{D}$	The set of documents.
$\mathcal{T}$	The set of distinct terms known to the vector space model.

Table 1.4: Symbols of components used to define vector space models, as various functions of the statistics defined in Table 1.3. For nomenclature consistent with the literature, definitions are as in Zobel and Moffat [74], some verbatim.

Symbol	Definition
$S_{q,d}$	The similarity of a query q with a document d.
w_t	Term weights, which are variants of inverse document frequency.
$w_{d,t}$	The weight of a term t within a document d—*i.e.* document-term weights and query-term weights.
$r_{d,t}$	Relative term frequency—*i.e.* weighted count—of a term t within a document d.
W_d	Length of document d.
$\mathcal{T}_{q,d}$	The terms in common between a document and query; formally, $\mathcal{T}_q \cap \mathcal{T}_d$.

A VSM *search engine* returns a set of *documents* corresponding to a user *query*, both of which are comprised of *terms*. Queries are measured against a *corpus* of documents ingested by a search engine. The engine returns the documents relevant to a query, ranked by their "nearness" to the query. "Nearness" is measured by treating documents and queries as vectors in $|\mathcal{T}|$-dimensional space, where $\mathcal{T}$ is the set of terms. The vectors' elements are weights, according to frequency of the terms and inverse frequency of the terms among corpus documents. Tabulatory statistics of the corpus, whose symbols are in Table 1.3, are combined by the weighting scheme, which uses the symbols in Table 1.4.

The *document-term weights* $w_{d,t}$ within the TFIDF matrix are defined as:

$$w_{d,t} = r_{d,t} \cdot w_t \tag{1.1}$$

Where the *inverse document frequency* w_t is:

$$w_t = \log{(1 + \frac{N}{f_t})} \tag{1.2}$$

And the *relative term frequency* $r_{d,t}$ is:

$$r_{d,t} = f_{d,t} \tag{1.3}$$

The length of a document W_d is:

$$W_d = \sqrt{\sum_{t \in \mathcal{T}_d} w_{d,t}^2} \tag{1.4}$$

The length of a query is also usually defined as in Equation 1.4. However, in this dissertation W_q needs to deviate from that typical definition. The typical definition iterates over $\mathcal{T}_d$, whereas here the query length iterates over $\mathcal{T}_q \cap \mathcal{T}$, clarifying that $\mathcal{T}$ is the set of terms known to the search engine, without any term associativity predictions typically available to unknown natural-language terms [35]. This restriction is due to the unusual situation of having queries with a high likelihood of having terms that were not in the training data. The search engine needs to ignore these unknown query terms, because their inverse document frequency according to the search engine's knowledge is undefined, needing to divide by the term frequency of 0. This is an issue that normally does not arise in natural language search, which has estimation techniques for unknown terms, such as part-of-speech estimation [67] and translation [9].

Finally, the *combining function* is the *vector cosine*:

$$S_{q,d} = \frac{\sum_{t \in \mathcal{T}_{q,d}} (w_{q,t} \cdot w_{d,t})}{W_q \cdot W_d} \tag{1.5}$$

A cosine value, or similarity, of 1 is a perfect match, while 0 is orthogonal, meaning no match at all. Another reason for only analyzing query terms in $\mathcal{T}_q \cap \mathcal{T}$ is maintaining a consistent interpretation of similarity of totally-dissimilar documents: Zobel and Moffat choose their similarity functions so when no terms are in common between a query and a document, the similarity is zero [74, Section 2].

Natural language search engines follow a few common strategies to reduce search index size and make terms recognizable to the engine. First, common terms are often ignored from documents and queries, by creating a *stop list* of terms to discard from all engine operations. This removes significant burdens from the index, *e.g.* by not tracking how many times “the” appears within a document. There are several strategies used in natural language search to construct stop lists. A stop list can be the most common terms among some language corpora. For specific classification tasks, like finding terms that indicate a message is spam or not, each term can have its information gain measured, and be added to a stop list if the information gain is too low [10]. However, information gain is suited for one classification task at a time. In this thesis, information gain calculations would be redundant with TFIDF, so they are not performed; but the most-common terms are identified and used in stop lists.

To address variants of terms that semantically resolve to a common idea, engines will also *stem* terms in documents and queries. Stemming is converting terms into more basic forms, so they can be detected in spite of light effects like misspellings or conjugations. For example, a common topic for documents containing the words “hunting” and “hunter” would be derived by stemming the two to “hunt.” A phonetic example of stemming is doing human name search by Soundex encoding, a phonetic spelling for names according to English pronunciation [62].

1.8.2 File systems and the Windows Registry

Computer storage systems have centered for decades around the *hierarchical namespace* design [63]. A disk is divided into *partitions*[1], usually one to four, and each partition is given a file system. Each file system then has a *superblock* with basic metadata like a label and pointer to a root directory. From the root directory, the rest of the file system attaches in a tree structure, or at least a directed acyclic graph if files are allowed multiple names.

In brief, the Registry can be considered a special hierarchical file system [40]. Many of the Registry's basic elements align straightforwardly with file system terminology:

Hive A hive is a collection of Registry items. It is analogous to a partition on a hard drive. A hive contains what amounts to a file system.

Hive file A hive is stored as a regular file in the NTFS file system. A typical Windows system contains several hives. Table 1.5 shows hives that appear in a typical Windows 8 system.

The global Registry namespace The Registry tree structure acts like a Windows file system, where partitions are presented at letter-colon points (*e.g.* "`C:`"). Hives mount their file systems at particular points in the namespace.

Cell A cell is analogous to a file, much in the same spirit as some file systems treat all file system objects as files, be they directories, regular files, soft links, *etc.* In the

[1]For historical reasons, the terms "Volume" and "Partition" are often used interchangeably.

Windows Registry, cells can be *keys* or *values*.

Key Keys are cells that can contain other keys or values. They are analogous to directories. Each key also has a timestamp of when it was last modified.

Value Values are analogous to regular files. Values have an explicit type (such as integer, or string list), but lack much of the metadata that file systems provide files. The type is known to be overloaded as an additional arbitrary data field, in at least one case by the Windows operating system: the SAM hive stores a numeric user identifier as the type of the default value of the key `\SAM\SAM\Domains\Account\Users\Names\`*`username`* [31].

Timestamp While some values store timestamps as their data, values themselves do not have any timestamp fields. Keys store their own last-modified time.

Slack Like some file systems, the Registry does not erase content on receiving a deletion command. The content is instead marked as being unallocated and left in place, awaiting reclamation. Hence, one can recover deleted cells [66]. This dissertation does not make use of deleted content, but the techniques presented here could be expanded to use deleted Registry cells.

Other references on Registry structures [66, 44] and analysis [8] are available. For this dissertation, it will suffice to understand hives, cells and values. Timestamps and slack space analysis provide future opportunities for new or improved signature construction unrealized here.

Table 1.5: Some typical hives of a Windows 7 system. These hives are usual analysis subjects in forensic investigations that require Windows Registry inspection [7, Table 4.1] [8, page 18]. A more extensive list of hives is given in Tables 3.8 and 3.9.

File system path suffix
`Users/`*`username`*`/AppData/Local/Microsoft/Windows/UsrClass.dat`
`Users/`*`username`*`/NTUSER.DAT`
`Windows/System32/config/COMPONENTS`
`Windows/System32/config/SAM`
`Windows/System32/config/SECURITY`
`Windows/System32/config/SYSTEM`
`Windows/System32/config/SOFTWARE`

1.8.3 Diskprints and lineage sequences

This work measures the storage effects of software as it is used on systems. Differential analysis [21] identifies what has changed from one system state to another; but the design and choice of compared states is as necessary as the ability to compare the states. This section addresses the basic design of the *Diskprint* data, the training data for the search models. The Diskprint data provide a corpus of machine states related by state lineage, which leads to *sequential forensic analysis*, an extension of differential forensic analysis. Where most forensic analysis today considers a single system state, differential analysis considers the changes of the entire computer from one state to another. To borrow calculus terminology, it is the "first derivative" of a forensic image with respect to time. Sequential analysis studies the "second derivative:" similarities and dissimilarities between change sets. By defining a data structure for forensic sequences, this work enables automated differential analysis, as system states are better organized for automated comparison.

1.8.3.1 NSRL Diskprints

The data on which the search models are trained are derived from the National Software Reference Library (NSRL) *Diskprint* project [65]. The NSRL provides metadata summaries of gathered, purchased, and donated software. To date, the NSRL has released its file content summaries as their Reference Data Set (RDS), a collection of metadata—particularly hashes—of files derived from installation media. As of edition 2.51, the RDS currently houses approximately 23,000 software packages. The software backing this hash set has not been installed, so there is a semantic gap between the data backing the RDS and the data one encounters in an investigation.

In search of a better software identification process than hashing installation media files, the NSRL team began the *Diskprint* project [65]. Diskprint-derived data come from installing software from the NSRL in a controlled virtual environment, measuring machine state at consistent times in an application's lifecycle. These measurements provide opportunity to take forensic measurements of RAM, storage, and network data.

A *software diskprint*[2] is a captured sequence of states of a machine, typically of a virtual machine. Each state contains the snapshot contents of the virtual machine, and is also bundled with a network capture since the last snapshot. The snapshots are taken according to progress on a *software lifecycle.* For generic software signatures, the lifecycle has these states to remain generically applicable to software packages,

[2]*Software diskprint* will often be simplified within this work to *diskprint.* When capitalized, "Diskprint" will refer to the NSRL project.

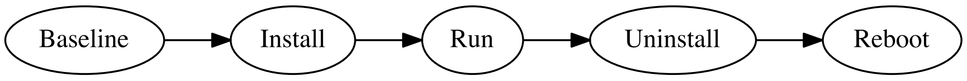


Figure 1.1: The software lifecycle captured by a diskprint. Each step has at least one virtual machine snapshot taken. The search models in this thesis are built on only the last snapshot taken in each lifecycle phase.

illustrated in Figure 1.1:

1. *Baseline* An operating system has been installed, and is ready for use.
2. *Install* An application has been installed.
3. *Run* The application has been run. In practice, diskprints often break *Run* into an *Open* step, one or more *Function* steps to deal with mandatory software actions, and a *Close* step signifying the software has been run once at least minimally.
4. *Uninstall* The application has been removed.
5. *Reboot* The machine has been re-booted.

In creating some of the diskprints, multiple snapshots were taken while in a particular lifecycle phase, because there was some decision point the diskprint creator wanted to capture. The analysis in this thesis is only based on the last snapshotted state of each point in the software lifecycle. A training sequence derived from a diskprint sequence will only be the states “Baseline,” “Last Install,” and “Last Close,” where “Last” acknowledges that a state may have taken several decision-point snapshots to

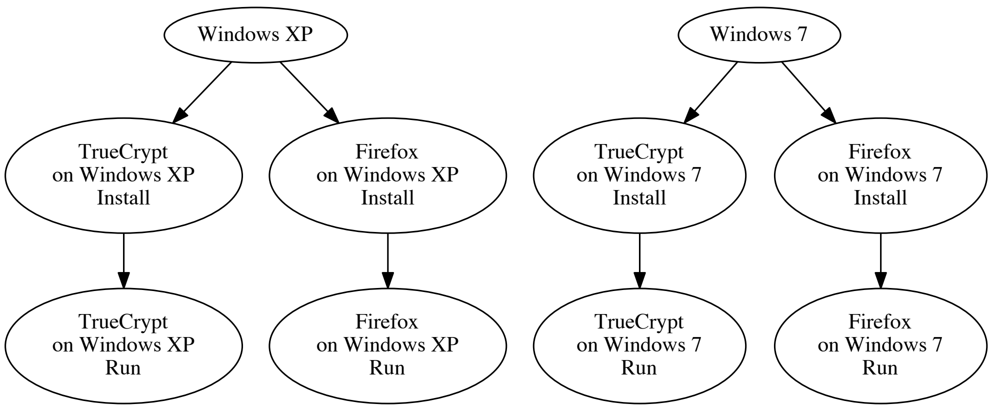

Windows XP
TrueCrypt on Windows XP Install
Firefox on Windows XP Install
TrueCrypt on Windows XP Run
Firefox on Windows XP Run
Windows 7
TrueCrypt on Windows 7 Install
Firefox on Windows 7 Install
TrueCrypt on Windows 7 Run
Firefox on Windows 7 Run

Figure 1.2: A sample lineage graph, showing the histories of two applications run on two baseline virtual machines. Edges are in the direction of progress through the software lifecycle (Figure 1.1).

capture.

1.8.3.2 Lineage graph and sequence

A *lineage graph* is a *forest*, or set of graph-theoretic trees, relating diskprint states to one another. Each tree represents the histories of a system that share a common baseline, with directed edges denoting ancestry. The Diskprint data induce a lineage graph scoped to applications installed on virtual machines. Figure 1.2 shows the combinations of application and operating system by installing TrueCrypt and Firefox on Windows XP and 7.

The workflow that analyzes the lineage graph frequently considers the *nodes* and *edges* of the graph. Each node represents the state of a virtual machine, including its network activity since the last state, its disk(s), and its RAM. An edge of the lineage graph points to a node's immediate ancestor as collected by the Diskprint process.

A *lineage sequence*—which will often be simplified throughout this document to "sequence"—is a subset of nodes of the lineage graph, with a set of edges defined to join the nodes. The subset preserves the lineage ordering, but may "skip" some of the recorded states for the sake of reducing state-change granularity.

1.9 Summary

Most user actions taken on a computer will affect the state of the storage system. This dissertation uses document search to weigh the importance of those effects, in order to recognize software usage. Search also provides a mechanism for determining the importance of every artifact relative to each software presence query. This weighting is a novel mechanism for weighing the millions of artifacts encountered in investigations, and is human-scalable at investigation time.

This work makes the following contributions:

- A novel forensic application of document search.
- Design of the Diskprint controlled corpus of software execution.
- An extension of forensic differential analysis, to a workflow that can be derived from the lineage graph of an evolving training corpus.
- A strategy for characterizing forensic features, from the Windows Registry in particular, as important to some software package or important to none.

Chapter 2

Review of the Literature

This chapter introduces the storage system analysis language and analytic technique that are used for developing software signatures, Digital Forensics XML and differential analysis. Other works on software signaure development, most of which involve manual artifact classification, are covered for comparison.

2.1 DFXML and hierarchical storage analysis

There are many types of storage analysis tasks that do not require access to raw storage data. For instance, storage system utilization, namespace depth surveying [1, 13, 38], and timelines need only file system metadata. Tree synchronization [21] and known file recognition [15] need only minimal file content summaries. These analyses were key motivations cited by Garfinkel in the design of Digital Forensics XML (DFXML) [17], an object model and XML syntax for describing storage systems. DFXML functions as an intermediary analytic format, fit for performing file system metadata analysis and exchanging results, without requiring access to the original disk image. This work uses a child language of DFXML developed to perform the same role for the Windows Registry [40].

DFXML is generated by a metadata extraction utility that walks a storage system. The initial generator, the userspace tool *fiwalk*, is a wrapper around storage system analysis functions provided by The SleuthKit [22]. *fiwalk* creates a storage metadata manifest of a disk image, without requiring mounting the disk and potentially exposing malformed or malicious file system content to the analysis machine's kernel.

The DFXML model enumerates metadata for partitions and files. Its initial design recorded the following:

- Partitions had file system type and physical location information recorded.
- Traditional file metadata, such as that found in inodes, was recorded in `<fileobject>` elements. The traditional file metadata was augmented with some NTFS specifics commonly used in The SleuthKit; and with content information, including libmagic information, cryptographic checksums, and data locations. Location data for file contents were recorded in `<byte_runs>` elements for each file.
- Provenance of a DFXML file was included within the header, including information about the generating utility, input file, and command line. From these data, the DFXML file could be recreated at a later date if necessary.

2.2 File system differencing

The key to deriving application footprints from post-mortem storage analysis is understanding what changes took place between two times. The general problem form here is *differential analysis* [17], observing the differences in two *images*, measured in counts and attributes of *features* within the images. For example, in disk-level storage analysis, a disk image would be an image, a file a feature, and a timestamp an attribute. These terms are as defined by Garfinkel *et al.* [21]. In Registry analysis, we analyze the same dimensions of Registry cells.

File system differencing is a multi-dimensional problem. The objective is to find files that were created, deleted, modified, or renamed. Post-mortem file system analysis has additional complicating factors. A file found in a disk at time t_0 may be found in the same disk at time t_1, under a different identifier number, directory, name, partition, data byte location, metadata byte location, or allocation status. Each of these properties serves to help identify a file, and each can be independently mutated, confusing the matching process. Fortunately, the situation is more straightforward for the Registry.

File system differencing is a specialized instance of a general, cross-domain differential analysis strategy [21]. The general strategy is similar to set-differencing features found in images. Features have an identity attribute, potentially multiple names, a location within the image, and other attributes. Features are matched on identity and name, and once matched can be further analyzed for each changed attribute if desired. In storage analysis, this strategy applies straightforwardly for reporting changes of disk partitions without inspecting partition content. For most file systems, however, a difference set of two images cannot be produced by simply assembling a list of names and a list of inodes. The general assembly and incremental removal to match files and their names is fraught with complications from file-matching ambiguities that storage systems typically present. Fortunately, these complications are absent from the Registry.

2.2.1 RegXML and Registry differencing

The RegXML language represents the above metadata and value content, in an XML syntax that overlaps with DFXML [40]. RegXML was designed to behave much like DFXML, so the Registry could be analyzed similarly to file systems. The RegXML programming interface is the basis of the Registry analysis software for the experiments in this work. The programming interface provides a similar differential analysis framework as with file systems.

Differencing Registry hives is similar to differencing file systems, but generally less complicated because:

- Registry hives have less metadata per cell than file systems have per node.
- Registry cells are not generally moved from one location to another.
- Registry keys are not renamed.

Thus, Registry differencing can generally be done with set subtraction. This is mainly due to the simple identity mechanism of Registry cells.

Garfinkel *et al.* implemented differential analysis in a program, *idifference.py* [21]. The analysis in this dissertation is built on a variant of that tool, *rdifference.py*, an implementation of Registry differencing at the hive level. When run by itself, the hive differencing program *rdifference.py* reports:

- New and deleted hive cells, including cells that have precisely matching full paths (*i.e.* a cell being fully duplicated, including name).
- Values with modified content or type.
- Keys with changed modification times (mtimes), from which one may partially infer a changed value's mtime.

When used as a library, *rdifference.py* provides the sets of added, removed, and matched cells for further analysis.

This dissertation only considers Registry cells added from step to step. Set differencing suffices to compute these changes, so the topic of differencing Registry contents will not be explored further in this document.

2.3 Software signature development

File system metadata is a frequent, and in many circumstances the best or only, data for determining how a system was used. Early metadata-based forensic inspection includes recognizing when executable files' checksums had changed from previously recorded states, as in *Tripwire* [33], an early intrusion detection tool.

File system metadata provides many dimensions of data one can use to devise behavioral or software signatures. Geiger evaluated anti-forensic tools for their completeness in removing forensically interesting file system artifacts [24]. One result of his research was a set of signatures of tool usage, based on his manual observation of patterns in file names and contents. Geiger reported these signatures in English, but most of the software traits could have a pattern-recognizer written from the description, such as file name and extension patterns. Geiger's signatures cover a broader scope of data than inspected in this dissertation, but relied on a human to determine the worthwhile patterns, whereas this dissertation uses document search to develop signatures.

Davis *et al.* approached the problem of developing software signatures, sharing developed signatures of data concealment programs—encryption, steganography, and data tools—by inspecting storage artifacts after distinct stages of use. They created and tested signatures of tool installation and execution by deploying tools on physical machines, with each deployment starting from a consistent baseline operating system image [12, Section 5]. After installation, execution, and removal, they would shut down the system and image the drive. There was no report of repeating an action to catch behavioral variances. The Diskprint data performed similar minimal data generation actions, but the data would differ in two significant ways using virtual machine snapshots instead of physical machines. First, Windows shutdown artifacts would not be integrated into each data capture. Second, artifacts in virtual machines would have some likelihood of not being flushed from RAM.

The methodology of Davis *et al.* included unspecified strategies for identifying significant artifacts, with overlapping artifacts like shared libraries handled by allowing a user to specify a scalar “Confidence” threshold. Confidence was calulated based on an unspecified function of matched artifacts. Tools were reported as installed or executed by their signature-matching engine if the signature match surpassed the user-requsted threshold. This dissertation also reports software installation and execution, but uses document search to calculate artifact relevance scores and match sofware signatures by artifact sets, and draws benefits from repeating software lifecycle phase captures.

Kälber *et al.* produce “file system fingerprints” of software usage, using only timestamps from file system metadata, and an automated repetitive state capture framework [32]. Their goal was to capture the effects of specific user actions, such as email and instant messaging activity, so the software signatures are an intermediary result towards their goal. Their signatures comprised of sets of files expected to have clustered timestamps if an action were taken. They constructed the signatures by performing an action in a virtual machine, creating a GUI script that could repeat their interface actions, and then running the script and capturing system state one hundred times. The files named in the signatures were the set-intersection of files with updated timestamps, among only the one hundred repeated operations. The Diskprint data don’t enjoy the same depth of repetitions, but they do have a greater breadth of software selection feeding the search models. The intersection among repetitions is one option that the search models use to combine grouped signature elements, but other options are explored in Section 3.3.3.5. Last, Kälber *et al.* had a difficulty with files including varying patterns, particularly the Mozilla pattern of using a random-character profile name in the file

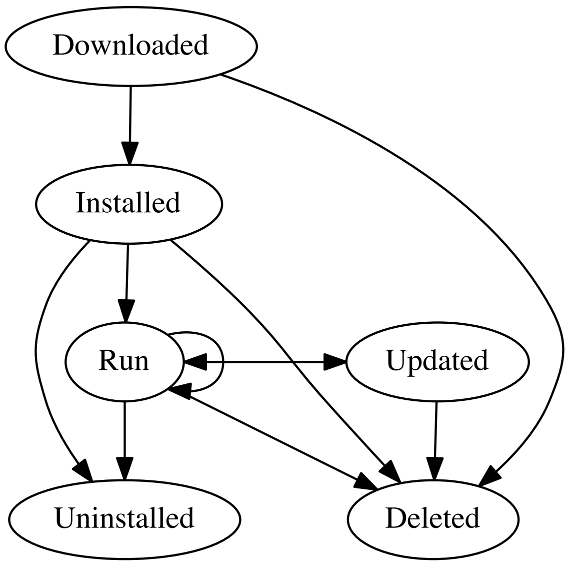


Figure 2.1: A previously proposed software lifecycle model. Reproduced from Davis *et al.* [12].

path: Those files could not be used in a signature without the investigator providing a wild cards framework. This dissertation addresses that problem by analyzing path components instead of whole paths, further described in Section 3.3.3.4.

2.4 Software lifecycle

The software lifecycle the Diskprint project takes focuses on a single version of software. Other work in signature development acknowledges the evolving states of software, particularly around updating. For instance, Davis *et al.* [12] use the software lifecycle depicted in Figure 2.1. Unlike the linear Diskprint lifecycle of Figure 1.1, this model adds an "Update" state.

Davis *et al.* did not address the update state beyond that figure, but there is a larger effort around representing the identity of software, including deployed state, vendor updates, and independent patching, called Software ID ("SWID") Tags [69]. SWID tags represent an installed instance of software with more precise version information than a version string, by incorporating authoring information; manifests of installed files and Registry entries; and supplemental product references, like patches. The granularity that patch references provides can help future work in software signatures distinguish behavioral artifacts between different versions of software, which is likely to matter if one is scanning for known-vulnerable software. If software vendors, package managers, and compiler tool chains move to automatically produce and provide SWID, then one of the questions posed in this thesis—was product x installed—will be easier to answer on systems in the future. However, SWID does not necessarily make detecting records of software execution easier; nor does it automatically help identify presence of software that exists today, unless signatures are developed, such as by methods inspected in this thesis.

Chapter 3

Research Methods

3.1 Research Design

This research starts with sequences of virtual machine snapshots, and from them derives software signature search engines. This first requires a framework to process related virtual machine states, in order to get difference data to ingest into a search engine. The design of the search engine is extensively parameterized, due to a number of design decisions that arise when considering how to combine the difference data into signatures. Ultimately, the search engine is converted from a Vector Space Model to a *Signature Searcher*, which passes a Registry to an underlying Vector Space Model, and uses signature similarity scores to return a set of triples: $(Application, Applicationstate, Present)$, with the third element a Boolean response.

Overall, the evaluation strategy is to demonstrate the best point in the configuration parameter space for Signature Searchers, judging by the Searcher correctly identifying software presence. The most effective combinations of parameters are sought by a full-factorial experiment [5] that tests all combinations of parameter values. The evaluation operates on two data sets. First, a controlled virtual machine has software installed, with snapshots taken after individual package installations and executions are complete. Search results from each machine state can then be compared to the *ground truth* for that state, where the ground truth is a set of quadruples: $(MachinestateID, Application, Applicationstate, Present)$.

By comparing Searchers' reports of software installation and execution with the ground truth history of software usage, the Seachers are given binary classifier evaluation, measuring effects on precision and recall. At times, precision and recall will be combined into F1 (their harmonic mean, $\frac{2pr}{p+r}$), as a single-number ranking measure of the Searcher's "Correctness." There will also be some consideration of counts of signatures produced per model. The ground truth is explicitly enumerated, so a Searcher response on an application-state presence in a machine state outside ground truth is discarded from the evaluation.

After evaluation against the control virtual machine's states, the Searcher operates on an independently-produced corpus of disk images, the M57 "Patents" scenario [72], attempting to reproduce a software presence report from the research literature. This dissertation compares against Roussev *et al.* [60], which also analyzed the M57 Patents scenario. The Signature Searchers make use of the Windows Registry, while Roussev *et al.* made use of the file system, captured system RAM, and in some cases network captures. The Signature Searchers produce the same application presence results, using only one forensic data source of the examination subjects.

3.2 Participants

The experiments of this thesis include three different data sets. First is the training data, the Diskprint data set produced by the NSRL team [71]. The first Signature Searcher evaluation is run against a Windows virtual machine instrumented specifically for this study. That machine has a set of applications, with snapshots taken as software was installed and executed. The second evaluation runs the top Signature Searchers according to the instrumented machine's states, and sees if they are equally performant against the M57-Patents corpus [72]. Though there are in total four machines used in the evaluation—not counting the training data—they are grouped into two sets, according to their sources of ground truth. The instrumented machine has a fully enumerated set of application installations and execution at each snapshotted state. The M57 corpus has an independently-produced set of application installations and execution.

3.2.1 Training data: NSRL Diskprints

Diskprints were created on baseline operating system states that contain a primary, administrator-level user that performs the software interactions; and a secondary user, not to be directly used. The secondary user is to help identify artifacts from software that purposefully affects each individual user's configurations, and artifacts that include some small pertuburations. For example, some Mozilla software creates paths in the user data directory that include a random string in a directory name.

Table 3.1: Application diskprint tallies, with versions as reported by NSRL practice, *e.g.* as recorded in software physical packaging [71]. A diskprint is a sequence of snapshots that capture a virtual machine as it installs, runs, and uninstalls an application. A number greater than 1 indicates a diskprint was repeated, with the same user actions taken with the same application on the same operating system. No prints other than a baseline were produced for 64-bit Windows 8. Continued in Table 3.2.

		Count on OS & arch.					
		XP	Vista		7		8
Application	Version	32	32	64	32	64	32
7-Zip	9.2	3	0	0	0	0	0
Adobe Acrobat Reader 3.0	Copyright 1995-1996	0	1	0	0	0	0
Adobe Dynamic Media Solutions	2.2002	0	1	0	0	0	0
Adobe Photoshop Elements 10	c. 2001-2011	0	0	0	1	0	0
Adobe Photoshop Elements 12	c. 2001 - 2013	0	0	0	0	0	1
Adobe Photoshop Lightroom 4	c. 2012	0	0	1	0	0	0
Brother HL-2170W	HL-2170W	3	0	0	0	0	0
Eraser	6.0.10.2620	0	0	0	1	0	0
Faronics Deep Freeze Standard	7.10.020.3176	0	0	0	1	1	0
Firefox	32.0.2	3	0	0	3	3	3
Google Chrome	28.0.1500.95	1	0	0	1	1	0
Google Chrome	4	0	1	0	0	0	0
HxD Hex Editor	1.7.7	0	0	0	1	0	0
Invisible Secrets	2.1	3	0	0	1	0	0
Limewire Basic	4.09.39	0	1	1	0	0	0
Microsoft Flight Simulator 2004 A Century of Flight	2003	0	0	0	1	0	0
Microsoft Office Home and Student 2010	2010	0	0	1	0	0	0
Microsoft Office Professional 2007	Version 2007	0	0	0	1	1	0
Microsoft Office Professional Edition 2003	2003	1	0	0	1	1	0

Tables 3.1 and 3.2 list the applications that have had diskprints produced, along with the number of times these applications were printed. The numbers in the remainder of this document stem from these applications.

Table 3.2: Continuation of Table 3.1.

		Count on OS & arch.					
		XP	Vista		7		8
Application	Version	32	32	64	32	64	32
Mozilla Firefox beta	19.0b2	1	0	0	1	1	0
Mozilla Thunderbird 2	2004-2007	3	0	0	0	0	0
Norton AntiVirus 2012 with Antispyware	c. 2011	0	0	1	0	0	0
Python	2.6.4	1	1	0	0	0	0
SDelete	1.61	0	0	0	1	1	0
Safari	5.1.7	1	0	0	1	1	0
Skype	6.1.0.129	1	0	0	1	1	0
StreetFinder Travel Navigation Software	c. 2003	0	1	1	0	0	0
TeamViewer	9.0.25942	1	0	0	1	1	0
TrueCrypt	6.3a	3	0	0	0	0	0
TurboTax Deluxe Plus State	5	0	1	1	0	0	0
TurboTax Premier For Tax Year 2013	c. 2013	0	0	0	1	0	1
Ultimate Packer for eXecutables for Windows 32-bit	3.09	0	0	0	1	1	0
WinZip 17 Pro	c. 2012	0	0	0	3	3	0
Winrar	5.00 Beta 6	0	0	0	1	1	0
Wireshark	1.8.0	0	0	0	3	1	0
World of Warcraft	c. 2004	0	0	0	1	0	0
XP Advanced Keylogger	2.1	1	0	0	0	0	0
Yahoo Messenger	11.5	0	0	0	1	0	0
mozilla Firefox 2	Copyright 2004-2007	0	1	1	0	0	0

Table 3.3: Appearance order of applications' installation and running on the experimental subject virtual machine. The "1st appearance" column is the numbered snapshot of the virtual machine, where the baseline OS was snapshot 0.

Application	Version	Lifecycle phase	1st appearance
XP Advanced Keylogger	2.1	Install	2
Wireshark	1.8.0	Install	3
HxD Hex Editor	1.7.7	Install	4
Invisible Secrets	2.1	Install	5
Mozilla Firefox beta	19.0b2	Install	6
Python	2.6.4	Install	7
Mozilla Thunderbird 2	2004-2007	Install	8
TrueCrypt	6.3a	Install	9
Microsoft Office Professional Edition 2003	2003	Install	10
Winrar	5.00 Beta 6	Install	11
SDelete	1.61	Install	12
Winrar	5.00 Beta 6	Run	12
WinZip 17 Pro	c. 2012	Install	13
WinZip 17 Pro	c. 2012	Run	13
Mozilla Firefox beta	19.0b2	Run	13
Eraser	6.0.10.2620	Install	14
Eraser	6.0.10.2620	Run	16
TrueCrypt	6.3a	Run	17
Invisible Secrets	2.1	Run	18
SDelete	1.61	Run	19

Table 3.4: Ground truth of applications installation and running on the experimental subject virtual machine. Because document grouping and software version matching affect the definition of ground truth, other variants of this table exist, but may be impractical to typeset. Here, document grouping is by application, with each application version considered distinct. The numbers in the header row are the snapshot number of the experimental subject machine, where in snapshot 1 no application had been installed or used, snapshot 2 had the Keylogger installed, *etc.*

Application	I/R	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Keylogger	I		X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Wireshark	I			X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
HxD	I				X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Inv. Secs.	I					X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Firefox	I						X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Python	I							X	X	X	X	X	X	X	X	X	X	X	X	X	X
Thunderbird	I								X	X	X	X	X	X	X	X	X	X	X	X	X
TrueCrypt	I									X	X	X	X	X	X	X	X	X	X	X	X
Office	I										X	X	X	X	X	X	X	X	X	X	X
Winrar	I											X	X	X	X	X	X	X	X	X	X
Winrar	R												X	X	X	X	X	X	X	X	X
SDelete	I												X	X	X	X	X	X	X	X	X
Firefox	R													X	X	X	X	X	X	X	X
Winzip	R													X	X	X	X	X	X	X	X
Winzip	I													X	X	X	X	X	X	X	X
Eraser	I														X	X	X	X	X	X	X
Eraser	R																X	X	X	X	X
TrueCrypt	R																	X	X	X	X
Inv. Secs.	R																		X	X	X
SDelete	R																			X	X

3.2.2 The experimental subject virtual machine

The experimental subject machine was a Windows 7, 64-bit virtual machine. The virtual machine had no network interface provided.

State one of the virtual machine was defined as the baseline state. Each state of the machine had some action taken as in the Diskprint data creation: attaching an installation media image and installing it, or running an application.

Table 3.3 lists the applications placed and run on the subject virtual machine. The ground truth of applications on the machine was then set to be a cumulative application-state set, where if an application was run at time t_1, it was considered run at time t_2, *etc.* Table 3.4 illustrates the appearance order of application installations and executions on the subject virtual machine. An application-state not within this table has a presence record of `False` in ground truth intead of null, as the machine was instrumented with precise knowledge of application usage.

3.2.3 The M57-Patents corpus

"M57-Patents" is a digital forensic training scenario, comprising of a set of four users in a fictitious company, using physical computers for a calendar month [72]. The users in the scenario commit various simulated crimes, and for trainees to solve the case, the scenario developers captured disk images, RAM, and network traffic daily over the course of the month's work weeks. To date, it is the most substantial, documented, longitudinal whole-system forensic research corpus.

Roussev *et al.* analyzed the scenario's images and reported on software pres-

Table 3.5: Applications and versions that Roussev *et al.* observed in the M57-Patents corpus [60, pages S67–S68]. The "First apperance" table cells contain two pieces of information: the scenario day where the application first appeared, and by what data source the application was discovered. Files on the disk that indicated application usage were found by similarity-hashing against RAM ("R"), network traffic ("N"), or by some other unspecified inspection ("-"). For example, the Cygnus hex editor was found on Charlie's machine on November 24th, finding files in RAM with similarity hashing. Some applications were listed without version information (denoted with "-"). AVG was noted as "updated," believed present before, but the first appearance was not reported in the text.

		First appearance		
Application	Version	Charlie	Jo-new	Pat
7-Zip	-	11-24/-		
Adobe Reader	9			11-19/R
AVG	-			12-03/R
Brother printer driver	-			11-30/R
Cygnus Hex Editor	Free Edition	11-24/R		
Firefox	-			11-16/R
Invisible Secrets	2.1	11-19/NR		
Java	-			11-16/R
MDD	1.3			11-16/R
Python	-			11-16/R
RealVNC	4			12-07/R
TrueCrypt	6.3a		12-03/R	
Win32dd	-			12-07/R
XP Advanced KeyLogger	-			12-03/R

Table 3.6: Applications that Roussev *et al.* observed in the M57-Patents corpus [60, pages S67–S68], with versions provided by embedded metadata of executable files. *exiftool* [25] extracted the metadata. First-observed dates are given according to presence of files in the file system—not the Registry. Presence in the baseline machine state (the "start" of November 12) is denoted "Base". Program presence not listed by Roussev *et al.* was not sought. Cells with a "(+)" annotation show a greater discovered appearance window than Roussev *et al.* reported.

		First appearance		
Application	Version	Charlie	Jo-new	Pat
7-Zip	4.65	11-24		
Adobe Reader	9.2.0.124			11-19
AVG Internet Security	9.0.0.706			(+) Base
Brother printer driver	1.07			11-30
Cygnus Free Edition	1.00.101	11-24		
Firefox	3.5.5			(+) 11-12
Invisible Secrets	2.1.0.1	11-19		
Java(TM) Platform SE 6 U17	6.0.170.4			(+) Base
MDD	1, 3, 0, 0			11-16
Python	2.6.1			(+) Base
VNC Server Free Edition	4.1.3			12-07
TrueCrypt	6.3a		12-03	
Win32dd	(none)			12-07
XP Advanced Keylogger	V 2.1			12-03

Table 3.7: Applications that Roussev *et al.* observed in the M57-Patents corpus [60, pages S67–S68], re-cast as gound truth definitions for Signature Searchers. Ground truth is denoted as the date followed by a diskprint-matching character, with "I" denoting program installed on and after this date, "R" denoting program run, and "-" denoting that NIST did not produce a diskprint for the program. Python used in the Pat machine was embedded in an installation of OpenOffice. The Brother printer driver did not have an execution diskprint created, due to lack of hardware. The key logger on Pat's machine was excluded from ground truth judgements, positive or negative, after Dec. 7, due to it being removed in the scenario.

	First appearance		
Application	Charlie	Jo-new	Pat
7-Zip	11-24/IR		
Adobe Reader			11-19/IR
AVG Internet Security			Base/-
Brother printer driver			11-30/I
Cygnus Free Edition	11-24/-		
Firefox			11-12/IR
Invisible Secrets	11-19/IR		
Java(TM) Platform SE 6 U17			Base/-
MDD			11-16/-
Python			Base/IR
VNC Server Free Edition			12-07/-
TrueCrypt		12-03/IR	
Win32dd			12-07/-
XP Advanced Keylogger			12-03/IR

ence using their similarity hash-based methodology [60]. They used disk state, RAM state, and in some cases network captures. Table 3.5 gives the applications and versions they reported. To confirm the application report, Table 3.6 provides the versions of those applications, by my manual inspection of the file systems and extracting the metadata embedded within executable files with *exiftool* [25]. The Signature Searcher evaluation against the M57 data uses Table 3.7 for ground truth. An application installation or execution that appears within the scenario on a day d_i has a presence of `False` in the ground truth for days before d_i, and `True` on d_i onward unless removed. If the application is removed on d_j, its presence is null in ground truth, as this dissertation is not studying the "Decay rate" of storage artifacts. Applications not listed in Table 3.7 are null in ground truth, to not influence the classification results.

3.3 Instrumentation

This section describes the steps taken to convert virtual machine states, to document search models, to software signature recognizers. First, the analytic framework that constructs the differences receives some explanation, as it has proven to be a productive design extension to forensic differential analysis. That framework provides the change sets that become software signatures in a search engine. Next, the parameter space for converting the change sets into signature documents is enumerated. Last, the strategies for identifying background noise culminate in a stop list parameter, which needs to interact with other Signature Searcher design parameters.

3.3.1 A framework for analyzing data lineage graphs

Once evidence is acquired in a digital forensic case, pre-processing steps are typically necessary for analysis to begin. These steps usually include hashing disk contents for evidence integrity, extracting files and hashing them, and extracting text into a search engine [59, 18]. These steps are examples of a *forensic workflow*, which generally can be considered a series of steps taken on source or intermediary data. Often, these workflows are concerned with a computer's state at only a single point in time.

To develop software signatures, a workflow must be able to relate multiple points in time to one another, for the sake of enumerating what has changed between those points in time. This section presents the workflow that produces forensic feature difference sets, making use of the lineage data provided by the Diskprint virtual machines' histories.

3.3.1.1 Single-state forensic workflows

A common forensic setting is receiving a computer for analysis, that has not been seen before. This single system state is then fed through some "ingest engine" that executes a workflow. For this document, we consider a *workflow* to be the automated steps taken starting from some set of input data, with each step having a clear set of inputs and outputs. For example, Figure 3.1 illustrates a workflow similar to what Roussev *et al.* used to identify software in the M57-Patents scenario. Their strategy was to extract executable files from the file system, and then use similarity hashing to determine which files had been loaded into RAM, indicating software execution.

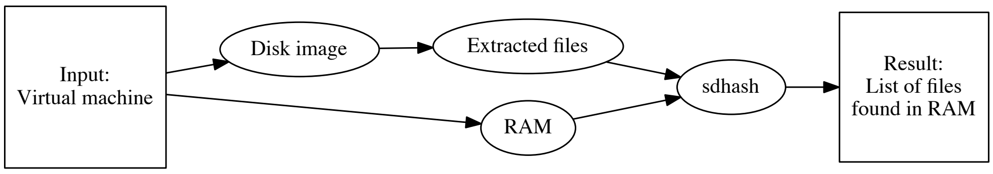


Figure 3.1: A forensic analysis workflow similar to what Roussev *et al.* used to identify software in machines where disk and RAM were captured [60]. Arrows indicate data flow. First the disk image and RAM state are extracted from the input system. A file extractor extracts all executable-related files (`*.exe`, `*.dll`, *etc*). Then the similarity hasher *sdhash* [58] uses fuzzy hashing to determine which files appeared in system RAM.

A single-state workflow needs only a single system state to derive all of its results. The workflow in Figure 3.1 could even be executed with the *make* utility [37], as the only varying factor is the single system state at the beginning. However, for software signatures, a workflow that can identify changes between multiple states is necessary.

3.3.1.2 A dependency graph processing strategy

When considering multiple states of a subject machine, more options become available in forensic workflow development. A single-state workflow can be run on each state without any additional creativity. However, with a lineage tree, we can relate virtual machine snapshots together and automatically assign differencing operations to run. This takes the current practice of *forensic differential analysis* to *forensic sequence analysis.*

The Diskprint data have lineage trees tying virtual machine states to one another. In the abstract, a lineage tree describes a history of states. In the tree, a state is a node, and a directed edge indicates ancestry. A state has one immediate ancestor, but can beget multiple points if a virtual machine rolled back to it to take actions starting from a common snapshot. A *multi-state workflow* executes several types of operations on those states:

Node operations A node operation relies on the data of a single state, *e.g.* extracting files from a disk image.

Edge operations An edge operation relies on data derived from two states, *e.g.* comparing sets of files between two disk images.

Sequence operations An operation on a non-branching subgraph of the lineage tree relies on data derived from a node of the subgraph, and its ancestors up to and including some other node. Sequence operations can follow subsets of a linear sequence of the graph, which would be desirable if some virtual machine snapshots are taken at too fine a granularity.

Graph operations An operation on a subgraph of the lineage tree—*e.g.* that describe all states spawning from a particular baseline—operates on all the nodes and/or edges in the subgraph.

The next section describes how Registry differences are extracted from Diskprint sequences. The extraction workflow works on the node, edge, and sequence levels of the lineage graph. The rest of this chapter is devoted to the graph-level analytic task of devising an optimal Registry search engine.

3.3.1.3 The Diskprint extraction workflow

The software signatures of this work are based on analysis of the differences in Windows Registry state between each of the virtual machine snapshots. The differences are gathered with the following workflow:

1. Extract file system metadata from each disk state, storing the metadata in DFXML.
2. Given file system metadata, which shows the location of file contents on disk, extract Registry hive files and convert them to the analytic format RegXML [40]. The tool *RegXML Extractor* [41] handles the hive extraction and hive content conversion.
3. Compute the differences between hives from state to state, using *rdifference.py*.
4. Export all differences to a central database [42] for graph-level analysis that is out of scope of the extraction workflow.

The diskprint lineage graph creates a programmatically-accessible relationship between system states, making it possible to execute difference operations only after their dependent prior tasks are completed. Figure 3.2 shows the workflow analyzing the differences between three Registry hive sets, and producing two sets of cells added since their respective prior snapshotted states. This is how the all the levels of lineage graph operations appear in the workflow, as illustrated and/or as implemented:

Node level Some Registry results are derived at the lineage graph's node level, by feeding hives into a tool (*RegRipper* [8]). These results do not require the context

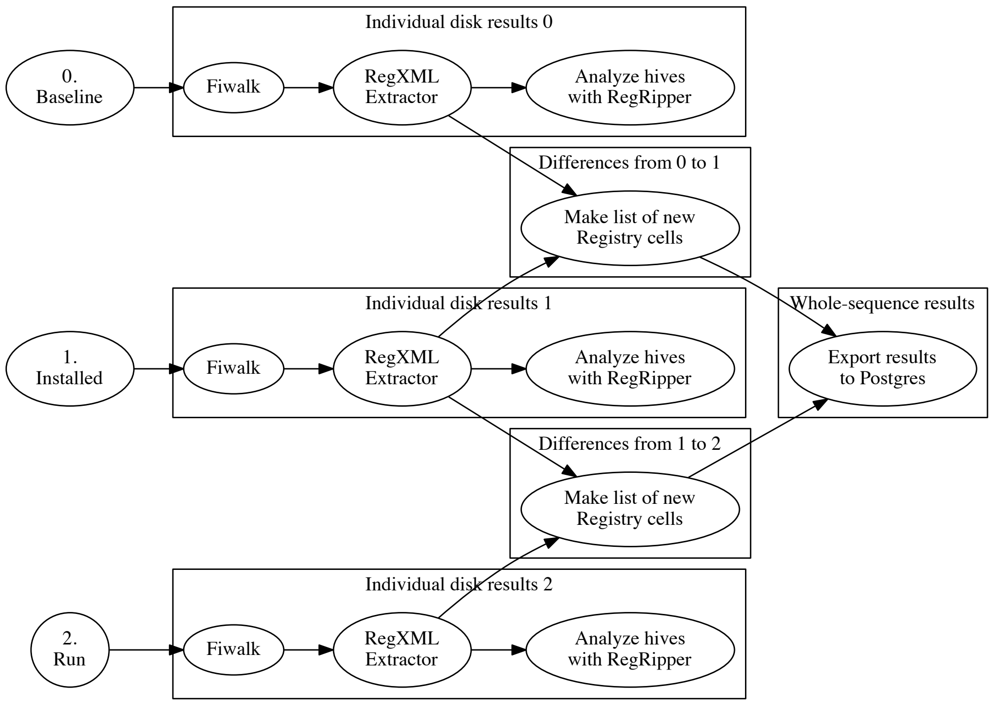


Figure 3.2: Diskprint workflow producing several results derived from “edge”-level analysis.

of other disk states.

Edge level Registry differences are illustrated here as being derived between two lineage nodes' results, but the way the workflow is implemented has the Registry differences extracted as a sequence-level operation (described next) which is logically equivalent. File system difference results are implemented as an edge-level operation, but these results are out of scope of this thesis.

Sequence level Between-node Registry differences are implemented as one script that reads all node-level *RegXML Extractor* results at once, instead of collecting results of one script run per lineage edge.

The workflow's current implementation is built on Bash and Python scripts, with data centralized in a Postgres database, and parallel processing handled by GNU Parallel [64].

3.3.2 Document search-based software signatures

Section 1.8.1 introduced concepts and nomenclature for document search, as the terms are normally used in human language document search. This section adapts search concepts to software signatures built from forensic features, focusing on cells extracted from the Windows Registry as a feature class. After an illustration of how an English language search engine is constructed, an elementary software signature search engine is constructed. Challenges arise in the elementary construction, and the next sections provide search model parameters to address those challenges.

3.3.2.1 An example natural language search engine

For a collection of documents $\mathcal{D}$, a set of users desires the ability to do keyword searches over those documents. A developer constructs a search engine, making these design decisions:

- The documents shall be treated as Bags of Words, with each document an "unordered collection of words with no relationships, either syntactic or statistical, between them" [10]. Word order is ignored after constructing n-grams. If the developer chooses n-gram length $n = 1$, word order will not matter at all.
- The words shall be filtered to words not on a stop list of common English words ("the," "it," *etc.*).
- The words shall be stemmed by a set of rules to remove suffixes, such as the Porter stemmer [56].
- The words will be transferred into n-grams, which are the search terms.
- The document matrix of the engine shall store TFIDF weights for terms, as defined by Equation 1.1.
- The similarity metric matching queries with documents shall be the vector cosine.

The developer then takes the documents, processed into lists of n-grams absent sentence punctuation, and ingests them into the engine. The search engine is then presented with a query:

“`the quick brown fox jumps quickly`”

The engine converts this to a vector of terms, using the same stop list, stem, and n-gram rules: `<brown: 1, fox: 1, jump: 1, quick: 2>`. Using the vector cosine, this document’s nearness is scored against each document vector in the TFIDF matrix, and the results are returned ranked in decreasing score order. The user browses the documents, likely finding those truly relevant to the query higher in the ranked results. The performance of this engine would be measured according to a pre-determined ground-truth relevancy definition of documents returned from queries. For example, one performance measure may be the “Precision at N,” the precision of the top N documents.

3.3.2.2 An example forensic feature search

In this example, an examination for installed and used software is requested for an analysis of a subject desktop machine. The requester suspects a set of software packages is present. To determine which of those packages are on the subject machine, a team of analysts creates diskprints of much of the software in the collection, one print per package, with each package printed on an operating system similar to the subject machine. They intend to search for the characteristic changes that happen when software is installed, by comparing the Registry differences between the diskprints with the Registry entries found on the subject machine.

Using the Diskprint machines’ snapshots and the forensic workflow described in Section 3.3.1.3, a *change set* of added Registry cells is constructed for each software package's installation, and another for the software execution since installation. If a piece of software was installed or run on the subject machine, a similar set of Registry cells should be present. An analyst constructs a search engine to determine what software status the set of all present cells indicates. The analyst makes these decisions:

- A Registry cell's entire path shall be a term.
- The paths on the baseline diskprint operating system image will comprise a stop list of terms generic to the operating system, and hence analysis-subject machine, instance.
- Each change set shall be a document.
- The search engine is otherwise constructed as the previous example's engine for natural-language documents: TFIDF document-term weighting, with the vector cosine for measuring similarity to queries.
- The set of all Registry paths in the subject machine shall be the query.

The analogy to natural language document search diminishes at this point, as the ratio of query length to average document length is comically disproportionate. A similarly-scaled natural language query would be finding the "most Shakespearean" employee in a company, by concatenating all of Shakespeare's works into a file, and using that file as a query against all of the company's individual emails.

Evaluation also differs from natural language search evaluation. In natural languages, the engine's objective is to provide an ordered set of relevant documents, but the ultimate objective is often partially satisfiable if the user leaves better-informed. The goal in software recognition is declaring presence or absence of software usage, a Boolean question. However, search scores range as real numbers between 0 and 1. For conversion to binary responses (the software is installed, or is not installed), some threshold needs to be established for software presence.

The most significant characteristic of the input data in this example is that each print was only performed once. This allows the model to treat a change set as synonymous with a document in the search engine. If the input data advance in complexity beyond singleton prints, the design space of the model expands significantly.

- Hive root cell names change between operating systems, or system architecture (*e.g.* 32-bit or 64-bit). If multiple operating systems are used within the training data, then the paths need to be normalized in some way to capture what may semantically be the "same" Registry cell, under a different root path. Normalizing path prefixes is analogous to stemming in natural language search.

- If a diskprint for a piece of software is performed multiple times, and each print is used for model training, then documents cannot be defined as change sets. Artifacts that are truly discriminatory within the change sets lose their distinctiveness among signatures if the first and second print of an application are independent signatures. Meanwhile, unrelated background noise that only appears in one of the signatures by accident suddenly becomes a key indicator—of a wrong answer. Similarity to the "Important" elements of *grouped* change sets is the target function.

Overall, several vector space model design issues arise when using training data more complex than in the example search engine of this section. In particular, combining data changes fundamental capabilities of the model, such as whether signatures can be of an application universal across Windows versions, or if they must be tailored to applications on each Windows version of interest. The next sections are devoted to treating each of these design questions, and more, as model parameters.

3.3.3 Model parameters from print repetition

This section explores strategies to combine change sets. Building a search-based model on only the added cells of one diskprint per application would be unreliable: all system state changes would be included, with no way to tell from just the changes whether they were caused by user-software actions or unrelated activity. Other diskprints can be added to reinforce the single-print data, representing the same software, at the same lifecycle phase, with the same user actions. Variances will occur in the prints, some by scheduled system processes, some input variation like a different operating system or software version. In the end, the software signature search engine design should follow the principle of more data improving the models. As discussed previously, this has not been the case in other digital forensics research based on statistical machine learning, *e.g.* in file fragment data type classification [20].

The forensic features from added-cell change sets need to be combined somehow, and there are several decisions to be made when combining features. The set of decisions made defines a single vector space model, which will have some performance characteristics that we can rank to choose the best model parameter set. Some of the decisions made with these models have incidental benefits to addressing problems with background noise infecting the signatures. Each of the following sections of this chapter explores one of the model parameters:

- Training sequences
- Operating system grouping
- Path normalization
- Signature document terms and n-gram construction
- Vector combinator
- Threshold selector
- Software version grouping
- Stop lists
- Interactions between stop lists and n-grams

3.3.3.1 Training sequences

Tables 3.1 and 3.2 showed that some of the applications were printed once on an operating system, and some were printed multiple times. Different subsets of all of the diskprints can be used to train a search model. The evaluation will consider three:

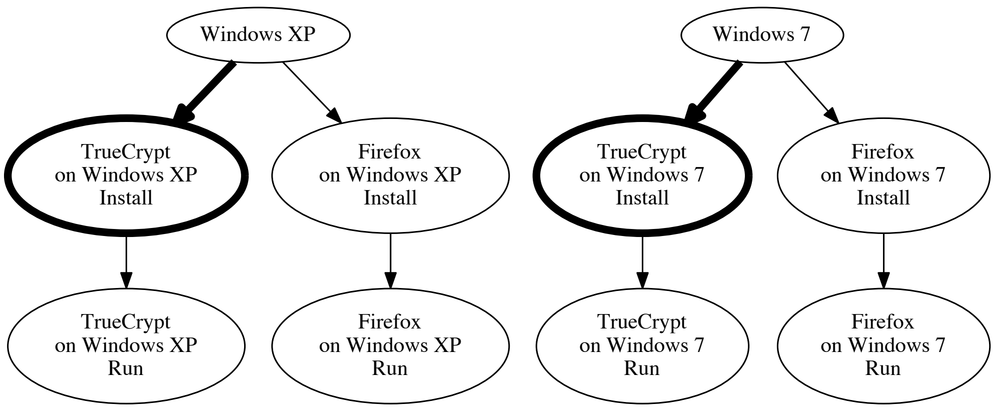
installclose The first simply uses all of the diskprint *baseline-install-close* sequences.

repeated The second subset only trains search models on applications that were printed on the same operating system repeatedly—*i.e.* the "3" records in Tables 3.1 and 3.2. This reduces the set of signatures that can be made, but removes chances for spurious data to unduly influence signatures, as each supplied diskprint will have a built-in notion of consistently-appearing cells. On the other hand, it may be that the singleton application prints will provide more instances of in-common cells that should be ignored. Hence, training on only repeated diskprint sequences simultaneously gains and loses background noise identification support.

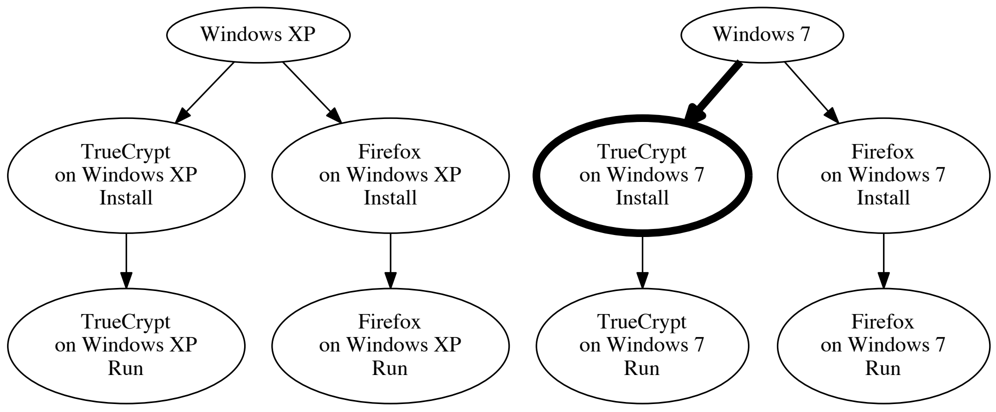
experiment1 The third subset is the "Experiment One" subset, of only diskprint sequences that were used in the developed experimental subject machine (Section 3.2.2). The applications are listed in Table 3.3. This subset will show the relative strength of training models on only the applications one expects to find on a subject machine. Overall, false positives should decrease as there is less to mistakenly recognize; but once again, the extra support for identifying background noise will be lost.

3.3.3.2 Operating system grouping

An important design question for the software signatures is the influence of the operating system. Does one search for signs of an application, tailored to an operating system version; or does one search with a signature that is operating system agnostic?


Windows XP
Windows 7
TrueCrypt on Windows XP Install
Firefox on Windows XP Install
TrueCrypt on Windows 7 Install
Firefox on Windows 7 Install
TrueCrypt on Windows XP Run
Firefox on Windows XP Run
TrueCrypt on Windows 7 Run
Firefox on Windows 7 Run

(a) Data supplying a signature of Truecrypt installation on all Windows versions, with boldfacing indicating included data to be combined.


Windows XP
Windows 7
TrueCrypt on Windows XP Install
Firefox on Windows XP Install
TrueCrypt on Windows 7 Install
Firefox on Windows 7 Install
TrueCrypt on Windows XP Run
Firefox on Windows XP Run
TrueCrypt on Windows 7 Run
Firefox on Windows 7 Run

(b) Data supplying a signature of Truecrypt installation on Windows 7, with boldfacing indicating included data.

Figure 3.3: A sample lineage graph, showing the histories of two applications run on two baseline virtual machines. Edges denote virtual machine progression to the next snapshot, and the accompanying change data between snapshots. Two kinds of signatures are derivable from this lineage: signatures for an application, without regard of the underlying operating system; and signatures of an application for each operating system.

The training diskprints can be grouped by only application, or by application and operating system. For example, if the application Firefox is printed on two operating systems, to group *by application* is to group both operating systems' prints into one signature document; grouping *by operating system and application* would make two documents. The illustration in Figure 3.3 can represent four signatures, or eight.

Document grouping shows how much impact the baseline operating system has on the created signature. Some cells will appear consistently regardless of the operating system version. However, determining the better of one signature that is operating system agnostic, versus one signature tailored for each operating system, requires measurement. This also has an impact on the work required to create signatures, such as needing to update signatures of each target application for every new release of the Windows operating system.

3.3.3.3 Path normalization

Cell path normalization is analogous to stemming in natural-language search engines. It is used primarily to address features that appear in every operating system, and potentially resolving issues with Registry prefix strings not matching. This concern comes from the root cells of hives having a different name depending on the version of Windows being inspected.

To compare cell paths across Windows versions, I "Normalize" the paths according to the hive's role in the system. The normalization strategy is to replace all of the root cell names with a single string distinct to the class of hive, effectively making

Table 3.8: Groupings, or "general types," of hives, for user hives. There are two types of hives that each user will have, "`NTUSER.DAT`" and "`UsrClass.dat`." This table reports other places in the file system where hives with those names appear. The parenthetical remarks denote the class assigned to these hives for path normalization, such as a "Default" `NTUSER.DAT` found in Windows XP.

File system path suffix	Hive type
`Documents and Settings/Default User/NTUSER.DAT`	NtUser (default)
`Documents and Settings/LocalService/Local Settings/Application Data/Microsoft/Windows/UsrClass.dat`	UsrClass (Loc. Svc.)
`Documents and Settings/LocalService/NTUSER.DAT`	NtUser (Loc. Svc.)
`Documents and Settings/NetworkService/Local Settings/Application Data/Microsoft/Windows/UsrClass.dat`	UsrClass (Net. Svc.)
`Documents and Settings/NetworkService/NTUSER.DAT`	NtUser (Net. Svc.)
`Users/Default/NTUSER.DAT`	NtUser (default)
`WINDOWS/repair/ntuser.dat`	NtUser (repair)
`Windows/ServiceProfiles/LocalService/NTUSER.DAT`	NtUser (Loc. Svc.)
`Windows/ServiceProfiles/NetworkService/NTUSER.DAT`	NtUser (Net. Svc.)
`Windows/System32/config/systemprofile/ntuser.dat`	NtUser (config)
Other `NTUSER.DAT`	NtUser (user)
Other `UsrClass.dat`	UsrClass (user)

Table 3.9: Groupings, or "general types," of hives, for system hives.

File system path suffix	Hive type
`WINDOWS/repair/sam`	Sam (repair)
`WINDOWS/repair/security`	Security (repair)
`WINDOWS/repair/software`	Software (repair)
`WINDOWS/repair/system`	System (repair)
`WINDOWS/system32/config/COMPONENTS`	Components (config)
`WINDOWS/system32/config/SAM`	Sam (config)
`WINDOWS/system32/config/SECURITY`	Security (config)
`WINDOWS/system32/config/software`	Software (config)
`WINDOWS/system32/config/system`	System (config)
`Windows/System32/config/SAM`	Sam (config)
`Windows/System32/config/SECURITY`	Security (config)
`Windows/System32/config/SOFTWARE`	Software (config)
`Windows/System32/config/SYSTEM`	System (config)

Table 3.10: Counts of hives of each class (as defined in Tables 3.8 and 3.9), contributed by each operating system. Some hive classes are specific to an operating system, like the “Repair” classes of hives only appearing in Windows XP. The counts in this table are of hives found in the diskprint training sequences. How many were successfully processed for analysis is a separate question, measured in Section 4.1.

	XP/32	V/32	V/64	7/32	7/64	8/32	8/64
NtUser (user)	328	54	50	187	143	51	16
NtUser (config)	0	55	51	183	139	53	17
NtUser (default)	178	55	51	179	135	53	17
NtUser (repair)	178	0	0	0	0	0	0
NtUser (Loc. Svc.)	178	55	51	183	139	53	17
NtUser (Net. Svc.)	178	55	51	183	139	53	17
UsrClass (user)	326	54	50	183	139	51	16
UsrClass (Loc. Svc.)	178	0	0	0	0	15	0
UsrClass (Net. Svc.)	178	0	0	0	0	15	0
Components (config)	0	55	51	183	139	53	17
Security (config)	181	55	51	183	139	53	17
Security (repair)	178	0	0	0	0	0	0
System (config)	181	55	51	183	139	53	17
System (repair)	178	0	0	0	0	0	0
Sam (config)	181	55	51	183	139	53	17
Sam (repair)	178	0	0	0	0	0	0
Software (config)	181	55	51	183	139	53	17
Software (repair)	178	0	0	0	0	0	0

the prefix of all cell paths for a hive "Class" consistent. Tables 3.8 and 3.9 present how the hives are grouped into classes for this research, and Table 3.10 counts how often each of the types appears in the training diskprints. For an example on what a cell path looks like after normalizing, the "`$$$PROTO.HIV`" at the root of an XP System hive is replaced with "`__NORMROOT_SYSTEM_CONFIG__`."

Table 3.11: Number of cell paths found overlapping between various baseline operating systems. Generated from exhaustive listings of all cells in the last slice of each baseline sequence, without running preinstalled applications. Windows XP (32-bit) is repeated because it had already been diskprinted twice, and the repetition's overlap illustrates consistency in cell paths. Paths are exactly as found from parsing the hive files.

	XP (1)	XP (2)	Vista-32	Vista-64	7-32	7-64	8-32	8-64
XP (1)	115,482	111,779	0	0	0	0	0	0
XP (2)		116,075	0	0	0	0	0	0
Vista-32			344,484	62	0	0	60	60
Vista-64				536,317	0	0	60	60
7-32					475,720	63	0	0
7-64						622,326	0	0
8-32							456,109	288,126
8-64								661,560

Table 3.12: Path prefixes for the Windows `ShutdownTime` value cell, under the first of two mirrored `ControlSet` keys in the System hive. Each of these prefixes starts from the root of the hive namespace, descending only to the parent key of `ControlSet001\Control\Windows\ShutdownTime`.

Version	Arch.	Absolute prefix
XP	32	`\$$$PROTO.HIV`
Vista	32	`\CMI-CreateHive{C619BFE8-791A-4B77-922B-F114AB570920}`
Vista	64	`\CMI-CreateHive{3406549D-D5AD-434A-9894-E927ABEC8146}`
7	32	`\CMI-CreateHive{F10156BE-0E87-4EFB-969E-5DA29D131144}`
7	64	`\CMI-CreateHive{2A7FB991-7BBE-4F9D-B91E-7CB51D4737F5}`
8	32	`\CsiTool-CreateHive-{00000000-0000-0000-0000-000000000000}`
8	64	`\CsiTool-CreateHive-{00000000-0000-0000-0000-000000000000}`

Table 3.13: Counts of overlapping cells within hives of Windows 8 32-bit and 64-bit.

1,325	Users/Default/NTUSER.DAT
3,271	Users/nsrl-user/NTUSER.DAT
1,394	Windows/ServiceProfiles/LocalService/NTUSER.DAT
1,341	Windows/ServiceProfiles/NetworkService/NTUSER.DAT
37,117	Windows/System32/config/COMPONENTS
153	Windows/System32/config/SAM
167	Windows/System32/config/SECURITY
197,494	Windows/System32/config/SOFTWARE
49,852	Windows/System32/config/SYSTEM
61	Windows/System32/config/systemprofile/ntuser.dat

Table 3.11 shows why the string prefix issue needs addressing: without normalizing, signatures built from the raw, full Registry paths of one operating system can't work on another operating system. The table shows the pairwise intersections of Registry paths, for each operating system against all others. Without any modifications to Registry cell paths, the paths are overwhelmingly distinct to the major operating system release and architecture (32-bit or 64-bit). This is due to each hive containing a string in its root cell name, distinct to the hive's role in the Registry namespace. For illustration, the `ShutdownTime` key appears in all Windows versions' System hives, and Table 3.12 lists their common-path prefixes.

Windows 8 is the one operating system with overlaps between architectures. Those overlaps are spread throughout most of the hives typically inspected during forensic investigations. Table 3.13 shows the overlap tallies between the Windows 8 architecture versions. Between other operating system versions, the instances of under one hundred cells overlapping (*e.g.* between Vista and Windows 8) are all in the "`systemprofile`" account's `ntuser.dat` hive.

Table 3.14: Cell paths found overlapping between various baseline operating systems. The same Registries as in Table 3.11 are presented, except paths are normalized here.

	XP (1)	XP (2)	Vista-32	Vista-64	7-32	7-64	8-32	8-64
XP (1)	218,843	212,028	60,788	57,659	57,349	54,940	48,562	46,162
XP (2)		219,084	61,789	58,656	58,436	55,934	49,278	46,861
Vista-32			348,607	248,176	162,163	158,193	119,974	117,065
Vista-64				540,419	157,009	248,238	115,895	193,237
7-32					482,279	315,247	164,071	154,402
7-64						628,867	154,602	257,082
8-32							460,159	293,895
8-64								665,631

Normalizing cell paths enables comparison across Windows versions. The same baseline cell distinctness from Table 3.11 is shown, “Path-normalized,” in Table 3.14.

3.3.3.4 Signature document terms

An initial perceived advantage of using Registry paths as terms within signature documents is the automatic scoring of terms bestowed by the TFIDF matrix. Each path could be scored as useful to each software signature or not, with a real value between 0 and 1. However, some points of fragility arise with this strategy:

- Some paths include a cell named by a varying pattern created at install time. For example, the `UsrClass.dat` root key will vary per user (including “Administrator”) and system. Table 3.15 shows the UsrClass root keys for the three M57-Patents Windows XP machines.
- If subtrees of the Registry are moved between versions of Windows, signatures built before the move was implemented would not match on systems running on

the later Windows version.

Table 3.15: Names of `UsrClass.dat` for the Windows XP machines in the M57-Patents scenario. If a signature is built on whole paths without normalizing as in Section 3.3.3.3, none of the cells in this hive can be used because of this varying root name. The exact cell path after the root will not be found on another system.

System	Account	`UsrClass.dat` root
Charlie	Charlie	`\S-1-5-21-682003330-329068152-1644491937-1003_Classes`
Charlie	Administrator	`\S-1-5-21-682003330-329068152-1644491937-500_Classes`
Jo	Jo	`\S-1-5-21-583907252-527237240-1606980848-1003_Classes`
Jo	Administrator	`\S-1-5-21-583907252-527237240-1606980848-500_Classes`
Pat	Pat	`\S-1-5-21-1292428093-1645522239-1547161642-1003_Classes`
Pat	Administrator	`\S-1-5-21-1292428093-1645522239-1547161642-500_Classes`

In short, while whole Registry paths are easy to explain when inspecting signature hits, they may prove too fragile to match against subject systems. One solution to matching components of the paths is to change the semantic mapping from linguistic search: instead of treating paths as terms, treat them as sentences of path components, using n-grams of components as terms. For illustration, consider the example key from the Introduction, that was measured to be useful in a Microsoft Office 2003 signature from one of the constructed Signature Searchers:

- `\$$$PROTO.HIV\Software\Netscape\Netscape Navigator\Suffixes\application/msexcel`

A unigram-based vector space model would convert that path into these terms:

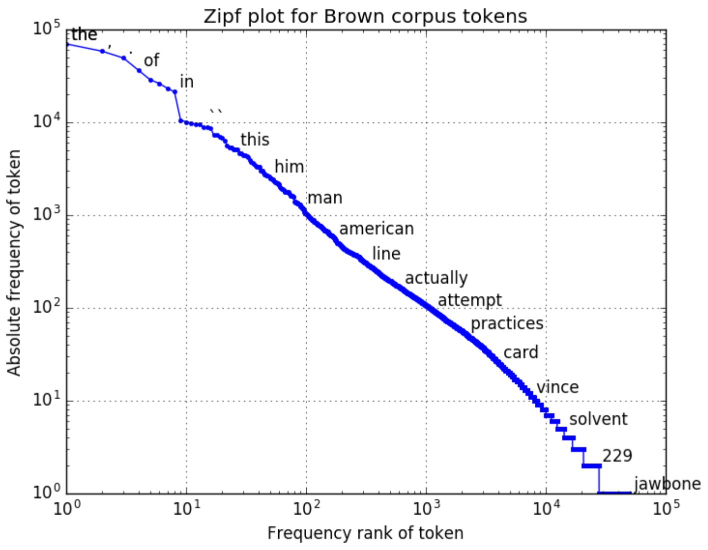
- `$$$PROTO.HIV`
- `Software`

- `Netscape`
- `Netscape Navigator`
- `Suffixes`
- `application/msexcel`

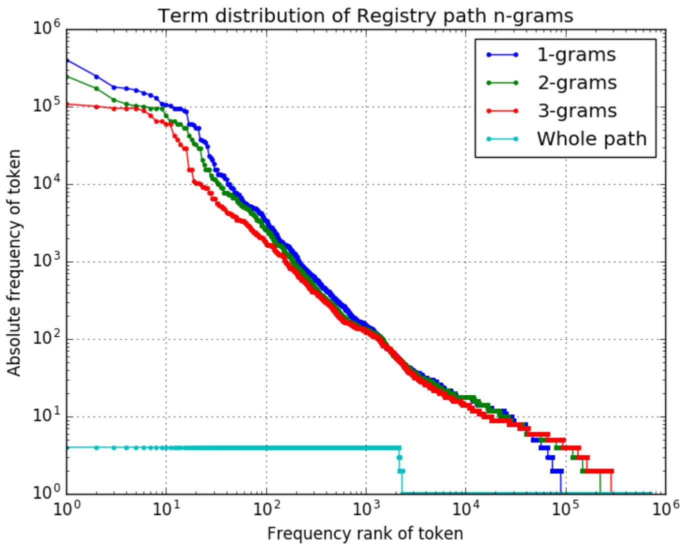
Some concerns are evident from this example of converting a Registry path to unigrams. First, some confusion could ensue from seeing particular names in unrelated products—here, "`Netscape Navigator`" in the path is an indicator of Microsoft Office. Bigrams, such as "`Suffixes\application/msexcel`," or trigrams, could be used instead to maintain a window of the original tree structure.

Second, there is a mild encoding difficulty with Registry path delimiter characters. The forward slash is a legal character in key names, so care must be taken to store paths with backslash delimiters. However, backslashes cannot be assumed to be a delimiter in an arbitrary Registry cell path, because the backslash is a legal character in value names, and is used frequently in representing file system paths (*e.g.* starting with "`C:\`") as value names. This adds a processing step for n-gram search models in this research, that cell parents need to be tracked.

Third, there is potentially an over-representation of the components that appear closer to the hive files' roots, possibly meriting some kind of stop-listing rule. However, this actually brings Registry search closer to linguistic search. In natural languages, term rankings follow a Zipfian distribution [35], as illustrated in Figure 3.4a for the Brown corpus of English language, nonfiction text documents. The most-common



(a) Distribution of unigrams in the Brown corpus, as provided by the Natural Language Tool Kit [4].



(b) Distribution of unigrams, bigrams, trigrams, and whole paths in the Registry of the last state of the experimental virtual machine used in the evaluation chapter.

Figure 3.4: English vs. Registry term distributions.

terms in the distribution are usually added to stop lists [10]. That won't work if terms are whole Registry paths, as the bottom line of Figure 3.4b shows that the distribution of Registry paths across all of the change sets is far from Zipfian in shape. However, if the terms are derived from path components, those distributions are closer to Zipfian, as the whole of Figure 3.4 illustrates.

The analysis of term distribution could continue at this point to fitting directory depth distributions, as done by Douceur, Agrawal, *et al.* in file system namespace depth analyses [13, 1]. However, none of the search mechanisms employed in this work specifically require the distribution to fit any particular distribution. A long tail of the distribution is all that this search application requires, and measurement in the evaluation chapter will show whether it is better for that tail to be utterly flat—as with whole paths—or to retain some descending curve.

It is possible that the last n of the path components may be sufficient for signatures, instead of all of the path. The evaluation will also consider these options.

3.3.3.5 Vector combinator

Each of the following sections that describes a model follows the same pattern: Each model creates a corpus of documents. Each term is a Registry cell path—or derivation from a path—added by some change set or sets. Each document is a combination of the cells of all of the change sets, grouped by application, lifecycle phase, and possibly operating system and/or application version. Where the models differ is in their assignment of values to a document vector, d, by applying a combinator. See

Table 3.16: Symbols used to define the models of repeated diskprints. $\mathcal{D}$ and $\mathcal{T}$ are re-cast from Table 1.3.

Symbol	Definition
σ	A signature ID, identifying application; phase of software lifecycle; and if the model is so configured, operating system and/or application version.
$\mathcal{P}$	The set of all Registry cell paths observed in the training data. Note that this is not scoped to known cell paths; paths not used in the models can exist in investigation subject machines.
$\mathcal{T}$	The set of all terms derived from Registry cell paths in $\mathcal{P}$.
$\mathcal{S}_\sigma$	The set of all change sets that are to be grouped into the signature σ.
$\mathcal{D}$	The set of all signature documents. Documents are constructed from change sets in $\mathcal{S}_\sigma$.
s	A change set, $s \in \mathcal{S}_\sigma$. A change set is represented as a vector of integers in $\|\mathcal{T}\|$-dimensional space.

Table 3.16 for the definitions of in-common symbols used to define these models. In the following definitions of combinators for documents' terms, let $i \in \mathcal{T}$.

Definition 3.1 (*Intersection combinator*)**.** Each document is a vector of binary values. A term is included in the vector IFF it appears in *all* change sets of the document grouping. In the formula, the inner min verifies presence, while the outer min handles conversion to binary.

$$d_i = \min(1, \min_{s \in \mathcal{S}_\sigma} s_i)$$

Definition 3.2 (*Union combinator*)**.** Each document is a vector of binary values. A term is included in the vector IFF it appears in *any* change sets of the document grouping.

$$d_i = \min(1, \max_{s \in \mathcal{S}_\sigma} s_i)$$

Definition 3.3 (*Summation combinator*)**.** Each document is a vector of non-negative integers. The value for each term is the number of times that term appears in the change sets of the document grouping.

$$d_i = \sum_{s \in \mathcal{S}_\sigma} s_i$$

Definition 3.4 (*Sumint combinator*)**.** This combinator is as the summation combinator, except only terms that appear in the intersection are summed. Other terms are excluded.

$$d_i = \min(1, \min_{s \in \mathcal{S}_\sigma} s_i) \cdot \sum_{s \in \mathcal{S}_\sigma} s_i$$

Pre-evaluation discussion of combinators

With repeated prints, consider for the document vector d the *unit vector* of magnitude one that is parallel to d. The summation model will continue to "point" in the same direction of the commonly occurring cells, presumably the strongest signature elements. A cell that appears rarely, or by a non-repeating pattern, will contribute little to change the unit vector's direction. For the union model, the unit vector's direction will be strongly affected by every newly occurring cell, since they are all given equal weight. The intersection model will be unaffected by non-recurring cells, but will have a reduced overall signature if a cell ever fails to reoccur.

The summation and intersection models focus on identifying the "correct" directions of the resulting document vectors. Their vectors favor repeated terms. The union model allows many wrong terms—suspected background noise—to enter documents, and provides all terms equal weight in the unit vector direction. Hence, the union model is expected to perform the worst of the three, acting as a strawman model parameter.

The "Sumint" model—short for "Sum of intersection"—is expected to perform well. It combines the recognition of the most-repeated terms that the summation model provides, while removing the influence of inconsistently-appearing terms by discarding all terms outside of the intersection.

3.3.3.6 Threshold selector

To this point, we have only described the search engines, and construction of their documents. A search engine ranks documents by their similarity to a query, and similarity is a score between 0 and 1. That score by itself is not an affirmation that a signature is a hit or miss. The score requires some notion of a threshold value, after which it can be interpreted as an affirmation of a signature hit.

The models constructed in this work operate on the assumption that there is not a universal threshold score, above which a signature match is a hit. This decision stems from the expected variance in signature sizes. Some software is expected to create a hefty presence in the Registry, such as an office software suite or a web browser, both of which set up a significant support infrastructure that is centered on the Registry. Meanwhile, other software may be designed to have a minimal impact on the Registry. One universal threshold score that accounts for the spectrum of Registry reliance is assumed impractical.

Forgoing a universal score, we instead create a *Signature Searcher* that bundles a search engine with a set of threshold scores for each of its signature documents. The Searcher thus operates as one binary classifier per signature document, returning a hit for each document similarity that passes the threshold value.

To determine what the threshold for each application is, the change sets that formed the search engines' documents are used to query the engine. Each change set then has a similarity score for the signature document that it trained. From this set of similarity scores, the *minimum*, *average*, or *maximum* score is selected as a threshold score for the signature document. This threshold represents the engine turned back onto the training data, after each signature has been influenced by all of the training data for other applications.

3.3.3.7 Software version grouping

Software frequently changes behaviors as later versions are released, potentially changing the detectable system footprint as well. Therefore, there is an option in constructing the search models: they can search for characteristics of each version of a software package; or instead they can identify artifacts that appear anywhere in the application's history.

It is possible that creating one signature per version of an application will cause weaker signatures, when measured by deploying against a future machine with a potentially newer version of a target software product. Before measuring in the evaluation chapter, let us suppose ten versions of an application are ingested into a search engine that creates a different signature per version. The "core" features that appear in each version will have significantly reduced inverse document frequency weights, from appearing in ten signatures. Measurements of the models will show whether this loss of signal strength in the core features is a good trade to make for being able to identify a specific version of interest, which may be a desired engine feature.

Unfortunately, there is a logistical difficulty in naming the signatures that would need to be resolved in order to group software version change sets together. Some name needs to be encoded to unify a software package's versions. Because of the coding logistics involved, the software version grouping parameter is instead implemented as two modes. In both, each application version has an independent signature developed.

- In the "Grouped" mode, a signature's ground truth definition is relaxed to allow matching other versions. For example, a signature for Firefox 2 would be expected to match a machine that had Firefox 38 on it.
- In the "Distinct" mode, a signature does not match the same application at a different version.

These modes affect the selection of the model threshold described in Section 3.3.3.6, and determining true- and false-positive classifications in the evaluation.

3.3.4 Background noise and stop lists

The Diskprint data collection process intentionally interacts little with the guest virtual machine's internals. For instance, no kernel hooks are placed, or real-time virtual machine introspection [2, 27] performed. Disk image states are captured whole. One consequence of this methodology is that the artifacts collected lack attribution to generating processes. “Noise” from operating system procedures affect the storage system state at the same time that the subject application is running and creating its own effects. This creates the undesirable situation that this background noise can enter into the signature of a software package without any relation to the package, triggering false positive recognitions of software usage.

The purpose of this section is to identify this “background noise,” to classify Registry paths or path components as noise or not. The paths populate stop lists, to exclude the paths from model construction and from queries, but the stop lists have interactions with other model parameters of the previous chapter.

There are two types of background noise, considering the context of a user executing some process:

Passive An unrelated system activity, like a scheduled software update check, creates storage artifacts.

Active The process execution creates artifacts that are generic to any process being run.

An example of *active* background noise comes from an early implementation of the search models in this dissertation [43]. A steganography application, “Invisible Secrets,” showed a similarity spike on two machines in the M57-Patents [72] corpus. However, only one of the scenario machines truly had the application installed. The diskprint change set had incorporated background noise from Notepad, causing a false-positive match because the system without "Invisible Secrets" had used Notepad for some unrelated reason. For illustration, some of the Registry cells that were part of the signature were as follows:

- `\$$$PROTO.HIV\Microsoft\Windows\CurrentVersion\App Management\ARPCache\Invisible Secrets 2.1_is1`
- `\$$$PROTO.HIV\Software\Microsoft\Windows\CurrentVersion\Explorer\UserAssist\75048700-EF1F-11D0-9888-006097DEACF9\Count\HRZR_EHACNGU:P:\Cebtenz Svyrf\Vaivfvoyr Frpergf 2.1\havaf000.rkr`
    - (The ROT13 portion decodes to: `UEME_RUNPATH:C:\Program Files\Invisible Secrets 2.1\unins000.exe`.)
- `\$$$PROTO.HIV\Software\Microsoft\Notepad\iWindowPosX`
- `\$$$PROTO.HIV\Software\Microsoft\Windows\ShellNoRoam\BagMRU\0\0\0`

Clearly, the first two cells are better signature component candidates than the last two cells that respectively belong to Notepad and a Windows most-recently used list mechanism. However, the latter two cells appeared in the training data and were not filtered in this model construction, so when the same generic cells appeared in the M57 sequence, a match appeared. One key issue that allowed this noise to appear is that the generic cells would appear only infrequently in the training data—particularly the last cell, which is a "Shellbag" with a rotating name pattern and is populated by typical user file interaction activities [73]. Few diskprints at the time included file interactions, so the scarcity of appearances across all the training data then was mistaken as an application-specific effect. This was from an early development stage when models were trained from single prints, and makes clear the importance of repeated and slightly-varying training data.

3.3.4.1 Noise from baseline continuations

The baseline operating system images contain artifacts that are not associated with any of the not-yet installed applications. It is possible to identify passive and active background noise by employing the baseline states for simple prints.

To identify passive noise, the baseline operating systems were left running idle for periods of hours to days.

To identify active noise, some of the preinstalled applications—*e.g.* Notepad, Calculator—were printed, omitting an “Installation” step. The diskprint for these applications booted the virtual machine, ran the built-in application, and then shut down.

3.3.4.2 Symmetric differences

Repeating prints provides one straightforward mechanism to identify candidates for background noise. If a diskprint of an application is taken twice, with snapshots taken for the same user actions, a change set s_1 can be logically grouped with its corresponding change set in the second run, s_2. Then, the set of *consistently added cells* seeds a new set: the inconsistently added cells. In set theory, this set is called the *Sym-*

metric Difference [70]. Any cells inconsistently added by multiple applications could be considered background noise, or could simply be irregularly created by an application. A fourth combinator, unfit for training models but that can be used for building stop lists, creates stop lists from grouped diskprints: terms that are in the union of terms that have appeared, but outside of the intersection.

Definition 3.5 (*Symmetric Difference combinator*)**.** Each document is a vector of binary values. A term is included in the vector IFF it appears in the *symmetric difference* [70] of change sets of the document grouping.

$$d_i = \min(1, \max_{s \in \mathcal{S}_\sigma} s_i) - \min(1, \min_{s \in \mathcal{S}_\sigma} s_i)$$

The inconsistently-appearing terms stoplist is inspired by observing that some early signatures had picked up Registry entries that appeared to be Most Recently Used lists, clearly ephemeral and not clearly related. For example, an early signature for the steganographic application "Invisible Secrets" included this path:

- `\$$$PROTO.HIV\Software\Microsoft\Windows\ShellNoRoam\BagMRU\0\0\0`

That same path happened to appear in test images where the software was known to not be present, creating false positives. Since observing that path, inconsistently-appearing terms have become a stop list parameter option.

The symmetric difference stop lists identify both active and passive background noise, without distinguishing between the two. These stop lists interact with earlier stop list decisions, so are treated as an independent parameter in the model constructions.

3.3.4.3 Stop lists and n-grams

If a Signature Searcher uses n-grams of path components instead of whole paths, the application of a stop list built from whole paths becomes confused. These options emerge:

Raw filter The paths of the stop list can filter change sets and queries before deconstructing path components into n-grams.

n-gram blacklist The paths of the stop list can be broken into n-grams, and those n-grams can be used as a blacklisting stop list.

n-gram threshold The paths of the stop list can be broken into n-grams, and those n-grams can be used as a subtraction vector. This way, an n-gram can be in the query calculation if it is more present in the signature *and* query than in the subtraction vector. The subtraction values in the baseline dimensions are likely to become overwhelming from stop list input data, but this option may interact best with the symmetric difference-incorporating stop list models.

3.3.4.4 Summary

This section presented two vector space model parameters that specifically attempt to address background noise. First, several strategies construct stop lists from the training data. Some source the stop list data from the baseline images. One option built on symmetric differences sources data from repeated prints, treating as suspect the data that don't appear consistently. Second, once the stop lists are constructed, their use in the querying process has to accomodate models that use path component n-grams.

The original search models [43] tried to make use of in-common data of many Registries to determine what cells were signatory of applications. Only having single diskprints of each application hampered the ability to determine valuable cells from background noise. Now, with more data in hand, and strategies to combine the data, the search models are more capable of determining both the unimportant common data, and the unimportant rare data.

3.4 Research Procedures

The overall production of a Signature Searcher starts with related virtual machine states. Section 3.3.1.3 covered the workflow that takes the Diskprint virtual machine states and their lineage graph, and performs the Registry-differencing operations on the lineage graph's nodes and edges. That section left "Graph-level analysis," *i.e.* relationships between lineage edges, for later discussion, to which we now come. The Registry differences from the Diskprint data are combined into Signature Searchers, from any combination of the parameters described in Sections 3.3.3 and 3.3.4. The research procedure is now identifying the best values of these parameters. This section goes through the Searcher construction process.

In a production Signature Searcher scenario, these steps would go into building a Searcher:

Ground truth From the training diskprint sequences, the ground truth is constructed from the lineage graph. As in the description of the evaluation subject machine, ground truth is cumulative: Once an application is installed, the next machine states are also flagged as having the application installation signature as present in ground truth.

Cell parents A database of cell parents is constructed for each virtual machine state. In the full evaluation, this is simplified to one database per corpus. This database is necessary because Registry paths cannot be broken into components by the backslash character, as that character is a legal name in Value cells. It may be possible for a backslash to be inserted in a Key cell name by a misbehaving program as well, but that has not occurred in any of the data used in this dissertation.

Stop list The stop lists get constructed, mostly from baseline sequences' states. An implementation nuance is that this step has some backtracking from TFIDF matrix construction, using the null-stoplist term frequency matrices, to construct the symmetric difference stop lists.

TFIDF matrix The TFIDF engine gets constructed from the change sets of the training sequences.

Queries The queries of the training sequences' machines get built.

Query scores The queries are run against the VSMs, producing a list of (*document*, *score*) pairs.

Signature Searcher The threshold matching score of each signature is selected from the training machines and their paired documents' search scores. A Signature Searcher is then constructed by attaching to the TFIDF engine the dictionary mapping signatures to threshold scores.

Table 3.17: Graph-level analysis execution order and direct parameter interaction, starting from the virtual machine disk images and Registry deltas computed by the Diskprint Workflow in Section 3.3.1.3. Dependencies in the third column are references to other steps that need to be completed in full. In the case of symmetric difference stop list construction, a particular instance of another step needs to be completed. Each Searcher parameter that directly influences a step is noted with a bullet. Since the symmetric stop lists use TFIDF matrix construction, some stop lists are only influenced by some parameters, denoted with circles.

#	Step	Deps.	Sequences	Path normalization	Combinator	Stop list	*N*-grams	*N*-gram & stop lists	Doc. grouping	Version matching	Threshold
1	Training disks										
2	Registry deltas	1									
3	Ground truth								•	•	
4	Cell parents	1		•							
5	Stop list	4 (6)		•			•	•	○		
6	TFIDF matrix	2, 4, 5	•	•	•	•	•	•	•		
7	Queries	1, 4, 5		•		•	•	•	•		
8	Query scores	6, 7									
9	Signature Searcher	3, 8									•

The above steps proceed nearly in that order, but some parallel construction is possible. For instance, the queries can all be independently generated once a stop list is constructed. Table 3.17 shows the step dependencies that can be used for dependency-ordered parallel execution. Figure 3.5 illustrates the execution order. Also, since most steps are affected by one or more of the model construction parameters, Table 3.17

shows the parameter interactions.

To evaluate the Searchers, for each machine in each corpus, the Registry of that machine is extracted, converted into a query as with the training queries, and run against the Searcher, with a set of document-query "Hits" reported. Each query-document score is compared to the threshold, with scores passing thresholds reported as "Hits" according to the Searcher. The constructed Searcher's hits are compared to ground truth for the machine's corpus, and tabulated to compute true and false positives and negatives. Per Searcher, these results are summarized as a precision, recall and F1 score, and inspected as described in the next section.

The entire construction and evaluation process was implemented with custom modules written in Python. The motivation to write a custom TFIDF engine stemmed from being able to control the TFIDF selection from the Zobel and Moffat Vector Space Model design space [74], along with non-standard stop list interaction allowances taken at various stages of implementation.

3.5 Data Analysis

The objective of laying out the parameter space of the previous section is to identify the parameters that create the most successful Signature Searchers. Two experiments carry out this exploration. The first is a binary classifier evaluation that compares the parameter effects on software recognition performance. This experiment identifies the best parameter values and best models, according to the Diskprint training

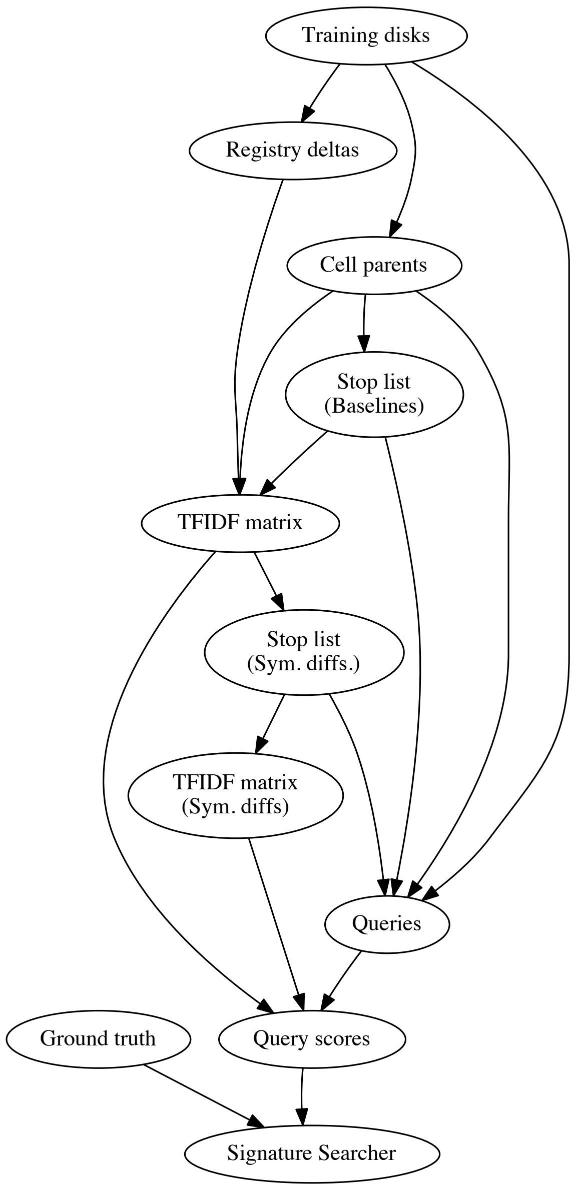


Figure 3.5: Data flow of Signature Searcher construction. With diskprint disk image sequences as input, their Registry deltas become search queries and TFIDF search engine matrices. Ground truth associations—of which signature documents should be "hits" for queries—establish thresholds of similarity scores. That set of thresholds, paired with a TFIDF matrix, is a Signature Searcher. This data flow shows the steps taken from disk image set to Signature Searcher, including an extra stop list construction for Signature Searchers that use inconsistently-appearing Registry cells—symmetric differences—in their stop lists.

data and evaluation subject machine. The second experiment compares the top models against the M57-Patents corpus, demonstrating which model parameters perform consistently. The second experiment needs little further description, but the first has important conventions taken and made, and is due some explanation.

The first experiment measures the parameter space against the evaluation subject machine where ground truth is known, with ground truth illustrated in Table 3.4. The general design of the evaluation against this machine's states is a nine-factor, full factorial experiment design [5]. Each Signature Searcher is treated as a binary classifier, where a querying Registry has a set of ground truth software signatures that should be returned in the query results as hits. The full factorial experiment measures the effect of each factor on precision, recall, and signature tally—*relative* to the other values for that parameter. The measurement of a parameter variable X's value x_1 is computing the average score of each response variable (precision, recall, and F1) over all Signature Searchers that have $X = x_1$, and comparing to the average for $X = x_2$, for each x_i X can assume. This will be presented as *main effects* plots [5], as in Figure 4.10 on page 106. The plots show which individual parameter values perform the best, averaging across all Signature Searchers, one Searcher for each combination of parameter values.

The reason for considering signature tally is that some early measurements showed that the precision and recall scores could be "gamed" by Signature Searchers that reported nearly perfect precision and recall scores, by providing signatures for only a few applications. Hence, the evaluation will balance signature tally with precision and recall.

Other measures of "Correctness" in binary classifier evaluation involve accuracy, and plotting recall versus sensitivity instead of precision versus recall. However, in the Signature Searchers' case, precision and recall is the more appropriate inspection target because the ground truth contains a significant bias towards condition-negative [11]. This ground truth bias is because only a subset of the application signatures are expected to be found in the analyzed corpora.

One cumbersome point of the inspection is naming the Signature Searchers. Each Searcher can be uniquely identified by the nine parameter values selected to create it, but a nonuple name is impractical to print. In order to identify a Searcher, a name will be constructed from its parameter values, much as in Zobel and Moffat's evaluation of vector space models in an eight-dimensional parameter space [74]. They used an eight-character string, with each position dedicated to a parameter and populated with the parameter value based on an enumerated order of the values. If a parameter X had three values, they would be identified as `A`, `B`, and `C`. A different parameter Y with two values would have them be identified as `A` and `B`. Then a model built with the third value of X and the second of Y would have a name starting with `CB`, and so on for the other six dimensions. For the Signature Searchers, Tables 3.18 through 3.26 enumerate the nine parameters' values for this same type of name scheme, but not repeating code letters. A name string then matches the pattern "`***-*-**-***`," with the parameters in this order:

1. Document grouping.

2. Application version grouping.

3. Path normalization.

4. n-gram length.

5. n-gram stop list strategy.

6. Training sequence.

7. Vector combinator.

8. Stop list.

9. Threshold selector.

Table 3.18: Document grouping (Section 3.3.3.2).

	Value	Description
A	*app*	By application, grouping operating systems together.
B	*osapp*	By operating system and application, creating a signature per operating system.

Table 3.19: Application version matching (Section 3.3.3.7).

	Value	Description
C	*distinct*	Applications of different versions not allowed to match.
D	*grouped*	Applications of different versions allowed to match.

This results in a $2 \cdot 2 \cdot 2 \cdot 7 \cdot 3 \cdot 3 \cdot 3 \cdot 4 \cdot 3 = 18,144$-run experiment, with one run performed for each combination of the factor values. Instead of a full-factorial

Table 3.20: Path normalization (Section 3.3.3.3).

	Value	Description
E	*raw*	All cell paths used in models exactly as extracted from hive files.
F	*normalized*	Cell paths normalized according to containing hive class, as in Tables 3.8 and 3.9.

Table 3.21: n-gram length (Section 3.3.3.4). A path component is one level of a Registry path, *e.g.* `foo\bar` has components `foo` and `bar`.

	Value	Description
WP	*all*	Whole cell path used as term.
A1	*1*	Cell path broken into individual components, contributes each component once as a term. (A path with n components contributes n terms.)
A2	*2*	Cell path broken into all in-order pairs of components, with each pair contributed once as a term. (A path with n components contributes n-1 terms.)
A3	*3*	Cell path broken into all in-order triplets of components, with each triplet contributed once as a term. (A path with n components contributes n-2 terms.)
L1	*last1*	Last component of the path contributed as a term.
L2	*last2*	Last two components of the path contributed as a term.
L3	*last3*	Last three components of the path contributed as a term.

Table 3.22: n-gram interaction strategy with stop lists (Section 3.3.4.3).

	Value	Description
G	*raw filter*	Stop list constructed from cell paths.
H	*n-gram black list*	Stop list constructed from path component black lists when Registry paths are broken into n-grams.
I	*n-gram threshold*	Stop list constructed from path component thresholds when Registry paths are broken into n-grams.

Table 3.23: Training sequence (Section 3.3.3.1).

	Value	Description
J	*installclose*	The data training the Searcher were all install-through-application-close diskprint sequences.
K	*repeated*	The data training the Searcher were all diskprint sequences that were performed multiple times. Referring to the manifest of sequences printed in Tables 3.1 and 3.2 these would be all the sequences, excluding the "1" entries.
L	*experiment1*	A control selection of diskprint sequences: Only diskprint sequences of applications used on the evaluation subject machine. Performance is expected to be overall better with this set, but the *a priori* knowledge is unrealistic in practice.

Table 3.24: Vector combinator (Section 3.3.3.5). The union combinator was not evaluated, though it could serve as a control, being expected to perform the worst.

	Value	Description
M	*intersection*	A term's query or document dimension is 1 if the term is present in each change set to be combined, 0 otherwise.
N	*summation*	A term's query or document dimension is the count of appearances in the combined change sets.
O	*sumint*	A term's query or document dimension is the product of the intersection and summation value for that term.

Table 3.25: Stop list (Section 3.3.4).

	Value	Description
P	*none*	No stop list.
Q	*baseline*	Information from the baseline machines' installation and idle running was used.
R	*bp*	Information from the baseline machines and running preinstalled applications was used.
S	*bps*	The baseline and preinstalled application data were joined with cells that appeared in the symmetric differences of applications that had been printed more than once, as described in Section 3.3.4.2.

Table 3.26: Threshold selector (Section 3.3.3.6).

	Value	Description
T	*min*	The minimum search score against the training machines in the ground truth of a signature was used.
U	*avg*	The average search score against the training machines in the ground truth of a signature was used.
V	*max*	The maximum search score against the training machines in the ground truth of a signature was used.

evaluation, however, two of the factors will have more qualitative analysis done first, for their fundamental effects on the interpretation of model results: The document grouping involving operating systems or not; and the n-gram length, focusing on whether cell paths should be broken into n-grams or not.

After measuring the Signature Searcher factors against the controlled machine, the second experiment measures recall against the M57-Patents applications reported by Roussev *et al.*, using the ground truth defined in Table 3.7. This experiment will not focus on precision, because the total set of applications installed on the M57 images is a superset of what Roussev *et al.* reported. We focus only on reproducibility of their results, using only the Registry where they used similarity hashing with files with intact names (for file extensions), RAM, and at times network capture.

3.6 Assumptions of the Study

The Diskprint data generation frequently assumed that Registry artifacts were steadily flushed from RAM. If the assumption is false, the impact would be that Registry cells created for a snapshot would not be picked up from the disk state until the subsequent snapshot. Example effects would be incorrectly attributing a cell associated with software installation with software execution, or missing a software execution cell. This assumption can be difference-tested in future studies against the in-RAM Registry state from the virtual machines.

The evaluation subject machine described in Section 3.2.2 had a baseline operating state that had been networked. To deploy software on the machine, networking was disabled. It was assumed that Registry artifacts would be created regardless of network effects. The Diskprint sequences were created with networking enabled.

The Diskprint data are used twice in training Signature Searchers, once for building the TFIDF matrices, and again for determining signature thresholds. The objective is to check the similarity of a training machine to its newly-appeared Registry paths, given the context of all other training machines. However, using the same machines can introduce a risk of over-fitting the Searchers to the training data. This concern can be addressed by independently creating another set of machines, at least one for each application and operating system desired. This can also be measured as another design parameter. However, for now this is left to future study, as any overfitting can be indirectly observed by comparing Signature Searcher performance between the two subject data sets in the evaluation.

3.7 Limitations of the Study

The Diskprint training data do not have much repetition when they are repeated, because the current process is time consuming. Methods exist to create far greater sets of repetitions once an initial diskprint is performed, such as done by Kälber *et al.* [32], though their method still requires manual labor required to create the initial print for scripting. The current study includes some variance within the training data, though it could include more.

The study's design limits the ability to make statistical claims with the main effects analysis. The evaluation runs one experiment many times, on fixed data from two sources: the evaluation subject machine, and the M57-Patents corpus. Typically, in a full-factorial analysis, an experiment is run with the factors as input variables, measuring their response on a sample from a larger population. Due to a lack of randomness in the fixed input of the corpora, this study's parameter effect claims can only be in performance ranking versus other parameter values. This limitation is part of a larger problem regarding the lack of annotated data in digital forensics, discussed in the Future Research section. If the problem of data availability were solved, a further philosophical challenge arises in defining what a *representative sample* of computer system states would be. That challenge is outside of the scope of this thesis. However, the basic research task of this thesis—demonstrating feasibility of document search-based software signatures—is still demonstrable with these limitations.

3.8 Summary

This study converts the Vector Space Model commonly used in document search into a binary classifier, which takes as input an entire Registry and outputs a set of applications believed to have been installed and/or run. Search provides a way to automatically assign relevancy scores to forensic artifacts, a different strategy from the manual artifact classification undertaken in prior work. There are nine dimensions of options to assemble these Signature Searchers, which are enumerated in Tables 3.18 through 3.26. The evaluation will compare the optimal selection of Signature Searcher, according to an instrumented machine, against a corpus of machines that fit a significantly longer usage life than the instrumented machine.

Chapter 4

Research Findings

This chapter presents the Precision-Recall oriented evaluation of the Signature Searchers. At first, the evaluation plots include the “Control” Searcher training sequence-set “experiment1,” which show Signature Searchers’ performance when they are trained precisely on the applications included in the evaluation subject machine. The evaluation then drops the control set in order to inspect successful parameter combinations between the evaluation subject machine and the M57-Patents corpus, without perfect *a priori* selection of training data.

Since the forensic process taken to go from virtual machine states to Searchers involves several steps, first the processing integrity of the workflow is measured.

4.1 Extraction workflow processing statistics

Each of the three data sets—Searcher training, the evaluation machine states, and M57—has a set of virtual machine images that need their contents extracted in order to perform experiments. The extraction workflow analyzed each of the disk images, running *Fiwalk* and *RegXML Extractor* to extract the Registry content. Because forensic extraction is not yet a perfected science, there is a *survival curve* to measure at each step of the workflow. A survival curve is usually used to illustrate how many subjects remain within an experiment until its conclusion. In the extraction workflow, artifacts may not reach a state where they can be analyzed because data can be lost at each workflow step from some process failing. In particular, the points of concern are:

- Failure to parse the file system with *Fiwalk* [22];
- Failure to extract files with the metadata extracted by *Fiwalk*;
- Failure to parse extracted hive files with *hivexml* [30]; and
- Failure to parse *hivexml*-generated XML—*e.g.* due to binary data causing XML parsing to fail (parsing failed consistently with Python's *ElementTree* library [57] and the *xmllint* XML integrity-checking tool [68]); and
- Failure to convert the XML of *hivexml* to RegXML [40].

Table 4.1 provides the counts of the forensic survival curve for each of the training diskprint images, the experiment virtual machine states, and the M57-Patents corpus. This is an update of the *RegXML Extractor* functional evaluation table, which had measured XML conversion and processing statistics for M57 and another data corpus [40, Table 2].

Table 4.1: Summary of data extracted from disk images into Signature Searchers. Points in the process where data can be lost to further analysis, *e.g.* translating between *hivexml* output and RegXML [40], have a minus sign annotation.

	Training	Evaluation	M57
Media images	181	21	79
Images with successful hive metadata extraction	181	21	79
Hives found	2,346	231	1,297
Hives extracted with matching SHA-1	2,322	231	1,297
(-) Hives extracted with SHA-1 discrepancy	24	0	0
Hives that hivexml could process	2,328	231	1,292
(-) Hives that hivexml could not process	18	0	5
(-) Hivexml files that xmllint could not process	0	0	0
(-) Hivexml files that dfxml.py could not process	0	0	0
(-) Cells discarded from dfxml.py failing	0	0	0
Total images in end analysis	181	21	79
Total hives in end analysis	2,328	231	1,292
Total cells in end analysis	79,458,145	14,190,751	28,055,999

For currently-unknown reasons, some of the hives failed a checksum verification after extraction, even though they were all allocated. The version of *RegXML Extractor* used in the analysis has what could be considered (with some generosity) a feature, where processing those hives was attempted regardless of the failed extraction. Six of those hives succeeded in further processing, and since none of the hives that failed extraction were deleted files, the results from those six hives are in scope for inclusion in the study.

The *hivexml* failures merit further attention here, in case some flaw in *hivexml* prevents an entire class of hives from being analyzed. Table 4.2 breaks out the *hivexml* failures by file path, also including the number of successfully processed hives to show whether issues are systematic. A *systematic* failure would be all hives failing to parse. This had occurred earlier in the research due to an out-of-date version of *hivexml* that failed due to a character encoding issue. Now, all the hive failures are accompanied by an outnumbering number of successes in all cases but the Terry NTUSER hives found on the "Pat" Windows XP machine later in the scenario. Those failed Terry hives were files that had been reallocated, and the deleted files had been overwritten by Pat's operating system. These failed hives have no impact on the analysis, since the study is scoped to analyze only not-deleted data.

Table 4.2: Report of which hives, by file path, *hivexml* failed to parse. For magnitude reference, the number of successful parsings by *hivexml* are reported alongside the number of failed parsings. Failures among the diskprint data are due to non-ASCII data appearing in the XML, causing integrity checking and processing libraries—*xmllint* and *ElementTree*—to fail.

Corpus	Hive file path	Successes	Failures
m57	Documents and Settings/Terry/NTUSER.DAT	5	5
training	Documents and Settings/nsrl-admin/NTUSER.DAT	48	1
training	Users/nsrl-user/NTUSER.DAT	97	5
training	Users/nsrl/NTUSER.DAT	28	2
training	WINDOWS/system32/config/software	45	4
training	Windows/System32/config/SOFTWARE	126	6

4.2 Overall Signature Searcher results

A general inspection of correctness levels of the Signature Searchers can start with visualizing precision and recall effects. Figure 4.1 plots a point in Precision-Recall space [11] for each Searcher, measuring precision and recall against the evaluation subject machine. Histograms along the axes show the overall distribution of the Searchers' precision and recall. This inspection shows some Searchers attain perfect precision, and some perfect recall, but none attained both.

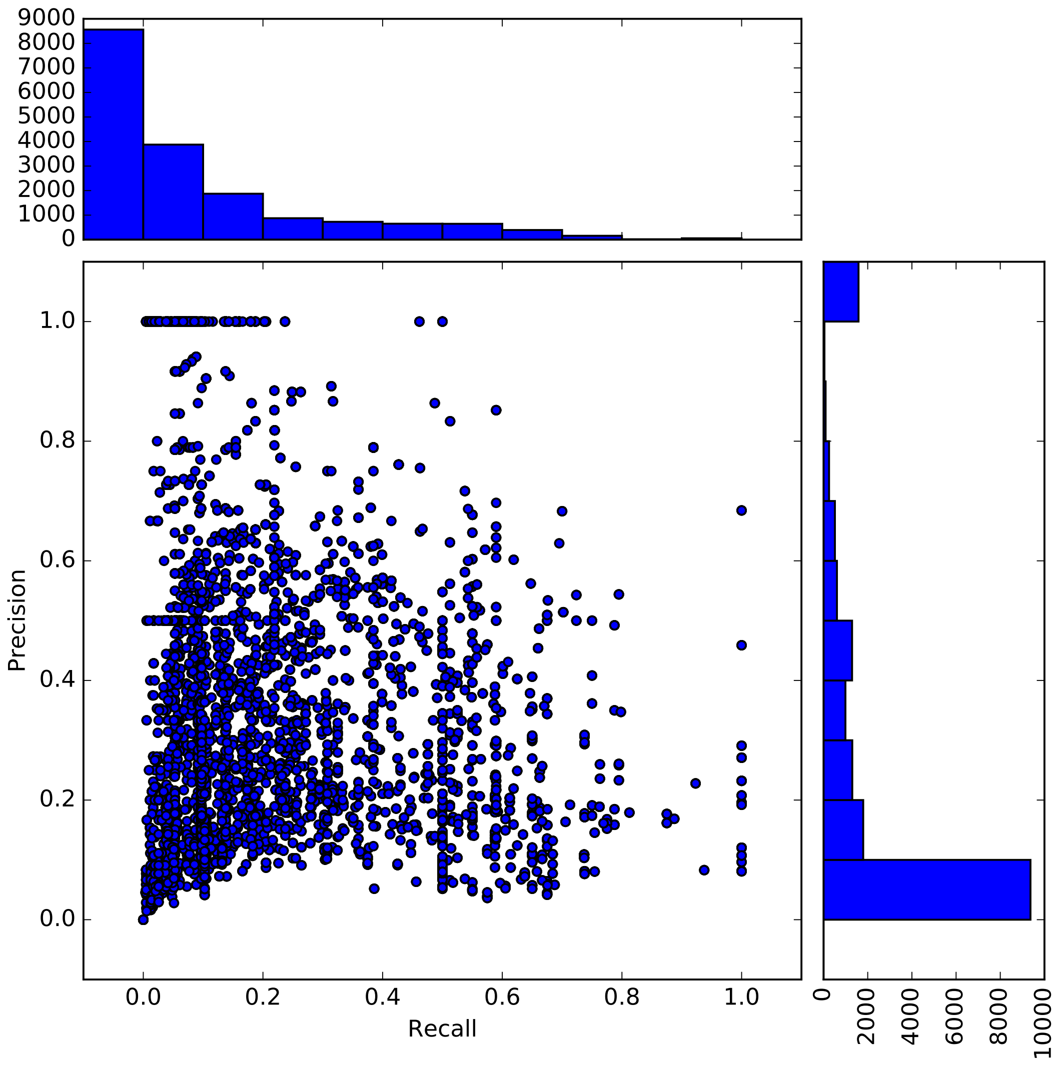


Figure 4.1: Scatter plot of precision-recall scores of Signature Searchers measured against the evaluation subject machine states, with axis histograms. Each point is a single Signature Searcher, as determined by parameter selection (Tables 3.18–3.26). The goal point in precision-recall space is $(1, 1)$, representing perfect precision—no false positives—and perfect recall—where every document that should have been returned as a match was returned as a match. This figure shows many Searchers attain perfect precision or perfect recall.

The “experiment1” control set has the potential to show unrealistic performance advantages from its diskprint training selection. To show the influence of the “experiment1” control set, Figure 4.2 shows where Searchers fall in precision-recall space, breaking out by the set of diskprints used to train each Searcher. Figure 4.4 shows the evaluation excluding the Searchers trained on the “experiment1“ set. The Searchers that have perfect precision or recall, and the highest of the other scoring factor, are not trained on the “experiment1” training set. Hence, inspection of the best perfect-precision models (and best perfect-recall models) is unaffected by the control training set.

Table 4.3: Number of documents for a Signature Searcher, according to training data and grouping of documents by just application or by application and operating system. The “experiment1” training set is provided here for reference, but not included in Figures 4.3 or 4.5.

Sequence set	Document grouping	Document count
experiment1	app	26
experiment1	osapp	44
installclose	app	79
installclose	osapp	135
repeated	app	15
repeated	osapp	23

To gauge whether any of the Searchers are using a small number of signatures for their scores, Figures 4.3 and 4.5 inspect *substantive document counts*, where a substantive document is a software signature containing more than zero terms. The number of substantive documents will always be less than or equal to the number of documents that can be produced, so while Figure 4.5 shows the substantive counts, Figure 4.3 shows the proportion of the possible documents per Signature Searcher that are substantive.

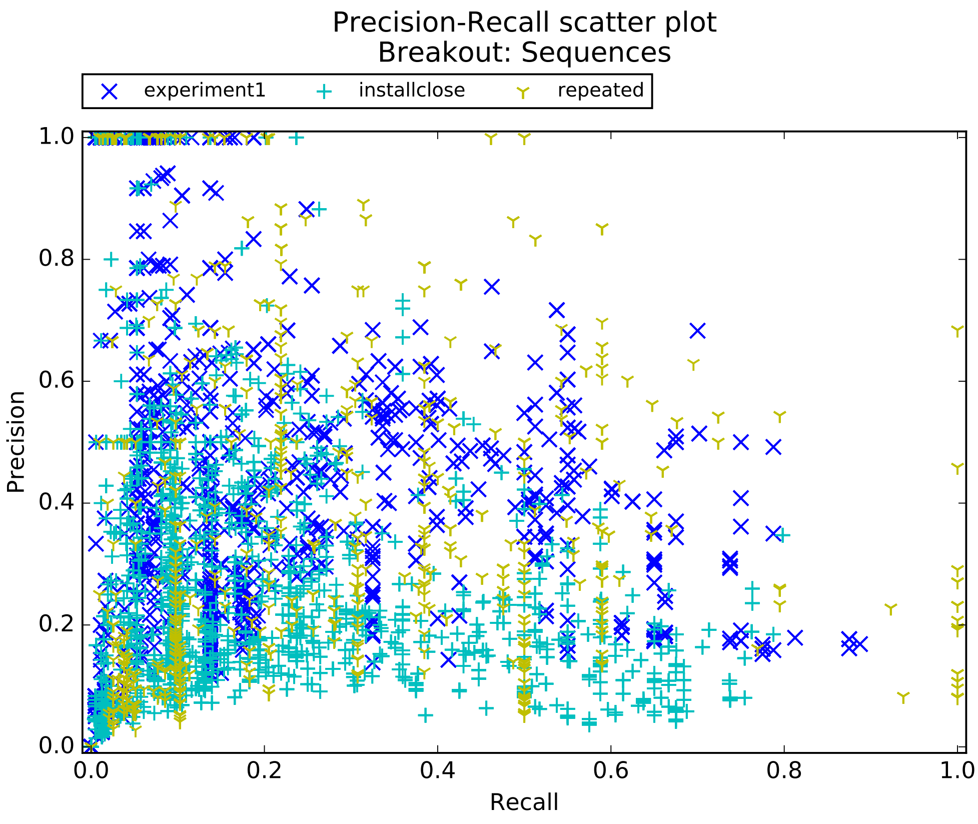


Figure 4.2: Scatter plot of precision-recall scores of Signature Searchers measured against the evaluation subject machine states, broken out by training sequences. For the evaluation subject machine, using only repeated diskprints enables perfect recall. Perfect precision was more often attained with the "installclose" training sequences.

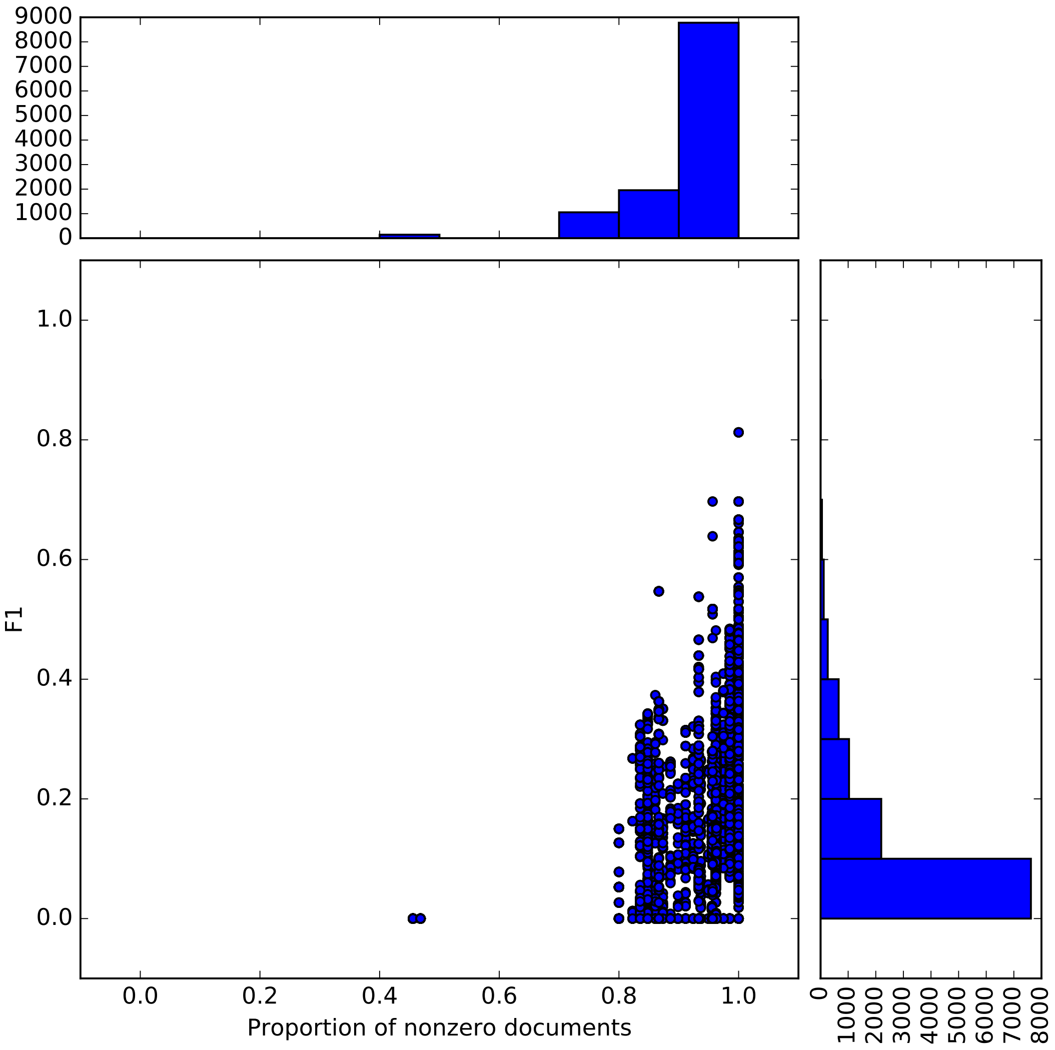


Figure 4.3: For each Signature Searcher, its F1 measured with the evaluation subject machine states versus its proportion of signature documents that are non-zero in length (*i.e.* have some number of terms with non-zero TFIDF weight). This plot does not include Searchers trained on the "experiment1" diskprint set. The Searchers' proportions of non-zero signatures almost entirely fell between 80 and 100%. The highest-F1 Searchers had the highest proportion of non-zero signatures.

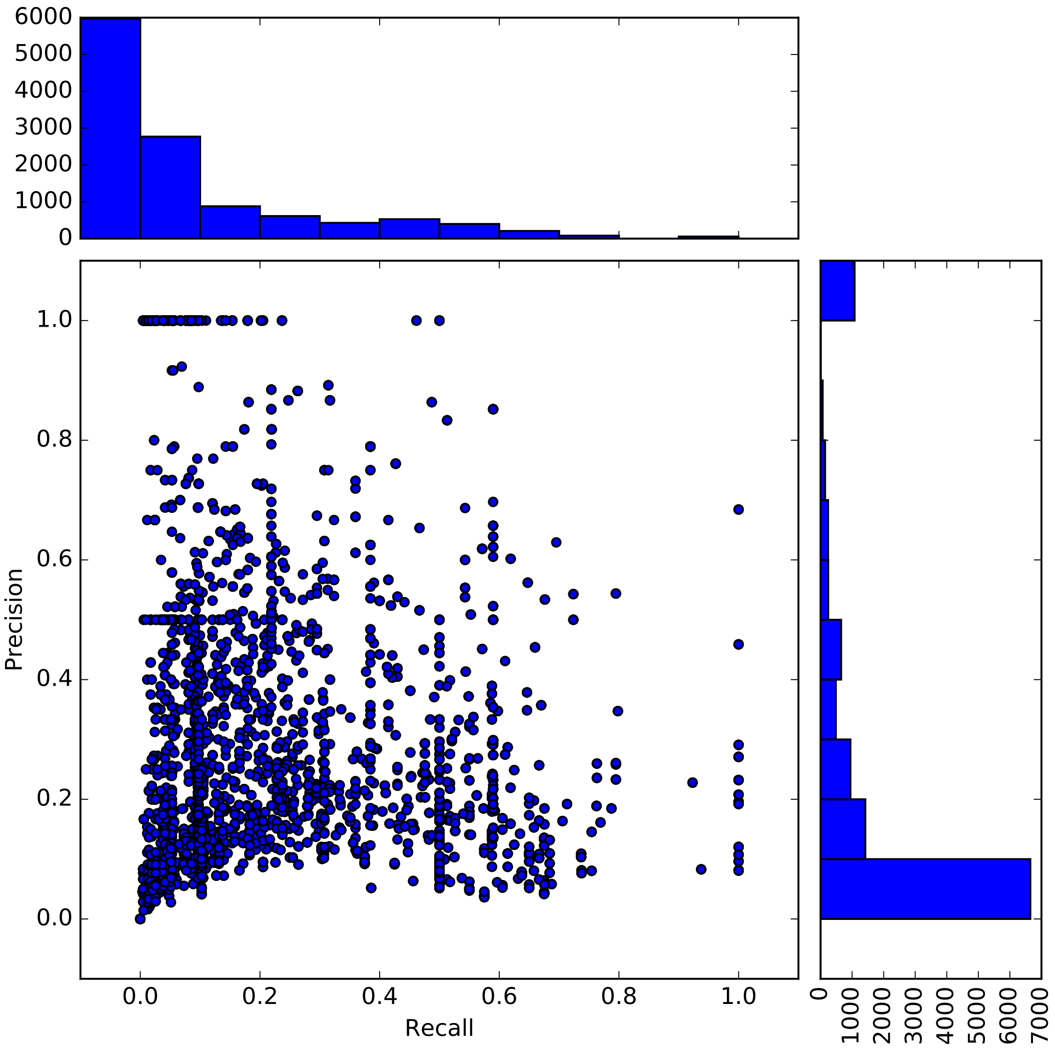


Figure 4.4: Scatter plot of precision-recall scores of Signature Searchers measured against the evaluation subject machine states, with axis histograms. Unlike Figure 4.1, this plot excludes Searchers trained on the "experiment1" training set.

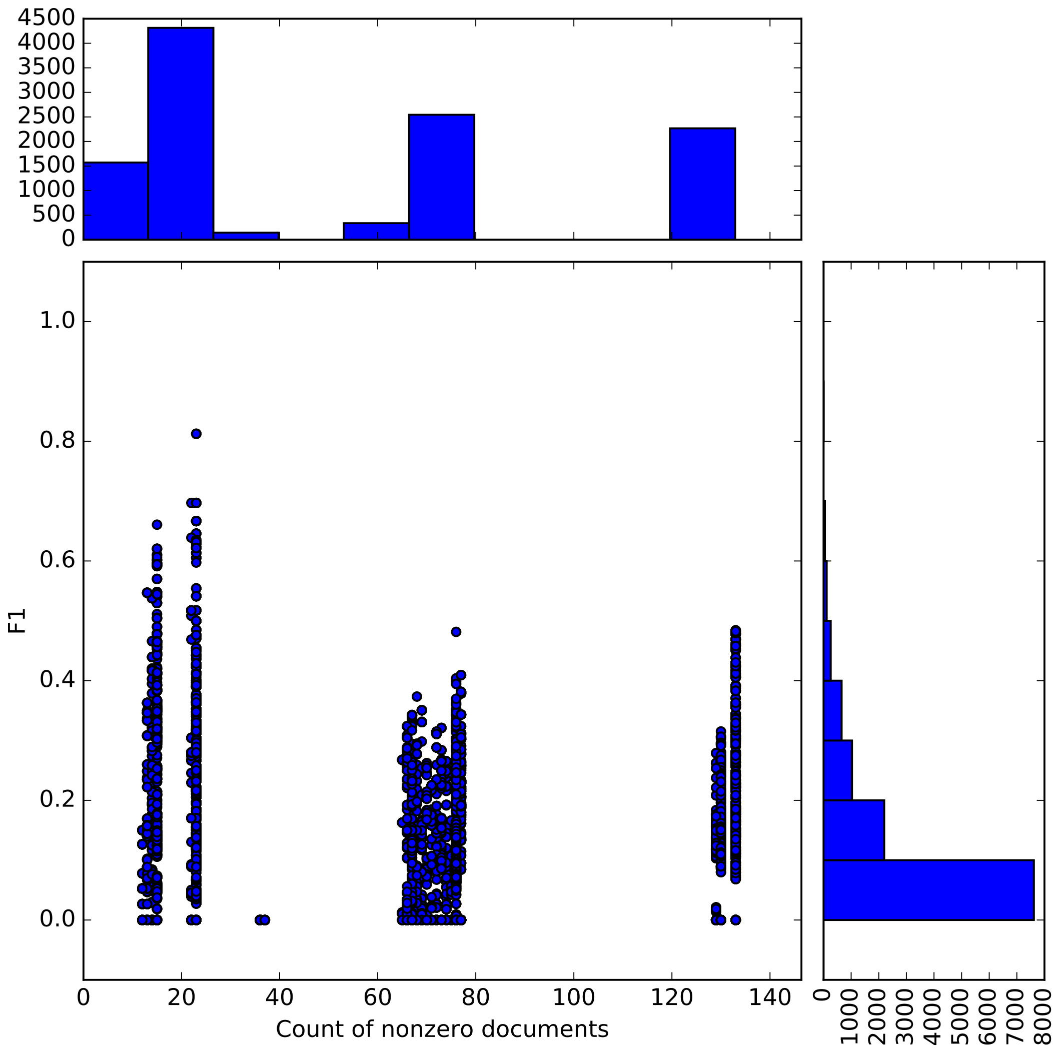


Figure 4.5: For each Signature Searcher, its F1 measured with the evaluation subject machine states versus its count of signature documents that are non-zero in length (*i.e.* have some number of terms with non-zero TFIDF weight). This plot does not include Searchers trained on the "experiment1" diskprint set. This figure, and the document counts in Table 4.3, give context to the proportions in Figure 4.3.

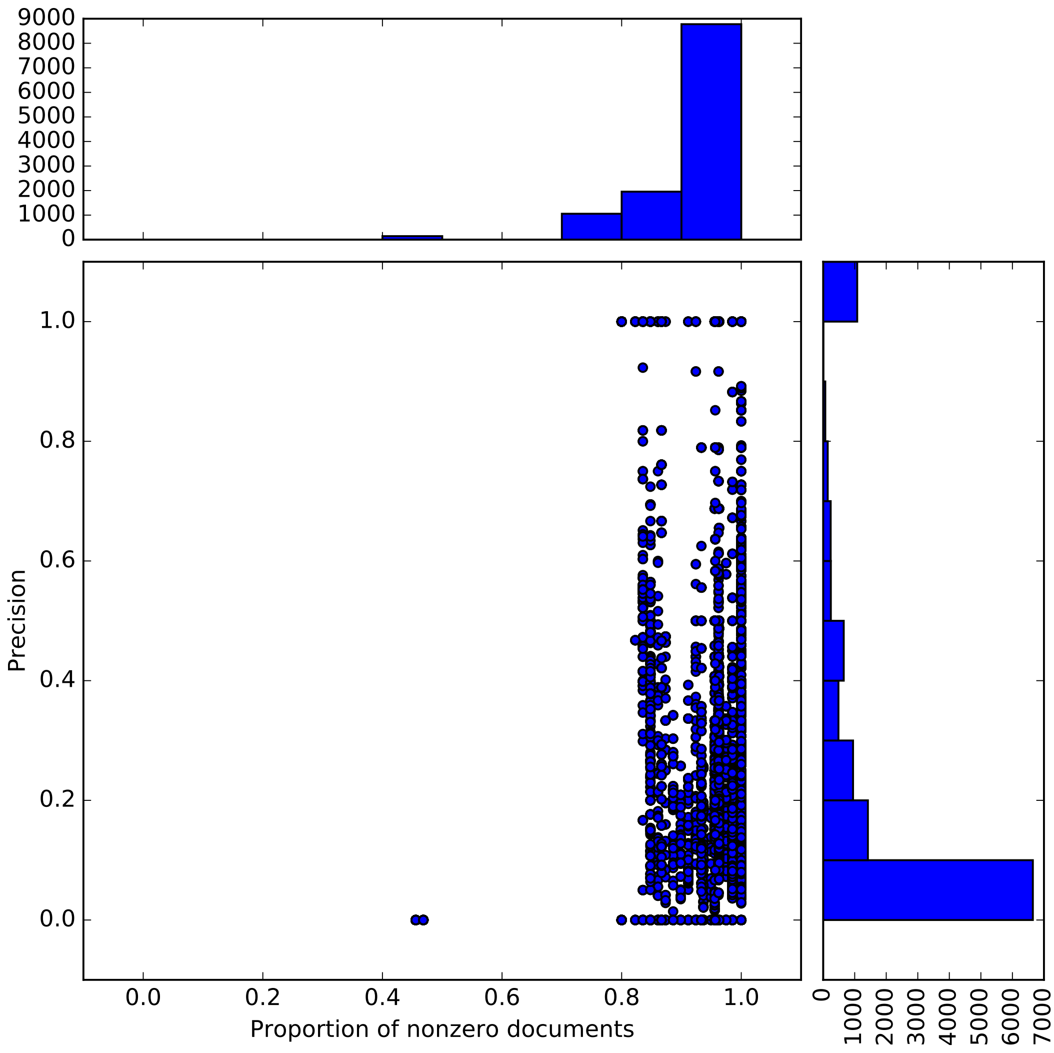


Figure 4.6: The precision for each Signature Searcher, measured with the evaluation subject machine states versus its proportion of signature documents that are non-zero in length (*i.e.* have some number of terms with non-zero TFIDF weight). This plot does not include Searchers trained on the "experiment1" diskprint set. This plot is as Figure 4.3 for F1, but shows that perfect precision is achievable with most proportions of non-zero documents.

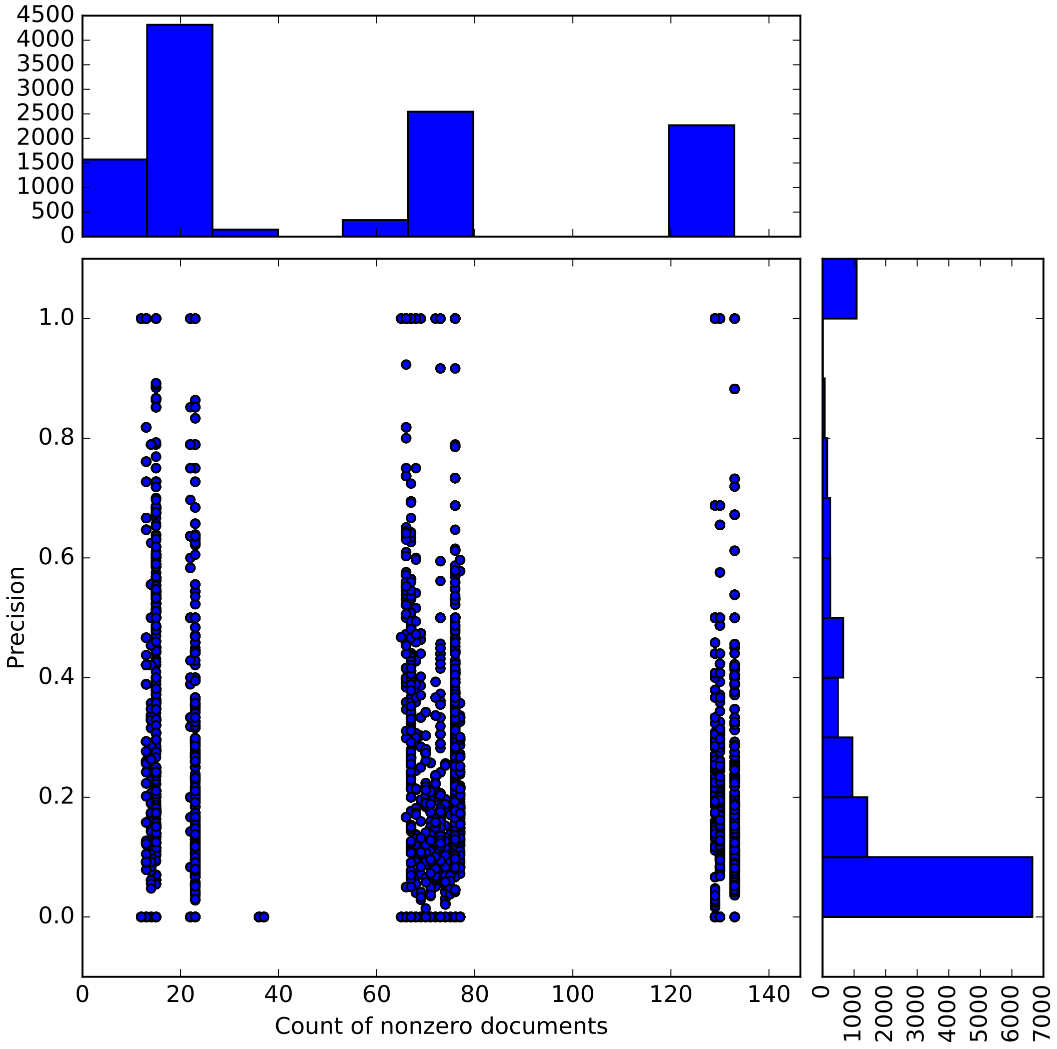


Figure 4.7: The precision of each Signature Searcher, measured with the evaluation subject machine states versus its count of signature documents that are non-zero in length (*i.e.* have some number of terms with non-zero TFIDF weight). This plot is as Figure 4.5 for F1, but shows that perfect precision is achievable with Searchers of all clusters of document counts.

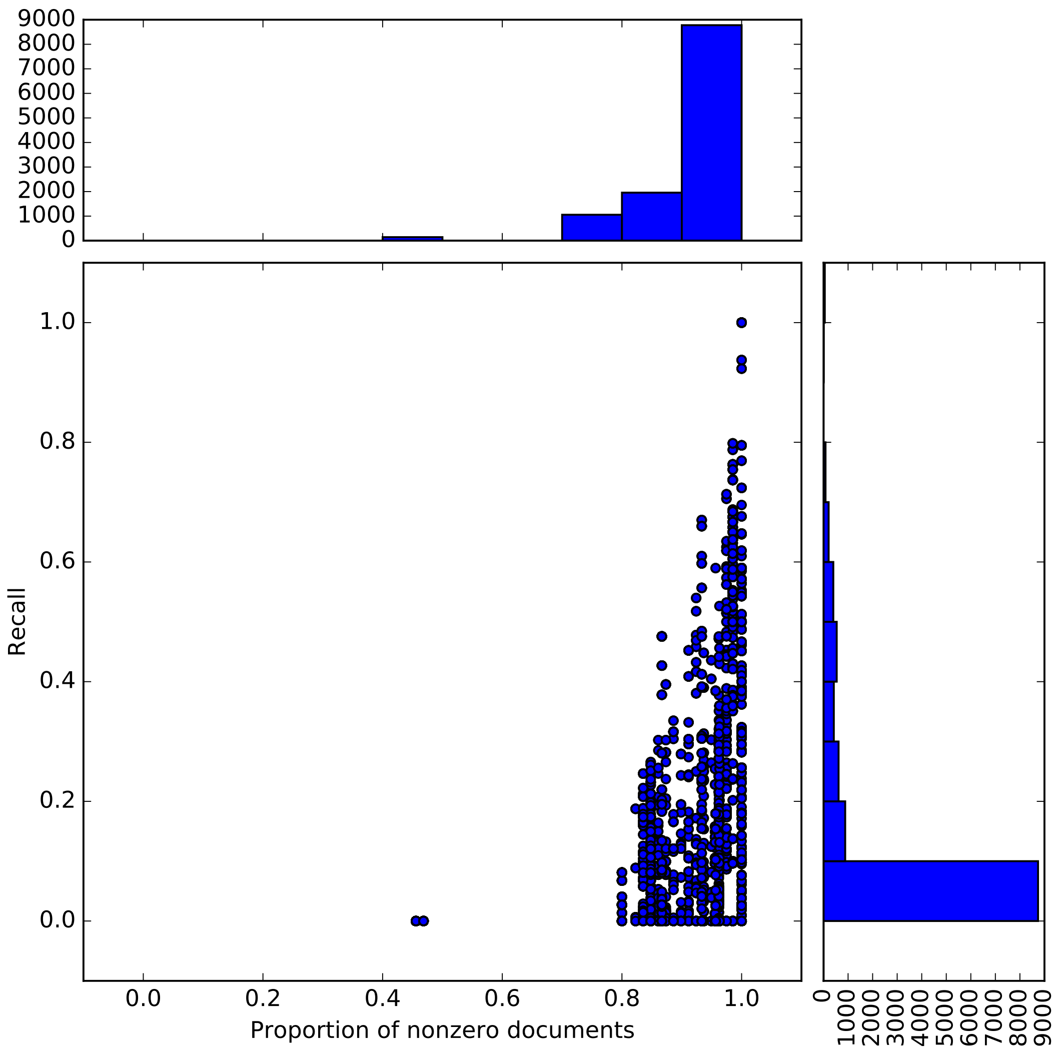


Figure 4.8: The recall for each Signature Searcher, measured with the evaluation subject machine states versus its proportion of signature documents that are non-zero in length (*i.e.* have some number of terms with non-zero TFIDF weight). This plot does not include Searchers trained on the “experiment1” diskprint set. This plot is as Figure 4.3 for F1, but shows that perfect recall was only attained with Searchers that had no zero-length documents in their construction.

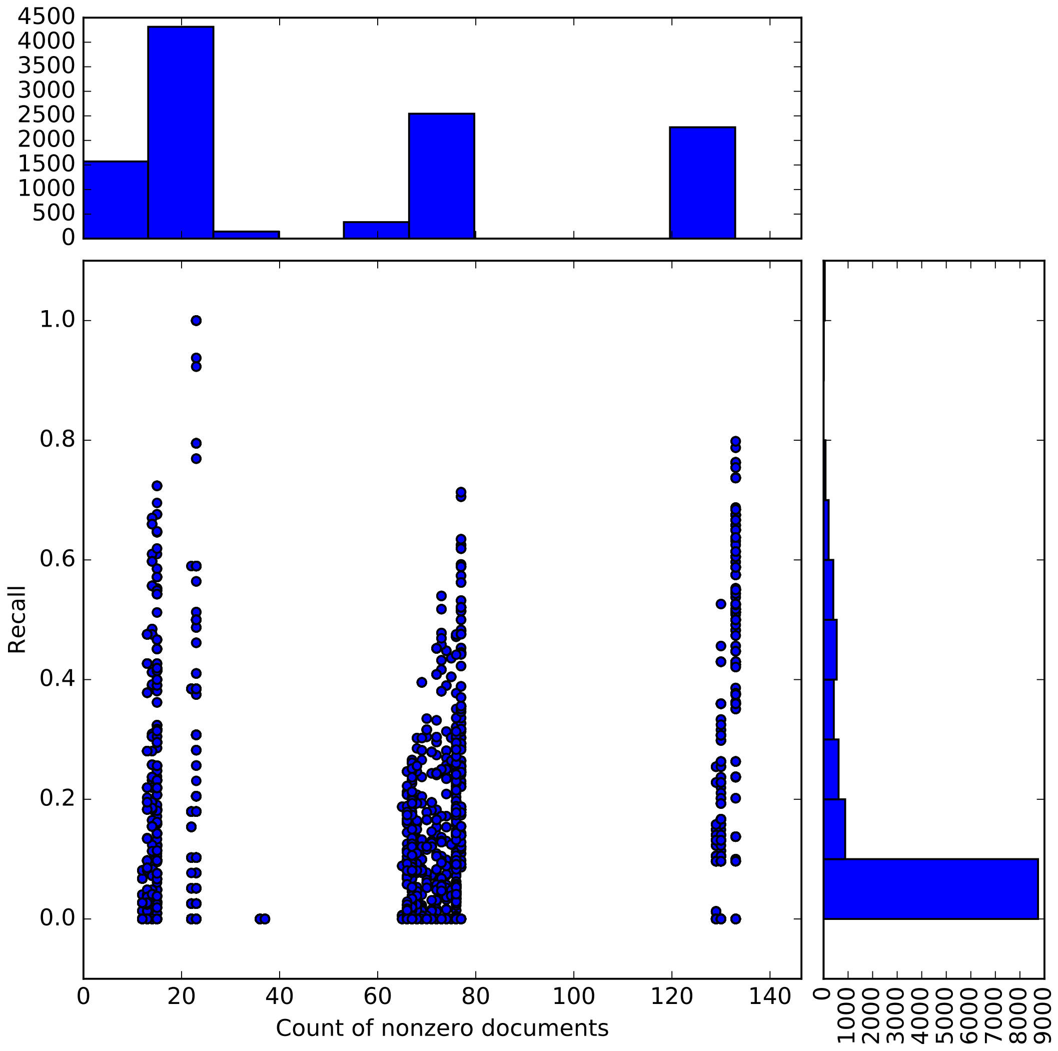


Figure 4.9: The recall of each Signature Searcher, measured with the evaluation subject machine states versus its count of signature documents that are non-zero in length (*i.e.* have some number of terms with non-zero TFIDF weight). This plot is as Figure 4.5 for F1, but shows that perfect recall was only attained with Searchers that have a low number of documents.

Since the precision-recall overview shows none of the Searchers simultaneously attained perfect precision and recall, precision and recall get the same substantive document inspection in Figures 4.6 through 4.9.

The substantive signature figures show a few characteristics arise based on how many of the possible signatures a Signature Searcher could produce do get produced.

- Figure 4.5 shows the F1 scores have a higher-reaching F1 range for Searchers with a higher ratio of substantive signatures. However, Figure 4.3 shows that the highest F1 scores only occur when the substantive signature count is smaller. Thus, if using F1 as the metric of Searcher correctness, the most-correct Searchers have the second-fewest signatures, according to the count clustering.
- If instead prioritizing precision, nearly any count or ratio of substantive signatures will do. Figures 4.6 and 4.7 show perfect-precision Searchers appear in all clusters of signature counts and ratios.
- The Searchers that favor perfect recall are more selective. Those Searchers produce only substantive signatures (per Figure 4.8) and are based on the fewest signatures (per Figure 4.9).

For context on why the substantive signature counts group into clusters in Figure 4.5 (as well as 4.7 and 4.9), consider that from the parameter space, two factors directly affect the count of signatures: The training sequence, and the document grouping. From Tables 3.1 and 3.2, the counts of diskprints performed show the upper limit of the number of documents per parameter pairing. Table 4.3 gives the maximum number of documents expected.

4.3 Main effects inspection

Figure 4.10 provides a main effects plot [5] of all of the Signature Searchers run against the evaluation subject machine. Each plot compares the average value of a response variable—precision, recall, and F1 in the top, middle, and bottom row, respectively—for each parameter value when grouping all of the Signature Searchers with those parameter values. For instance, with the document grouping variable ("Docs by"), grouping by application instead of operating system with application shows an increase in precision, decrease in recall, and overall decrease in F1.

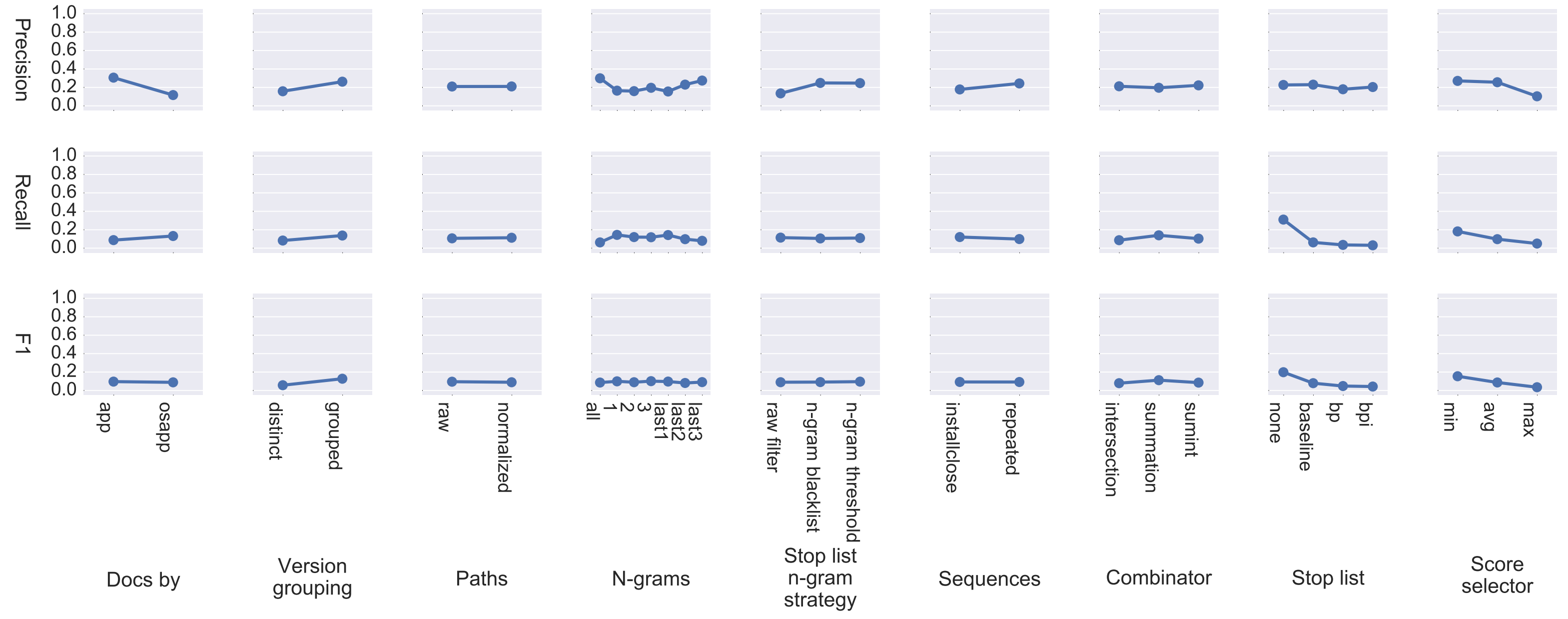


Figure 4.10: Main effects overview plot for precision, recall, and F1. Here, all Signature Searchers were run against the evaluation subject machine. One result shown by the averages comes from the n-grams column: the Signature Searchers built on whole Registry paths—instead of any sort of n-gram of path components—have the best precision on average, but the worst recall, resulting in a weak F1 score compared to most n-gram breakouts.

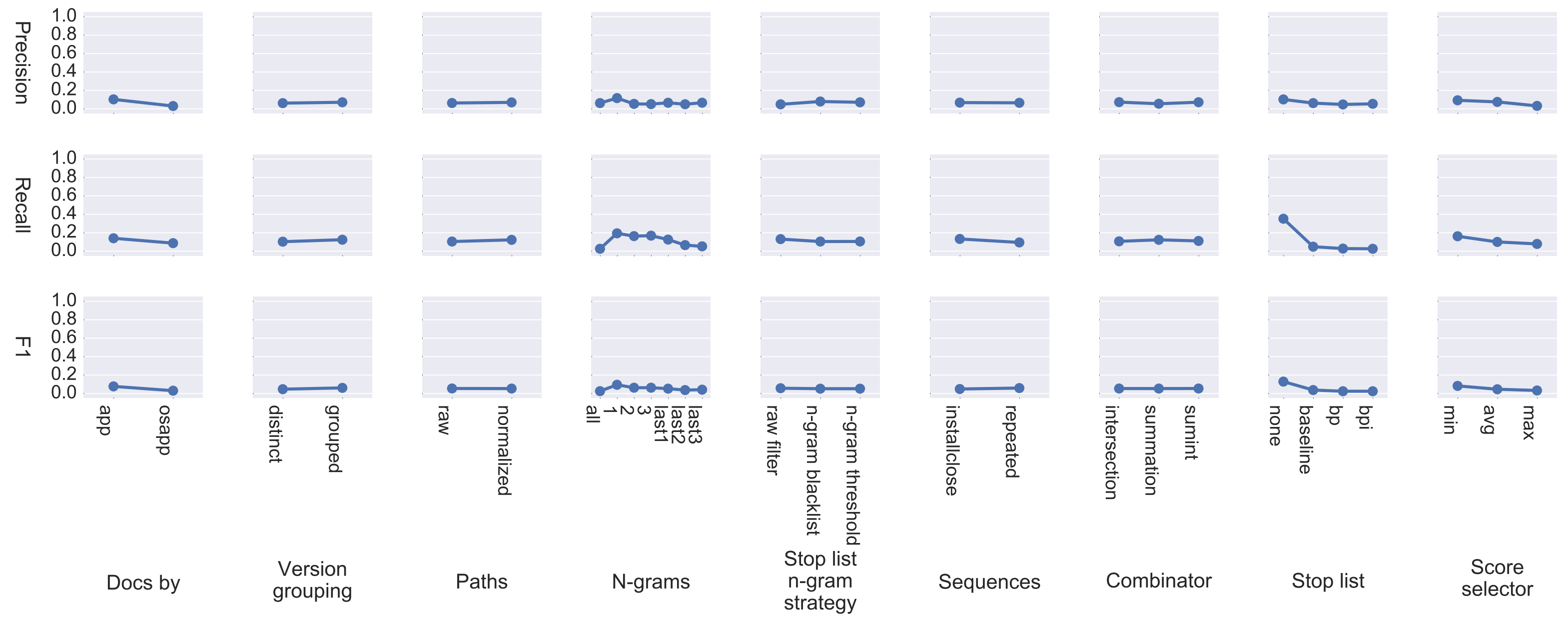
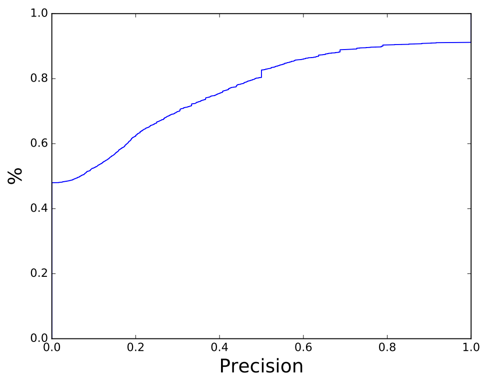


Figure 4.11: Main effects overview plot for precision, recall, and F1. Here, all Signature Searchers were run against the M57-Patents corpus. Average response variable values here differ in some cases from the evaluation subject machine used in Figure 4.10. For instance, for M57 the Searchers that tailored signatures to operating systems had a lower average recall (seen in the first column), but for the evaluation subject machine the Searchers with operating system agnostic signatures had the lower average recall.

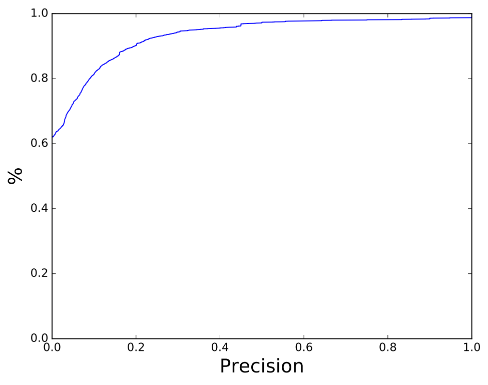
Figure 4.11 also provides a main effects plot, measuring the Signature Searchers against the M57-Patents corpus. Comparing the average precision and recall scores of each parameter value when measuring against M57, the relative average performance—*i.e.* which parameter value has the higher average score—changes for these parameters:

- Document grouping favors per-application grouping in recall, and thus in general since precision already favors per-application grouping. With the evaluation subject machine, operating system grouping has better recall.
- Version grouping still favors grouping application versions, but by a much more narrow margin.
- Whole-path terms now provide the worst average precision, and unigrams the best, a reversal from the evaluation subject machine.
- The evaluation subject machine average precision was best when using the "experiment1" control sequence set. Within M57, the control set had average precision on par with the other sequence sets.

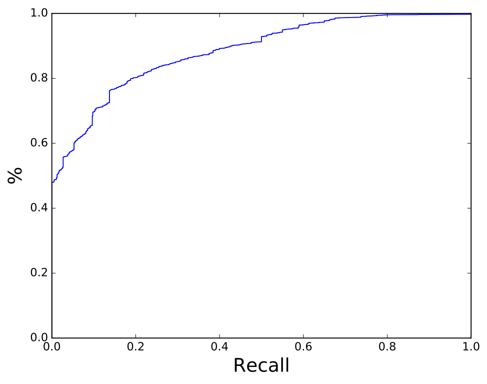
The main effects plots show which parameter values perform the best on average, considering the few well-performing models alongside the much more numerous poorly-performing models. To illustrate the score distributions, Figure 4.12 shows the cumulative histogram scores of all of the Searchers' performance measures against both corpora. Note that precision's upper-right gap in both corpora show that false positives are a surpassable challenge with the right Searcher parameters, perfected for 8.9% and 1.3% of all the Searchers against the evaluation subject machine and M57, respectively.



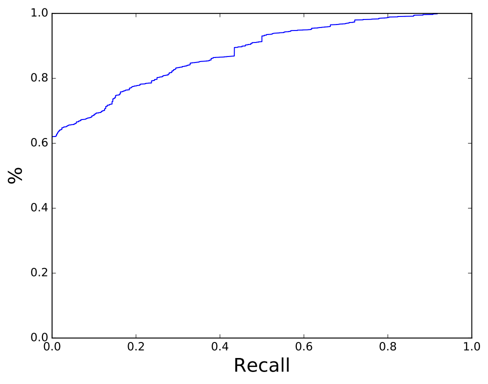
(a) Evaluation precision.



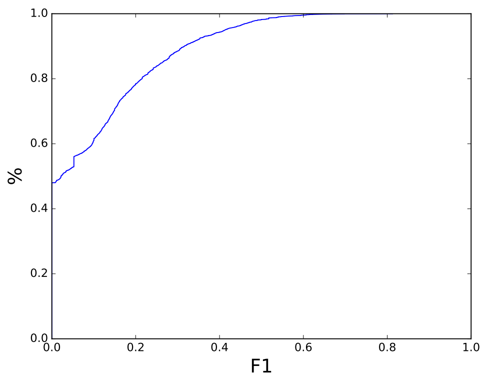
(b) M57 precision.



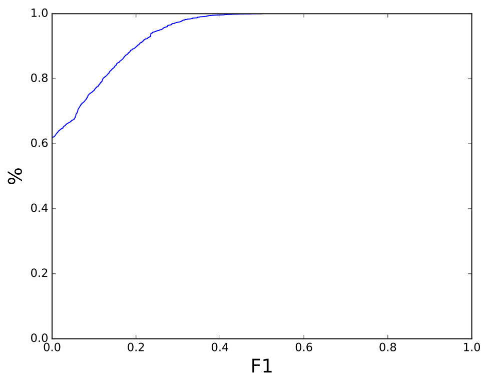
(c) Evaluation recall.



(d) M57 recall.



(e) Evaluation F1.



(f) M57 F1.

Figure 4.12: Cumulative histograms of precision (top), recall (middle), and F1 (bottom) scores for the Signature Searchers run against the evaluation machine states (left) and M57 data (right). Note that the apparent discontinuities at precision=1 are gaps between the penultimate scores and the scores of 1 that some Searchers attained.

4.4 Parameter inspection

This section onward excludes the Signature Searchers built on the "experiment1" set of diskprint training sequences.

Average parameter value performance, as shown in the main effects plots, is one method of determining how best to build a Signature Searcher. However, it may also prove produent to identify the current best Searchers, according to inherently desirable characteristics. For instance, two of the parameters affect basic interpretation questions for the Signature Searchers. The document grouping determines whether Searchers are better built for applications paired with an operating system, or can be built on any operating system. The n-gram parameter inspection will show whether whether there is a consistently performant Searcher configuration that uses whole Registry paths, which would simplify reporting for an investigator.

The main effects plots suggests that Searchers that make signatures for applications, agnostic of operating system, overall have stronger performance metrics, save recall with the evaluation subject machine. Likewise with Searchers built on unigrams, save precision with the evaluation subject machine. To visualize the independent effects of these two parameters deeper than a main effects plot, Figures 4.13 and 4.14 show where each Searcher falls in Precision-Recall space, with Figure 4.13 showing the document grouping breakout and 4.14 the n-gram breakout.[1] Those figures show that the average performance scores are not necessarily indicators of the best parameter values, since for example the Searchers that grouped by application performed on average

[1]For completeness, Appendix A provides the breakouts of the other parameters.

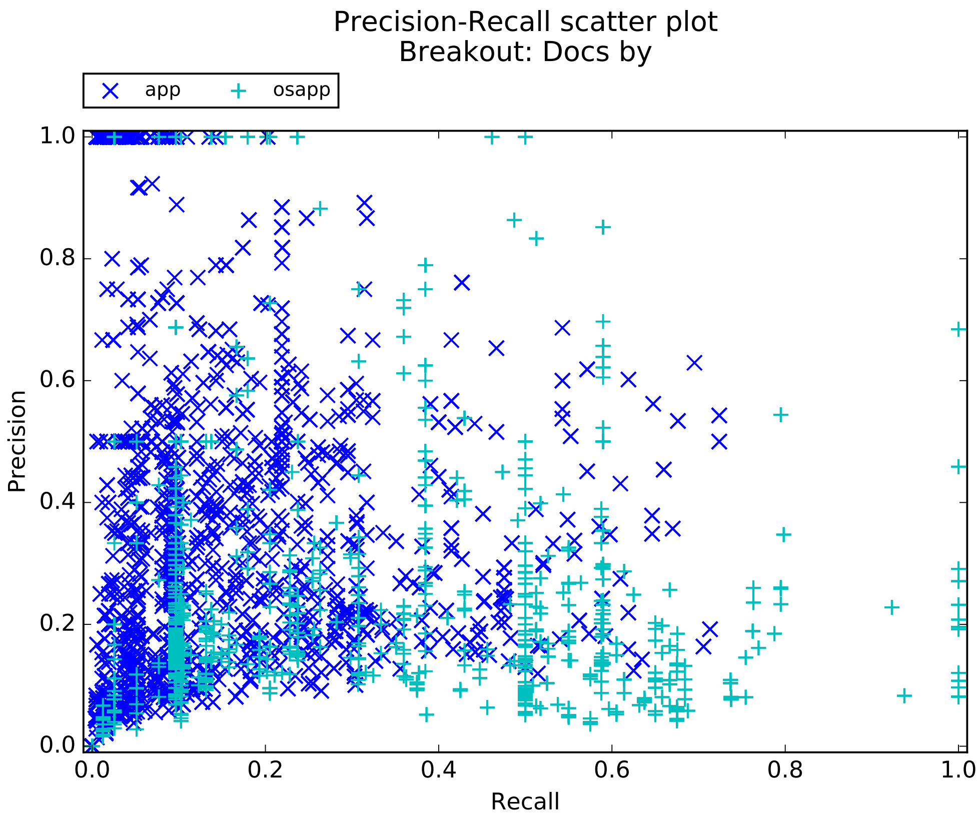


Figure 4.13: Scatter plot of precision and recall of Signature Searchers measured with the evaluation subject machine states, broken out by document grouping. The clump of plusses on the right show tailoring signatures to operating systems increases recall.

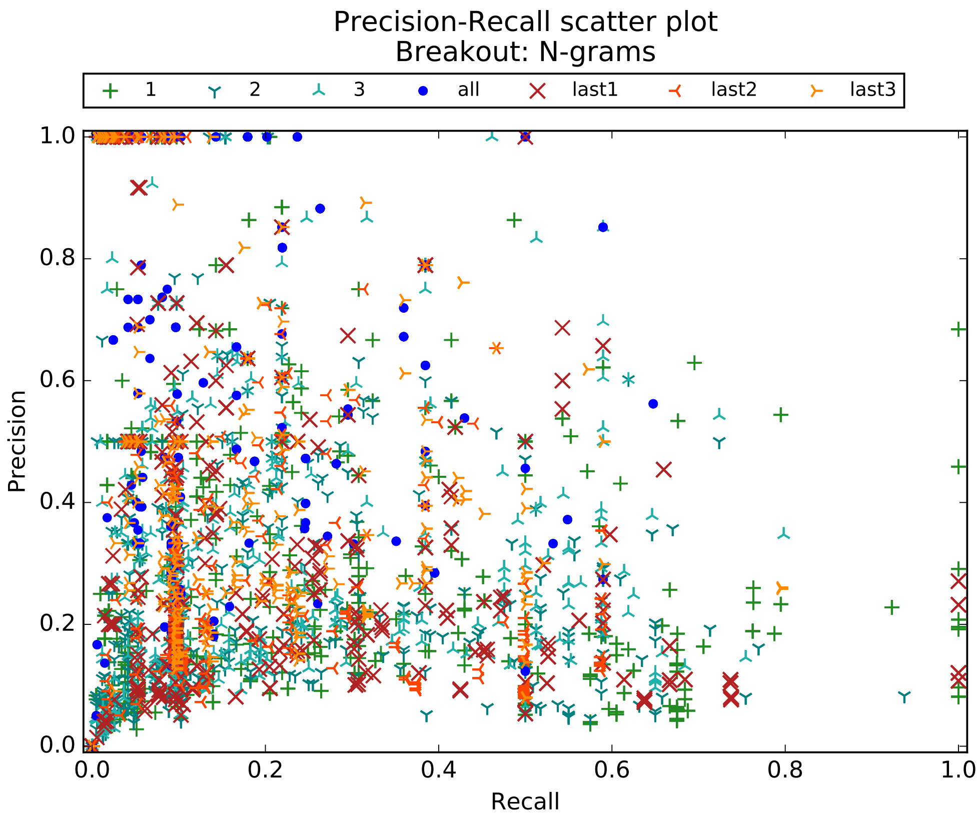


Figure 4.14: Scatter plot of precision and recall of Signature Searchers measured with the evaluation subject machine states, broken out by n-gram length, with “all” denoting Registry paths were not broken into n-grams.

better, but were not the best performing for precision or recall.

To test for top and consistent Signature Searcher performance, a ranking evaluation measures both the evaluation subject machine's states, and the M57 corpus. Since Searchers performed perfectly for only precision or recall and not both, there will be a ranking for those who would prioritize precision, and another for recall.

Tables 4.4 and 4.5 show the top twenty Searchers for the two parameters' values, ranked by precision percentile against the evaluation subject machine. Tables 4.6 and 4.7 do the same for recall.[2] To show consistency with the M57 data and the reported results of Roussev *et al.*, the tables also show whether a top-ranking Searcher against the evaluation subject machine is also in a top percentile of the Searchers run against M57. The number of documents reported is the number of substantive signatures. The number of runs reported is the number of machine states tested against a document in the ground truth of the respective corpus. Figure 4.12 relates the absolute scores and the score percentiles.

Figure 4.12 showed there were different amounts of Searchers that attained perfect precision and recall in both corpora—in particular, that more Searchers attained perfect precision. That is reflected in the top rankings of consistent Searchers. If the Searcher configuration priority is on recall, then there were no Searchers that had perfect recall between both corpora, as shown by the top entries of Tables 4.6 and 4.7 not having a recall of 1.0 for both corpus columns. If instead the Searcher production priority is on precision, then there are a number of searchers that attain a perfect precision score

[2]Searcher ID codes are in Tables 3.18 through 3.26 (pages 83–86).

Table 4.4: The top twenty Signature Searchers built on grouping by application (top table) vs. application and operating system (bottom table). Ranking is determined by precision against the evaluation subject machine, and compared to precision in M57.

Searcher ID	# docs.	# runs		Eval. precision		M57 precision	
		Eval.	M57	Prec.	Percentile	Prec.	Percentile
Grouping signatures by application							
ADF-WP-IJ-NSU	76	1448	3152	1.0	91.1	1.0	98.7
ACF-L1-IJ-OSU	68	1294	2806	1.0	91.1	1.0	98.7
ADF-L1-IJ-OSU	68	1294	2806	1.0	91.1	1.0	98.7
ACF-WP-IJ-OSU	67	1277	2783	1.0	91.1	1.0	98.7
ADF-WP-IJ-OSU	67	1277	2783	1.0	91.1	1.0	98.7
ACF-L1-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ADF-L1-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ACE-L1-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ADE-L1-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ACE-L1-IJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ADE-L1-IJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ACF-L2-HJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ADF-L2-HJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ACF-L2-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ADF-L2-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ACE-L2-HJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ADE-L2-HJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ACE-L2-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ADE-L2-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ACF-L2-IJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
Grouping signatures by application and operating system							
BDE-A1-GK-NQU	23	437	943	1.0	91.1	0.2	90.2
BDE-WP-HJ-MPU	133	2527	5453	1.0	91.1	0.19	89.6
BDE-WP-HJ-OPU	133	2527	5453	1.0	91.1	0.19	89.6
BDE-WP-IJ-MPU	133	2527	5453	1.0	91.1	0.19	89.6
BDE-WP-IJ-OPU	133	2527	5453	1.0	91.1	0.19	89.6
BDE-WP-GJ-MPU	133	2527	5453	1.0	91.1	0.19	89.6
BDE-WP-GJ-OPU	133	2527	5453	1.0	91.1	0.19	89.6
BDE-A1-HK-OPV	23	437	943	1.0	91.1	0.16	87.4
BDE-A1-IK-OPV	23	437	943	1.0	91.1	0.16	87.4
BDE-A1-GK-OPV	23	437	943	1.0	91.1	0.16	87.4
BDE-A3-HK-NPV	23	437	943	1.0	91.1	0.16	87.4
BDE-A3-IK-NPV	23	437	943	1.0	91.1	0.16	87.4
BDE-A3-GK-NPV	23	437	943	1.0	91.1	0.16	87.4
BDE-WP-HJ-NPU	133	2527	5453	1.0	91.1	0.16	86.9
BDE-WP-IJ-NPU	133	2527	5453	1.0	91.1	0.16	86.9
BDE-WP-GJ-NPU	133	2527	5453	1.0	91.1	0.16	86.9
BDF-A3-HK-OPV	23	437	943	1.0	91.1	0.09	79.9
BDF-A3-IK-OPV	23	437	943	1.0	91.1	0.09	79.9
BDF-A3-GK-OPV	23	437	943	1.0	91.1	0.09	79.9
BCE-L3-HJ-MPU	133	2527	5453	1.0	91.1	0.0	0.0

Table 4.5: The top twenty Signature Searchers built on n-grams (top table) vs. using whole Registry paths (bottom table). Ranking is determined by precision against the evaluation subject machine, and compared to precision in M57.

Searcher ID	# docs.	# runs		Eval. precision		M57 precision	
		Eval.	M57	Prec.	Percentile	Prec.	Percentile
Breaking Registry paths into n-grams for signature terms							
ACF-L1-IJ-OSU	68	1294	2806	1.0	91.1	1.0	98.7
ADF-L1-IJ-OSU	68	1294	2806	1.0	91.1	1.0	98.7
ACF-L1-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ADF-L1-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ACE-L1-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ADE-L1-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ACE-L1-IJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ADE-L1-IJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ACF-L2-HJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ADF-L2-HJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ACF-L2-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ADF-L2-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ACE-L2-HJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ADE-L2-HJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ACE-L2-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ADE-L2-HJ-OSU	67	1273	2747	1.0	91.1	1.0	98.7
ACF-L2-IJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ADF-L2-IJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ACE-L2-IJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
ADE-L2-IJ-MSU	67	1273	2747	1.0	91.1	1.0	98.7
Using whole Registry paths as signature terms							
ADF-WP-IJ-NSU	76	1448	3152	1.0	91.1	1.0	98.7
ACF-WP-IJ-OSU	67	1277	2783	1.0	91.1	1.0	98.7
ADF-WP-IJ-OSU	67	1277	2783	1.0	91.1	1.0	98.7
ACF-WP-HJ-MSU	65	1235	2665	1.0	91.1	1.0	98.7
ADF-WP-HJ-MSU	65	1235	2665	1.0	91.1	1.0	98.7
ACF-WP-HJ-OSU	65	1235	2665	1.0	91.1	1.0	98.7
ADF-WP-HJ-OSU	65	1235	2665	1.0	91.1	1.0	98.7
ACF-WP-IJ-MSU	65	1235	2665	1.0	91.1	1.0	98.7
ADF-WP-IJ-MSU	65	1235	2665	1.0	91.1	1.0	98.7
ACF-WP-GJ-MSU	65	1235	2665	1.0	91.1	1.0	98.7
ADF-WP-GJ-MSU	65	1235	2665	1.0	91.1	1.0	98.7
ACF-WP-GJ-OSU	65	1235	2665	1.0	91.1	1.0	98.7
ADF-WP-GJ-OSU	65	1235	2665	1.0	91.1	1.0	98.7
ADF-WP-IK-NSU	15	270	651	1.0	91.1	1.0	98.7
ADF-WP-IK-NST	15	270	651	1.0	91.1	1.0	98.7
ACF-WP-IJ-ORT	67	1275	2765	1.0	91.1	0.45	96.2
ACF-WP-IJ-OST	67	1277	2783	1.0	91.1	0.45	96.2
ACF-WP-HJ-MRT	66	1254	2706	1.0	91.1	0.45	96.2
ACF-WP-HJ-ORT	66	1254	2706	1.0	91.1	0.45	96.2
ACF-WP-IJ-MRT	66	1254	2706	1.0	91.1	0.45	96.2

Table 4.6: The top twenty Signature Searchers built on grouping by application (top table) vs. application and operating system (bottom table). Ranking is determined by recall against the evaluation subject machine, and compared to recall in M57.

Searcher ID	# docs.	# runs		Eval. recall		M57 recall	
		Eval.	M57	Rec.	Percentile	Rec.	Percentile
Grouping signatures by application							
ADF-A2-HK-NPT	15	285	615	0.72	98.8	0.74	98.0
ADF-A2-IK-NPT	15	285	615	0.72	98.8	0.74	98.0
ADF-A2-GK-NPT	15	285	615	0.72	98.8	0.74	98.0
ADF-A3-HK-NPT	15	285	615	0.72	98.8	0.74	98.0
ADF-A3-IK-NPT	15	285	615	0.72	98.8	0.74	98.0
ADF-A3-GK-NPT	15	285	615	0.72	98.8	0.74	98.0
ADF-A2-HJ-NPT	77	1463	3157	0.71	98.7	0.86	99.3
ADF-A2-IJ-NPT	77	1463	3157	0.71	98.7	0.86	99.3
ADF-A2-GJ-NPT	77	1463	3157	0.71	98.7	0.86	99.3
ADF-A1-HJ-NPT	77	1463	3157	0.71	98.7	0.84	99.1
ADF-A1-IJ-NPT	77	1463	3157	0.71	98.7	0.84	99.1
ADF-A1-GJ-NPT	77	1463	3157	0.71	98.7	0.84	99.1
ADE-A1-IK-NQT	15	285	615	0.7	98.7	0.29	81.9
ADE-A1-HK-MPT	15	285	615	0.68	98.5	0.79	98.5
ADE-A1-IK-MPT	15	285	615	0.68	98.5	0.79	98.5
ADE-A1-GK-MPT	15	285	615	0.68	98.5	0.79	98.5
ADE-A2-HK-MPT	14	266	574	0.67	98.2	0.69	96.7
ADE-A2-IK-MPT	14	266	574	0.67	98.2	0.69	96.7
ADE-A2-GK-MPT	14	266	574	0.67	98.2	0.69	96.7
ADF-L1-HK-OPT	14	266	574	0.66	97.8	0.56	94.4
Grouping signatures by application and operating system							
BCF-L1-HK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDF-L1-HK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCE-L1-HK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDE-L1-HK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCF-L1-IK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDF-L1-IK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCE-L1-IK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDE-L1-IK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCF-L1-GK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDF-L1-GK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCE-L1-GK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDE-L1-GK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCF-A1-HK-NPU	23	437	943	1.0	99.7	0.43	86.9
BCF-A1-HK-NPV	23	437	943	1.0	99.7	0.43	86.9
BCF-A1-HK-NPT	23	437	943	1.0	99.7	0.43	86.9
BDF-A1-HK-NPU	23	437	943	1.0	99.7	0.43	86.9
BDF-A1-HK-NPT	23	437	943	1.0	99.7	0.43	86.9
BCE-A1-HK-NPU	23	437	943	1.0	99.7	0.43	86.9
BCE-A1-HK-NPT	23	437	943	1.0	99.7	0.43	86.9
BDE-A1-HK-NPU	23	437	943	1.0	99.7	0.43	86.9

Table 4.7: The top twenty Signature Searchers built on n-grams (top table) vs. using whole Registry paths (bottom table). Ranking is determined by recall against the evaluation subject machine, and compared to recall in M57.

Searcher ID	# docs.	# runs		Eval. recall		M57 recall	
		Eval.	M57	Rec.	Percentile	Rec.	Percentile
Breaking Registry paths into n-grams for signature terms							
BCF-L1-HK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDF-L1-HK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCE-L1-HK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDE-L1-HK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCF-L1-IK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDF-L1-IK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCE-L1-IK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDE-L1-IK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCF-L1-GK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDF-L1-GK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCE-L1-GK-NPT	23	437	943	1.0	99.7	0.47	90.5
BDE-L1-GK-NPT	23	437	943	1.0	99.7	0.47	90.5
BCF-A1-HK-NPU	23	437	943	1.0	99.7	0.43	86.9
BCF-A1-HK-NPV	23	437	943	1.0	99.7	0.43	86.9
BCF-A1-HK-NPT	23	437	943	1.0	99.7	0.43	86.9
BDF-A1-HK-NPU	23	437	943	1.0	99.7	0.43	86.9
BDF-A1-HK-NPT	23	437	943	1.0	99.7	0.43	86.9
BCE-A1-HK-NPU	23	437	943	1.0	99.7	0.43	86.9
BCE-A1-HK-NPT	23	437	943	1.0	99.7	0.43	86.9
BDE-A1-HK-NPU	23	437	943	1.0	99.7	0.43	86.9
Using whole Registry paths as signature terms							
ADF-WP-HK-NPT	15	285	615	0.65	97.3	0.57	94.7
ADF-WP-IK-NPT	15	285	615	0.65	97.3	0.57	94.7
ADF-WP-GK-NPT	15	285	615	0.65	97.3	0.57	94.7
BDF-WP-HK-NPT	23	437	943	0.59	95.8	0.14	72.9
BDF-WP-IK-NPT	23	437	943	0.59	95.8	0.14	72.9
BDF-WP-GK-NPT	23	437	943	0.59	95.8	0.14	72.9
BDE-WP-HK-NQT	23	437	943	0.59	95.8	0.0	0.0
BDE-WP-HK-NPT	23	437	943	0.59	95.8	0.0	0.0
BDE-WP-IK-NQT	23	437	943	0.59	95.8	0.0	0.0
BDE-WP-IK-NPT	23	437	943	0.59	95.8	0.0	0.0
BDE-WP-GK-NQT	23	437	943	0.59	95.8	0.0	0.0
BDE-WP-GK-NPT	23	437	943	0.59	95.8	0.0	0.0
ACF-WP-HK-NPT	15	285	615	0.55	94.2	0.37	85.3
ACF-WP-IK-NPT	15	285	615	0.55	94.2	0.37	85.3
ACF-WP-GK-NPT	15	285	615	0.55	94.2	0.37	85.3
ADF-WP-HJ-NPT	77	1463	3157	0.53	93.9	0.61	95.0
ADF-WP-IJ-NPT	77	1463	3157	0.53	93.9	0.61	95.0
ADF-WP-GJ-NPT	77	1463	3157	0.53	93.9	0.61	95.0
BCF-WP-HK-NPT	23	437	943	0.5	91.2	0.14	72.9
BCF-WP-IK-NPT	23	437	943	0.5	91.2	0.14	72.9

Table 4.8: Counts of Searchers that had a precision of 1.0 for both the evaluation subject machine and the M57-Patents corpus, broken out by document grouping.

Docs. by	Count
app	51
osapp	0

Table 4.9: Counts of Searchers that had a precision of 1.0 for both the evaluation subject machine and the M57-Patents corpus, broken out by n-gram length.

n-grams	Count
all	15
1	0
2	8
3	4
last1	8
last2	12
last3	4

in both corpora. Table 4.4 shows more are consistent if selecting for signatures built for applications, not tailored to operating systems. If tailoring for operating systems, inconsistent performance between corpora is apparent from the precision of 0.2 for M57. Table 4.8 confirms this, showing that no operating system-tailored Searchers had consistent precision of 1.0 between both corpora, while about fifty application-grouped Searchers had consistent precision of 1.0. For precision ranking and n-grams, fifteen whole-path Searchers had consistent precision of 1.0 between both corpora, the most of any individual value of the n-gram breakout parameter.

For more illustration of consistency between the two corpora, Figure 4.15 plots the F1 of each Signature Searcher against both corpora. Many Searchers exhibited com-

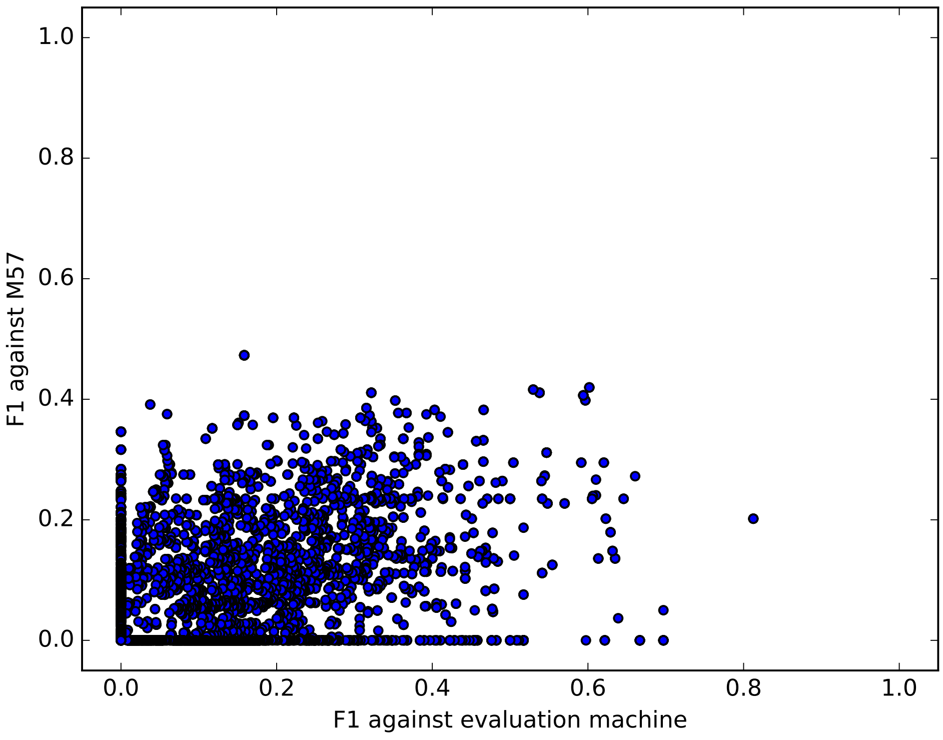


Figure 4.15: Scatter plot of F1 of each Signature Searcher measured against the evaluation subject machine states (along the x axis) and the M57-Patents corpus (along the y axis). Distance from the $y = x$ line indicates a Searcher is more overfit towards one of the corpora. Many Searchers showed they could correctly identify application presence for one of the corpora, but not at all for the other, being along the $y = 0$ or $x = 0$ lines. Some Searchers perform consistently well for both corpora, appearing in the upper-right corner.

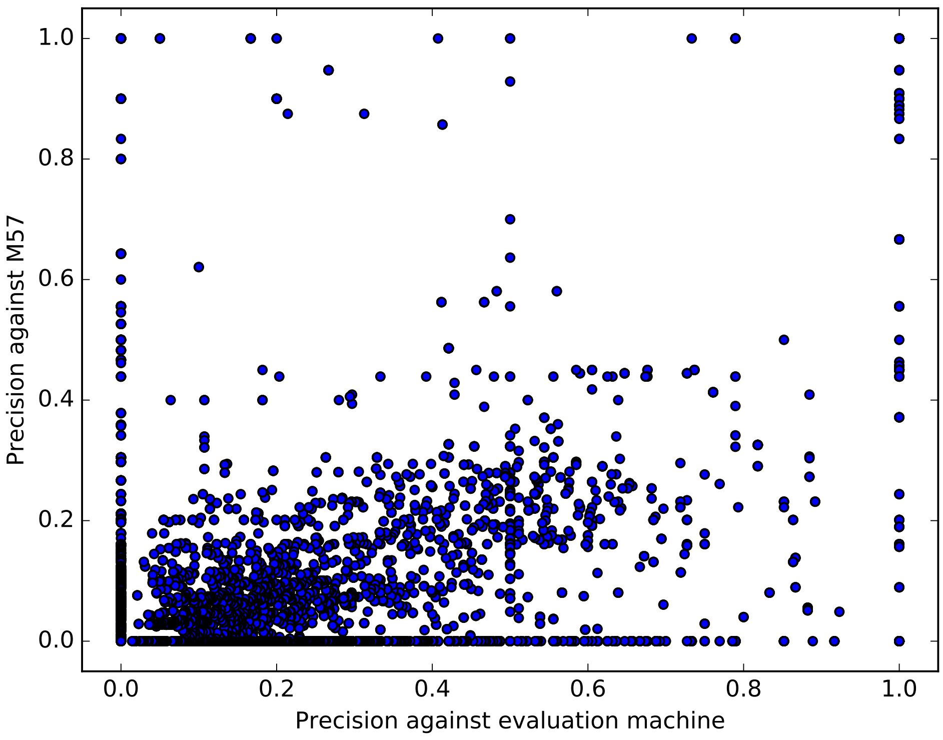


Figure 4.16: Scatter plot of precision of each Signature Searcher measured against the evaluation subject machine states (along the x axis) and the M57-Patents corpus (along the y axis). The evaluation subject machine tended to higher precision scores than the M57 corpus.

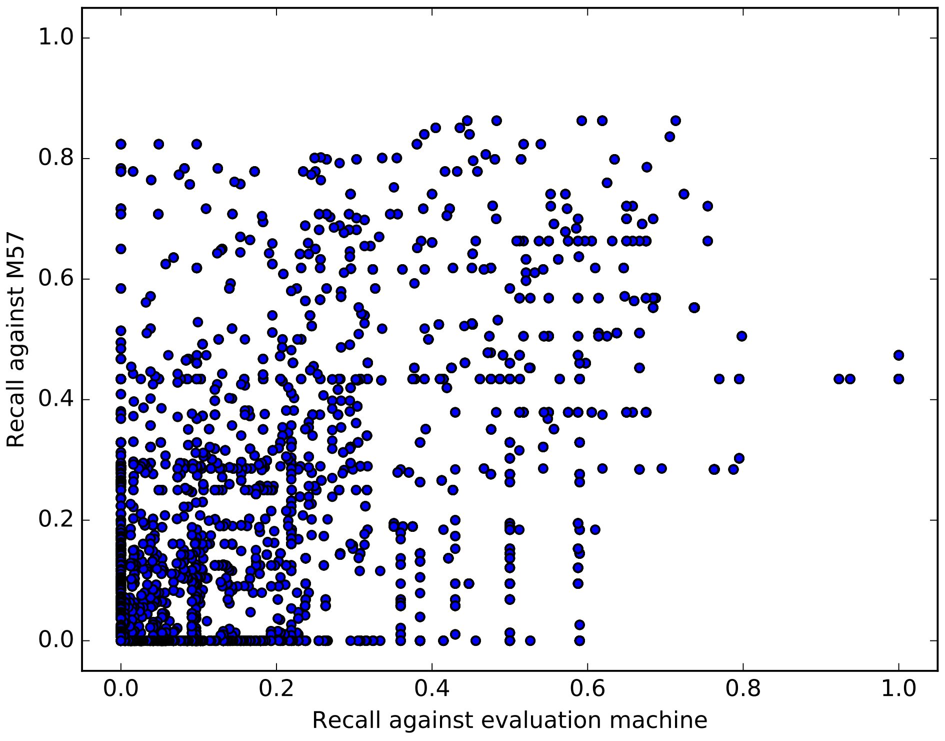


Figure 4.17: Scatter plot of recall of each Signature Searcher measured against the evaluation subject machine states (along the x axis) and the M57-Patents corpus (along the y axis). The M57 corpus tended to have higher recall scores than the evaluation subject machine, though the bias is not as pronounced as in Figure 4.16.

plete inconsistency between the two corpora, having an F1 score of 0 for one corpus but not the other. Figure 4.15 shows again that neither corpus had Searchers that attained perfect precision and recall. Figures 4.16 and 4.16 show a little more information, however. Considering the identity axis $y = x$, there are biases in the lower end of the identity axis, with precision favoring the evaluation subject machine, and recall slightly favoring the M57-Patents corpus.

Chapter 5

Conclusions, Discussion, and Suggestions for Future Research

5.1 Summary

The evaluation showed document search can create effective software signatures based on Registry artifacts, but an investigator would need to tune a Searcher's construction differently, depending on prioritizing fewer false positives or fewer false negatives. Searchers built for applications, agnostic to the trained operating system, have a higher consistent precision between the developed evaluation subject machine and the software observed in the M57 Patents corpus. Other parameter decisions are less straightforward. There are Searchers built for whole-path terms, as well as n-grams for terms, that have perfect precision for both the evaluation subject machine and the M57-Patents corpus.

If prioritizing recall, rankings by percentile suggest that, again, Searchers built for applications without tailoring to operating systems are the more consistently performant choice (see Table 4.6). The last unigram of paths seem to be more consistent as well, based on the percentiles that appear in the top twenty Searchers for n-gram breakouts. The whole-path Searchers vary to the point of including a 0-recall Searcher.

Since Signature Searchers based on Registry path components proved more consistently performant in some cases, there is a challenge to address in results presentation. Section 5.3.1 covers this, after the report of configurations of Signature Searchers that appear to be optimal according to relative average performance.

5.2 Analytic conclusions

The main effects plot (Figure 4.10) against the evaluation machine provides rankings of parameter values. Unfortunately, the study design does not allow performing statistical significance tests, so significance here is only approximated by being able to observe a slope in the line joining average-points, and by seeing if the M57 main effects plot (Figure 4.11) arrived at a different better-on-average conclusion. The averages' comparisons show the following ID patterns suggest the best Searcher configurations. First, prioritizing for precision:

Document grouping – *app* Cross-operating-system signatures had the better average precision for both corpora.

Application version grouping – *grouped* Signatures for a version of an application should be allowed to match signatures of other versions.

Path normalization Path normalization by design is consequential for signatures that combine whole paths from different operating systems. However, averaged across all the Signature Searchers, there is not a clear best choice.

n-gram length – ***not all-bigrams*** n-gram length has no consistent best average between the two corpora. All-bigrams was a consistently lower option.

n-gram stop list strategy – ***n-gram blacklist*** Blacklisting all n-grams derived from stop listed Registry paths had precision fairly consistent with the threshold option. However, the threshold option is more likely to have varying performance based on the number of baseline operating systems provided for stop lists, so the blacklist option is selected here.

Training sequence – ***repeated*** The M57 main effects doesn't favor any selection of sequences for precision. However, the evaluation subject machine performed better when restricted to repeated training sequences.

Vector combinator No combinator clearly performed best for precision.

Stop list – ***none*** Having any stop list lessened precision, for both corpora.

Threshold selector – ***min*** The score selector improved precision with the permissiveness of the threshold selector, with the minimum selector having the best average precision for both corpora.

And then, prioritizing for recall:

Document grouping The two corpora, on average, disagree on whether it is better to tailor application signatures to operating systems or to train a cross-operating-system signature.

Application version grouping – *grouped* As for precision prioritization, signatures for a version of an application should be allowed to match signatures of other versions.

Path normalization As for precision prioritization, there is not a clear best choice.

n*-gram length – *1 Whole Registry paths make for the worst average recall score. The corpora both have the highest recall for all-unigram models.

n*-gram stop list strategy – *raw filter Filtering on whole Registry paths, before splitting the path into *n*-grams, provided the best average recall score.

Training sequence – *installclose* Including all training sequences, even singleton prints, improved recall for both corpora.

Vector combinator – *summation* The summation combinator had the best average recall for both corpora.

Stop list – *none* The empty stop list had the best average recall for both corpora.

Threshold selector – *min* As for precision prioritization, the minimum-score selector provided the best recall.

Table 5.1: The top twenty Signature Searchers, according to the evaluation subject machine states, the M57-Patents corpus, and the consistently-highest precision values of the main effects inspections. Ranking is determined by average precision between the two corpora, with the average taken by the harmonic mean.

Searcher ID	# docs.	# runs		Eval. precision		M57 precision	
		Eval.	M57	Prec.	Percentile	Prec.	Percentile
According to evaluation subject machine							
ADF-WP-IJ-NSU	76	1448	3152	1.0	91.1	1.0	98.7
According to M57							
ADF-WP-IJ-NSU	76	1448	3152	1.0	91.1	1.0	98.7
According to main effects of both corpora							
ADF-L3-HK-OPT	13	247	533	0.73	89.1	0.44	96.1
ADF-L3-HK-MPT	13	247	533	0.76	89.6	0.41	95.7
ADE-L3-HK-MPT	13	247	533	0.76	89.6	0.41	95.7
ADE-L3-HK-OPT	13	247	533	0.65	87.3	0.44	96.1
ADF-WP-HK-MPT	13	247	533	0.82	90.4	0.33	94.8
ADF-WP-HK-OPT	13	247	533	0.82	90.4	0.33	94.8
ADF-L2-HK-MPT	13	247	533	0.42	76.6	0.49	97.0
ADE-L2-HK-MPT	13	247	533	0.42	76.6	0.49	97.0
ADE-WP-HK-NPT	15	285	615	0.54	84.1	0.37	95.3
ADF-WP-HK-NPT	15	285	615	0.56	84.9	0.33	95.0
ADF-L3-HK-NPT	15	285	615	0.62	86.4	0.29	93.9
ADE-L3-HK-NPT	15	285	615	0.62	86.4	0.29	93.9
ADF-L1-HK-MPT	14	266	574	0.5	80.3	0.32	94.7
ADE-L1-HK-MPT	14	266	574	0.5	80.3	0.32	94.7
ADF-L1-HK-OPT	14	266	574	0.45	78.4	0.32	94.7
ADE-L1-HK-OPT	14	266	574	0.45	78.4	0.32	94.7
ADE-A1-HK-OPT	15	285	615	0.51	82.8	0.29	93.9
ADF-L2-HK-OPT	13	247	533	0.42	76.6	0.33	94.9
ADE-L2-HK-OPT	13	247	533	0.42	76.6	0.33	94.9
ADE-A1-HK-MPT	15	285	615	0.53	83.7	0.27	93.3

Table 5.2: The top Signature Searchers, according to the evaluation subject machine states, the M57-Patents corpus, and the consistently-highest recall values of the main effects inspections. Ranking is determined by average recall between the two corpora, with the average taken by the harmonic mean. While Table 5.1 shows the top twenty, the searcher ID pattern suggested by the main effects plots for recall had a more narrow selection, reducing the parameter space to just four choices.

Searcher ID	# docs.	# runs		Eval. recall		M57 recall	
		Eval.	M57	Rec.	Percentile	Rec.	Percentile
According to evaluation subject machine							
BCF-L1-HK-NPT	23	437	943	1.0	99.7	0.47	90.5
According to M57							
ADF-A2-HL-NPT	26	494	1066	0.7	98.7	0.92	99.8
According to main effects of both corpora							
ADF-A1-GJ-NPT	77	1463	3157	0.71	98.7	0.84	99.1
BDF-A1-GJ-NPT	133	2527	5453	0.68	98.5	0.66	96.0
ADE-A1-GJ-NPT	77	1463	3157	0.48	91.1	0.86	99.3
BDE-A1-GJ-NPT	133	2527	5453	0.68	98.5	0.38	85.6

These top values suggest the best Signature Searchers for precision would have the ID pattern `AD*-{WP,A1,A3,L2,L3}-HK-*PT`, and for recall, `*D*-A1-GJ-NPT`. Table 5.1 shows the top Searchers according to the precision pattern, and Table 5.2 shows the same for recall. For comparison, they include the top Searcher according to the evaluation subject machine, and according to M57. The Searcher ID pattern derived from main effects for recall provides for fewer Searcher suggestions.

Tables 5.1 and 5.2 show that the top-performing Searchers vary in their level of agreement with the best-average Searcher parameters. Table 5.1 shows that the main effects suggestions for parameter values provide weaker-precision Searchers, supported by few substantive signatures. Yet, Table 4.8 (and others) had shown that there were more Searchers that attained perfect precision between both corpora. For recall, the best-average Searcher parameters performed well in rankings, the majority being over

96th-percentile.

For these two corpora, the main effects measurement showed better ability to find high-recall Searchers than high-precision Searchers. Further study—or in case over-fitting, evaluation data—will be needed to identify the discrepancy between Searchers with the best precision and the Searchers with the on-average best parameter choices.

5.3 Discussion

5.3.1 Reporting software presence and justifying terms

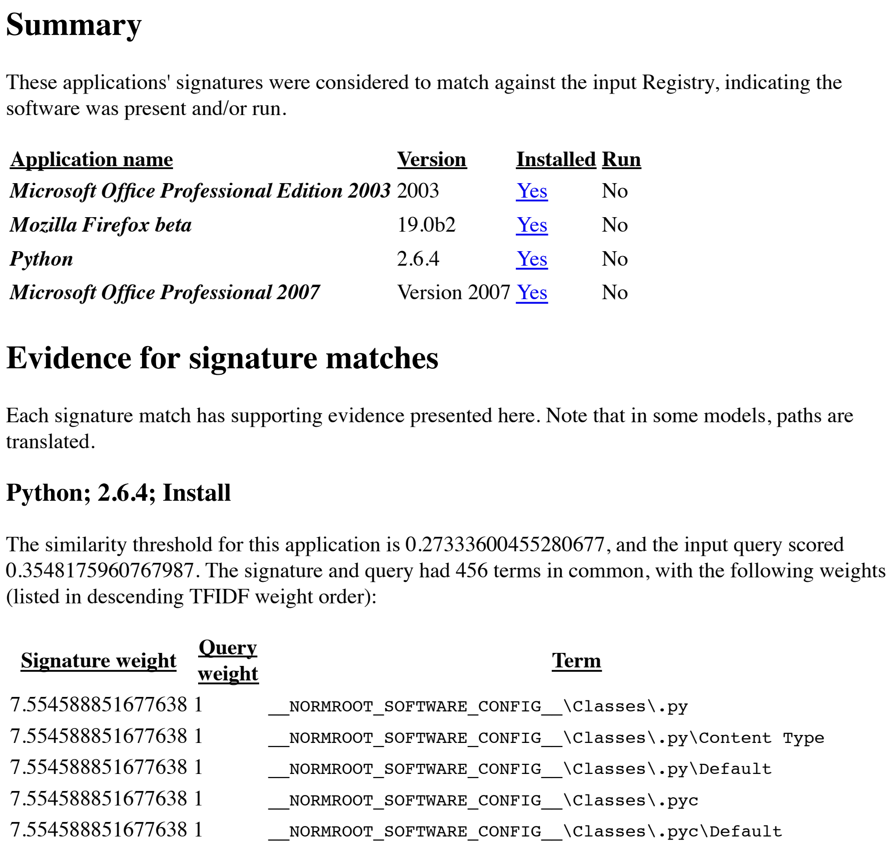
To an investigator, the important results of any of the Signature Searchers is not only the initial estimated presence guess made by the Searcher, but the supporting evidence for that guess. To that end, there is an HTML-formatted report generator that formats Signature Searcher results, providing a table of "Yes" and "No" guesses for each trained software package. Each guess links to a supporting term table, where each Registry path of the subject image is reported, along with each path's corresponding weights. This way, the Searcher emphasizes the signatory value of each path. Due to n-gram models being among the top performers, there are report design considerations to consider to make this research operational.

When a model is built that doesn't use paths as terms, but instead path component n-grams, a new challenge arises in *term* vs. *path* signatory value. Investigators may or may not find a component n-gram's signature weight to be informative, but they are likely to desire some score of the Registry path from which the term was derived.

Investigators need to be able to make reports as semantically close to the subject hard drive as possible, and a Registry path, found in a hive at some file path, suffices as a common-ground unit of explanation.

There is a straightforward solution to explain the signatory value of a Registry path, when the search model is built on whole paths as terms. The TFIDF weight of the path within each signature is readily available, and becomes the score in a hit for that application. However, when the model is built on path components instead of paths, there is need of an aggregation strategy to tabulate a path's importance. One strategy could be summing all the terms derived from a path as the path's score and presentation sorting field, boldfacing the portions of the path with a high TFIDF score for the particular signature. This would be analogous to Keyword-in-Context snippets used in natural language search results presentation [35]. Other options are available, such as simply sorting the source Registry paths, but ultimately determining the most useful presentation would require a usability study that is out of scope of this dissertation.

Figure 5.1 provides a screenshot of a prototype generated report. The report's summary section shows the Registry triage aspect of signature search, giving yes and no answers to believed software presence and usage. Each table of paths justifies, or shows the insufficient justification, for claiming an application influenced the machine. These evidence tables help emphasize the distinction between a negative response and an absent response to software presence. A Signature Searcher's "Yes" affirms software influence presence, but its "No" does not affirm software influence absence. The tables



Summary

These applications' signatures were considered to match against the input Registry, indicating the software was present and/or run.

Application name	Version	Installed	Run
Microsoft Office Professional Edition 2003	2003	Yes	No
Mozilla Firefox beta	19.0b2	Yes	No
Python	2.6.4	Yes	No
Microsoft Office Professional 2007	Version 2007	Yes	No

Evidence for signature matches

Each signature match has supporting evidence presented here. Note that in some models, paths are translated.

Python; 2.6.4; Install

The similarity threshold for this application is 0.27333600455280677, and the input query scored 0.3548175960767987. The signature and query had 456 terms in common, with the following weights (listed in descending TFIDF weight order):

Signature weight	Query weight	Term
7.554588851677638	1	`__NORMROOT_SOFTWARE_CONFIG__\Classes\.py`
7.554588851677638	1	`__NORMROOT_SOFTWARE_CONFIG__\Classes\.py\Content Type`
7.554588851677638	1	`__NORMROOT_SOFTWARE_CONFIG__\Classes\.py\Default`
7.554588851677638	1	`__NORMROOT_SOFTWARE_CONFIG__\Classes\.pyc`
7.554588851677638	1	`__NORMROOT_SOFTWARE_CONFIG__\Classes\.pyc\Default`

Figure 5.1: Screenshot of Signature Searcher report.

of evidentiary Registry paths' scores can help an investigator conclude software influence absence, particularly if the "Most relevant" paths in the subject machine appear dubious.

5.4 Suggestions for Future Research

5.4.1 Strengthening generalizability of findings

In this study, the generalizability of the Signature Searchers' best parameter values is only demonstrated by performance consistency between the two evaluation corpora. Generalizability may be better determined by analyzing more substantial corpora, but here the field of digital forensics is challenged by—with all due irony—data availability. The Real Data Corpus is the most substantial research corpus of real computer use available today, with over two thousand disk images [23], but it suffers from a total lack of ground truth in all but basic characterizing dimensions like disk image size. Creating sizeable, realistic, and annotated corpora remains a significant research challenge.

5.4.2 Search function tuning

Instead of what Zobel and Moffat refer to as the "Standard formulation" of raw counts, Croft *et al.* suggest using the "Logarithmic formulation" $r_{d,t} = 1 + \log f_{d,t}$. "Retrieval experiments have shown that to reduce the impact of these frequent terms, it is effective to use the logarithm of the number of term occurrences in *tf* weights rather than the raw count" [10, page 241]. Among the VSM parameter space, this tuning for equation 1.3, or further use of logarithms in term frequency and inverse document frequency [3], seem to be the most prudent tuning parameters to inspect. However, Zobel and Moffat offer a parameter space of 1.5 million search variants on top of the extensive construction parameter space already explored in this dissertation. Some inspection strategy will be necessary to further improve the search formulation.

5.4.3 Alternate difference basis for term sources

This thesis is scoped to analyze the set of Registry cells added from diskprint state to diskprint state. Some of the analysis in this research had been performed on an erroneous calculation of “added” cells. The error arose in some code in the Diskprint workflow that computed all of the differences for a sequence—including cell removals and content or metadata modifications—in one script invocation. (The invocation occurred just prior to the Postgres export illustrated on the right in Figure 3.2 on page 46.) Cells that were truly added between states were identified as added, but other cells were erroneously identified as added as well. The error is corrected in the analysis now, by using a simplified added-cell derivation script. However, the analysis on the erroneous difference sets made for intriguing results. Many of the Searchers built on the erroneous added-cell sets attained perfect F1 scores for either of the evaluation corpora, though none managed both simultaneously. This indicates that some characteristic of the erroneous cell sets, when identified, may make for stronger software signature recognition.

Because this thesis is scoped to added cells, the results here are only derived from added cells. Future research can incorporate cells of other differential-analytic sources. The incorrectly-computed difference data is available along with the other resources for this thesis.

5.4.4 Registry differential analysis model variants

Garfinkel *et al.* [21] described several types of differential results. In particular, feature sets can be changed by *adding* new features, *removing* old features, *modifying* features' content or properties, and *renaming* old features. This dissertation analyzed a model based on added features in depth, but for symmetry's sake with the prior differential analysis work, it is also possible to develop models based on deleted and changed content.

5.4.4.1 Subtractive model variant

Where the additive model focused on presence of cells added since a baseline system state, the subtractive model variant would rely on absence since the baseline state. Using the TFIDF VSM, a term is a cell path removed from the *baseline* image, and a change set is the collection of all those cell paths. Removal from the previous image would likely cause great confusion from well-garbage-collected temporary paths. Under this construction, the most *irrelevant* documents for any given query—*i.e.* with cosines near or at 0—may be the strongest-indicated software effects.

Unfortunately, the subtractive model rests on a strong assumption that likely will render it ineffective. For general usage, a general, minimal set of cells expected to be in all Registries would be necessary. This likely demands signatures be tailored to each operating system.

I expect constructing subtractive model variants to be a multiplicatively intensive process, as applications may need to be printed once for each Windows version and architecture of interest. I also suspect that scoring by low vector nearness will produce confusing results. Overall, I expect the additive model—built on cell presence, rather than absence—to provide more generally useful results than the subtractive variant, even when considering uninstalled software.

5.4.4.2 Mutative model variant

Differential analysis allows us to make note of metadata changes to cells, between a pre- and post-image. However, a Registry is unlikely to start from a universal constant state, where each value stores the same data. Since we are designing our search models on the assumption that all changes will have already happened by the time of investigation, and no prior value states will be visible, we must work only with what is present in the post image. Value data is unlikely to be predictable in aggregate. However, value data types, and further, cell types (key or value) may indicate Registry effects. Also, the presence or absence of data is a Boolean measure that can be taken, and has been shown to work in de-anonymization work in the past. For instance, the Netflix study collapsed user numeric scores to Boolean score presence, and still was able to de-anonymize users [39]. We can note presence or absence of value data and key timestamps.

A mutative model variant would act much like the additive model, integrating structural information into the term definition. The additive model makes use of cell paths as terms. The mutative model would use cell paths, grouped with their cell type, data type and nullness if the cell is a value, and timestamp nullness if the cell is a key.

The terms would be all paths of cells that change their types since the previous image, paired with the structural and data state.

5.4.5 Alternate data sources

The design pattern of using search to score forensic artifacts is applicable to many classes of artifact. In this document, the Registry acted as a particular file system type. File system artifacts, like file metadata or file content hashes, can fit in the same framework. Any data type that fits the general forensic differential analysis framework can fit in the search framework. For example:

- If analyzing file system metadata, file paths can be used much as Registry paths were in this document.
- If analyzing generic file contents, hash sets can be constructed for entire files, or for file sectors [19]. The NSRL Reference Data Set (RDS) of file hashes can be used to establish a TFIDF-based scoring system for hashes, but only for installation media files. Without the search score, using the RDS to identify known files on subject systems could also be used to identify some indicators of software presence, but with the overall accuracy of the "Union"-based search model, which will provide low-precision information.
- Some files are strictly associated with software execution, such as Windows prefetch files [16]. Observing the appearance of these files in a system can correlate with other features of software execution.

- RAM structures can be tied to software execution actions by differential analysis in memory. An artifact matching function is necessary so changes in structures can be established. The Differential Analysis of Malware in Memory work provides one example of this design pattern [36].

There has been preliminary analysis on sector hashes for software identification, using file system differences produced by the workflow of Section 3.3.1.3 [29].

5.5 Data and code availability

The NSRL Diskprint virtual machines are not planned to be published, because in some cases they are diskprints of proprietary software. The raw hives will not be published either, in case licensing data for those applications is included. However, file system metadata extractions and Registry cell metadata extractions of the Diskprint data are available on the NSRL website. The Diskprint workflow, Registry analysis code, generated figures, and source of this thesis are available as Git repositories. The page at this URL lists the locations of resources used to support this work: `https://users.soe.ucsc.edu/~ajnelson/research/nelson_dissertation/`.

5.6 Conclusion

This study set out to convert the voluminous metadata one encounters on a Windows system into a practical profile of computer usage. The metadata of the Windows Registry—just cell paths, without inspecting data—proved sufficient to identify software presence and usage, by assigning per-software-package scores to Registry path data, using a document search model normally used to match short natural language queries to documents. While initially the study sought to assign relevancy scores to entire Registry paths, measurements instead favored scoring path components. It also proved detrimental to exclude data from Windows systems without any additional software installed on top of the operating system—in other words, unlike in natural languages, it is worthwhile to include every term for scoring. This search methodology was able to reproduce a report of software present and used on a corpus of systems, where the previous report had used system RAM and files identified with file system differential analysis. While the previous study used differential analysis, this study made use of sequential differential analysis to organize a forensic processing framework.

Looking forward, there are challenges in automation and scale. However, this study shows that meeting those challenges will turn every entry in the voluminous Windows Registry into actionable insight of a system's usage history.

Appendix A

Supplementary Figures

Figures 4.2, 4.13 and 4.14 are scatter plots of every combination of Signature Searcher parameters, illustrating the performance of Searchers that hold different parameter values. This appendix provides the broken-out scatter plots for the other six parameters.

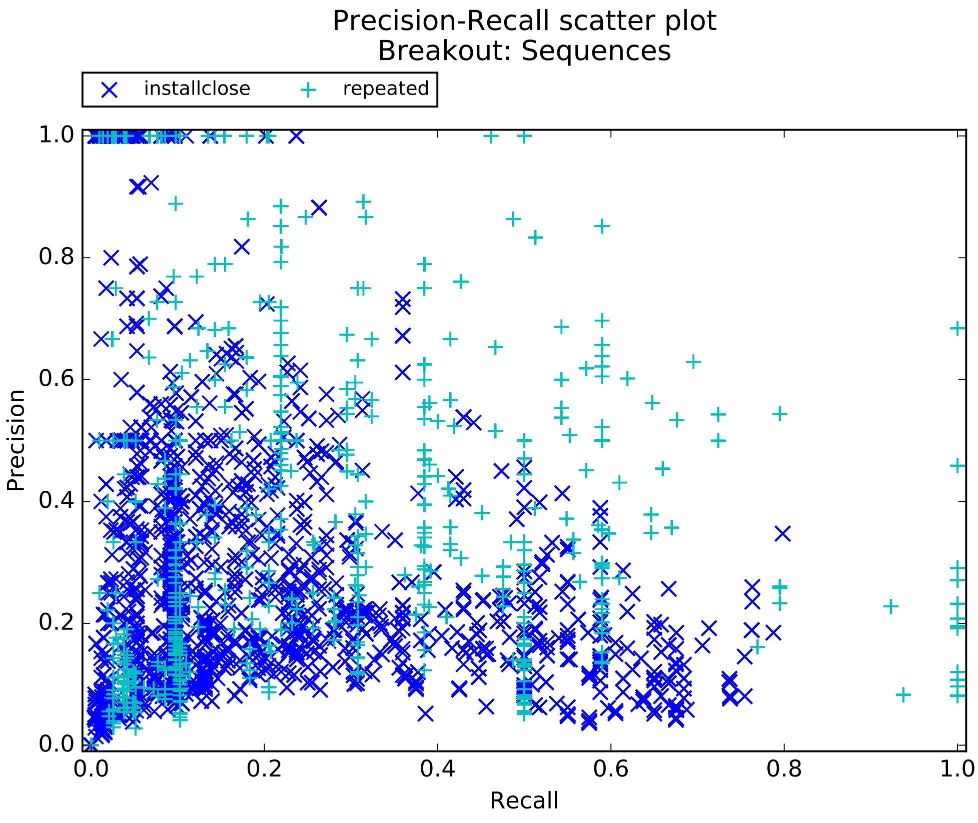


Figure A.1: Scatter plot of precision and recall of Signature Searchers measured with the evaluation subject machine states, broken out by application version grouping. Unlike Figure 4.2, this plot does not have the "experiment1" control set.

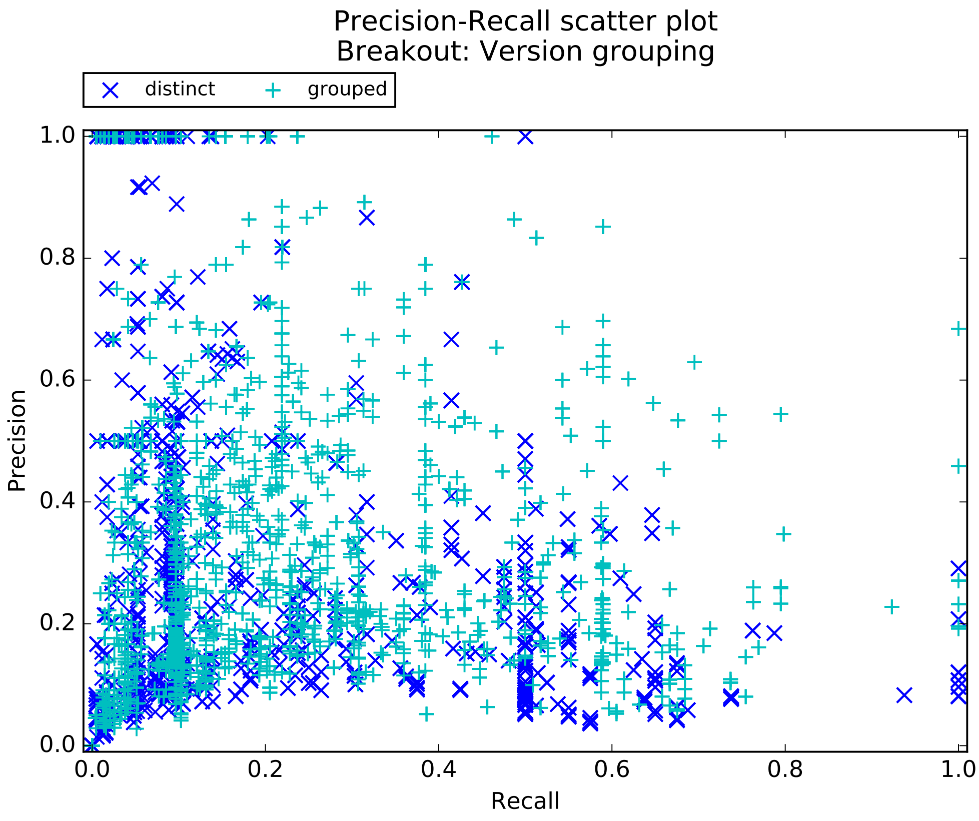


Figure A.2: Scatter plot of precision and recall of Signature Searchers measured with the evaluation subject machine states, broken out by application version grouping.

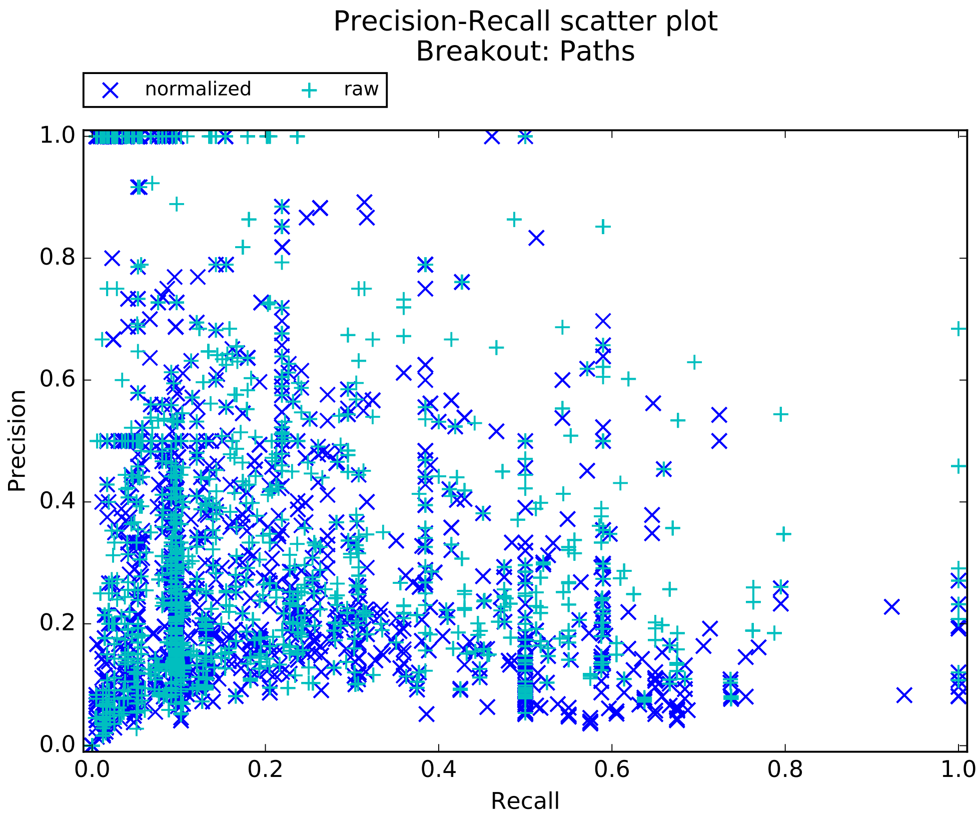


Figure A.3: Scatter plot of precision and recall of Signature Searchers measured with the evaluation subject machine states, broken out by path normalization.

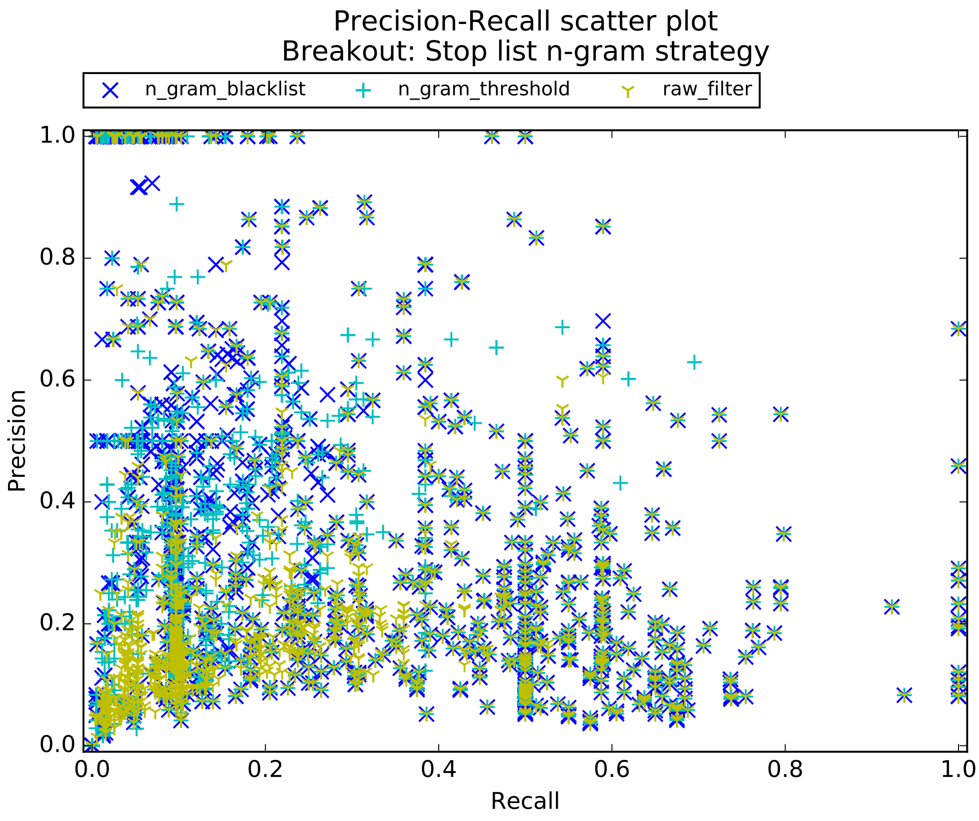


Figure A.4: Scatter plot of precision and recall of Signature Searchers measured with the evaluation subject machine states, broken out by *n*-gram stop list interaction strategy.

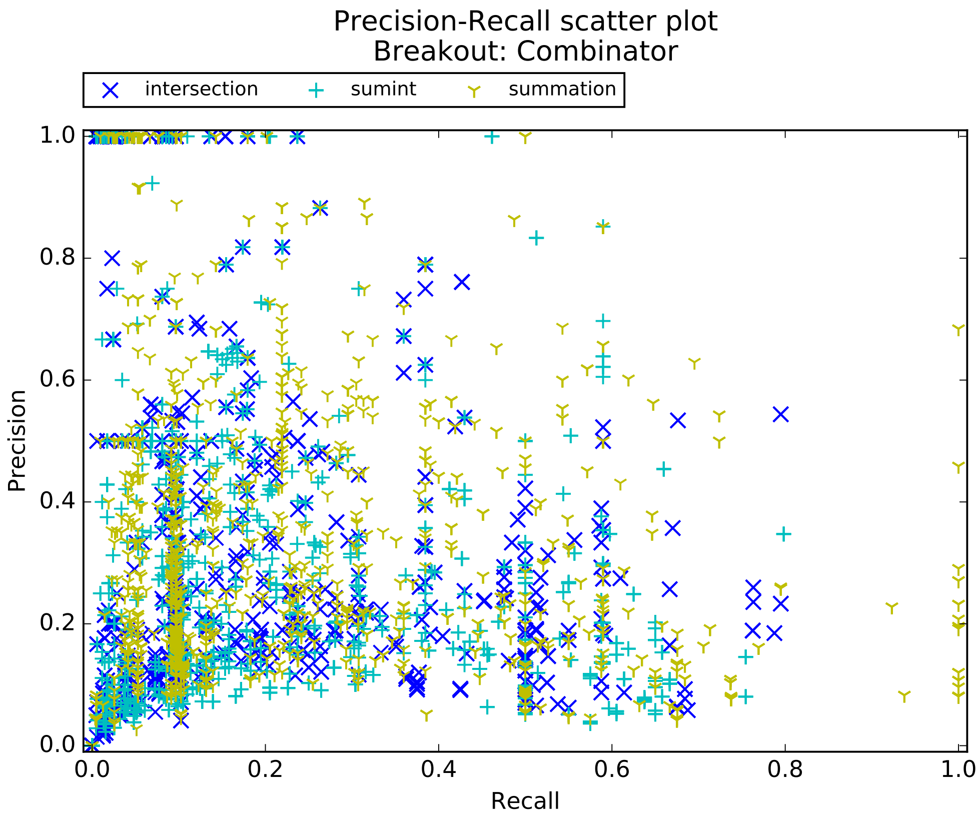


Figure A.5: Scatter plot of precision and recall of Signature Searchers measured with the evaluation subject machine states, broken out by vector combinator.

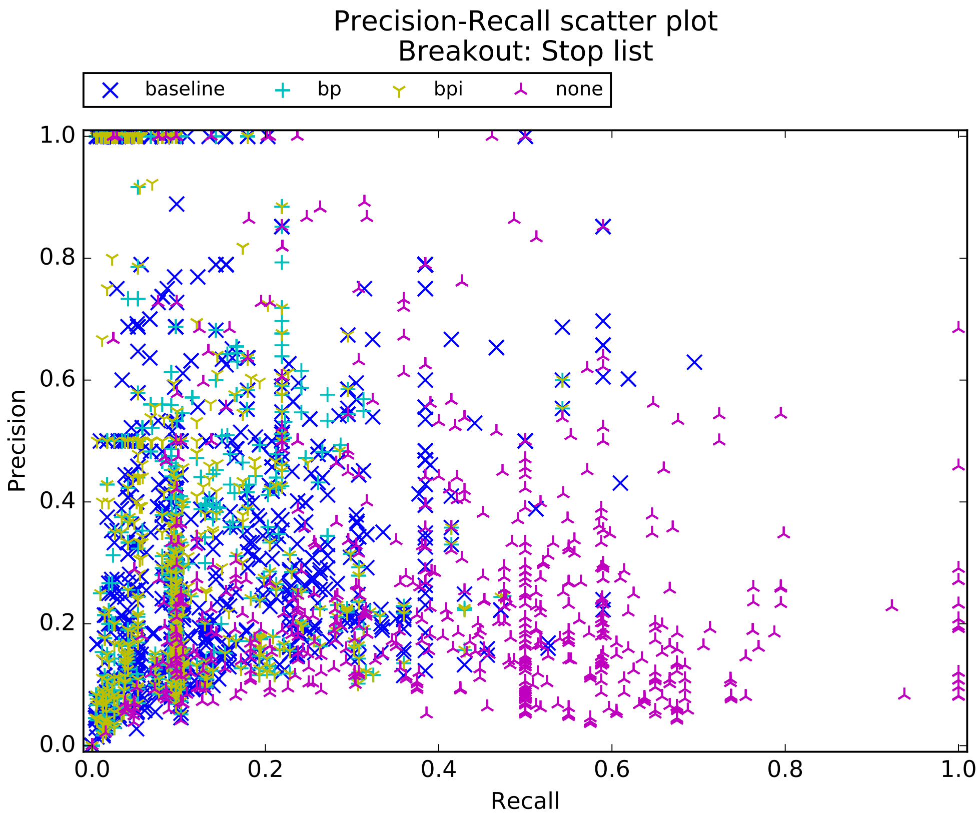


Figure A.6: Scatter plot of precision and recall of Signature Searchers measured with the evaluation subject machine states, broken out by stop list.

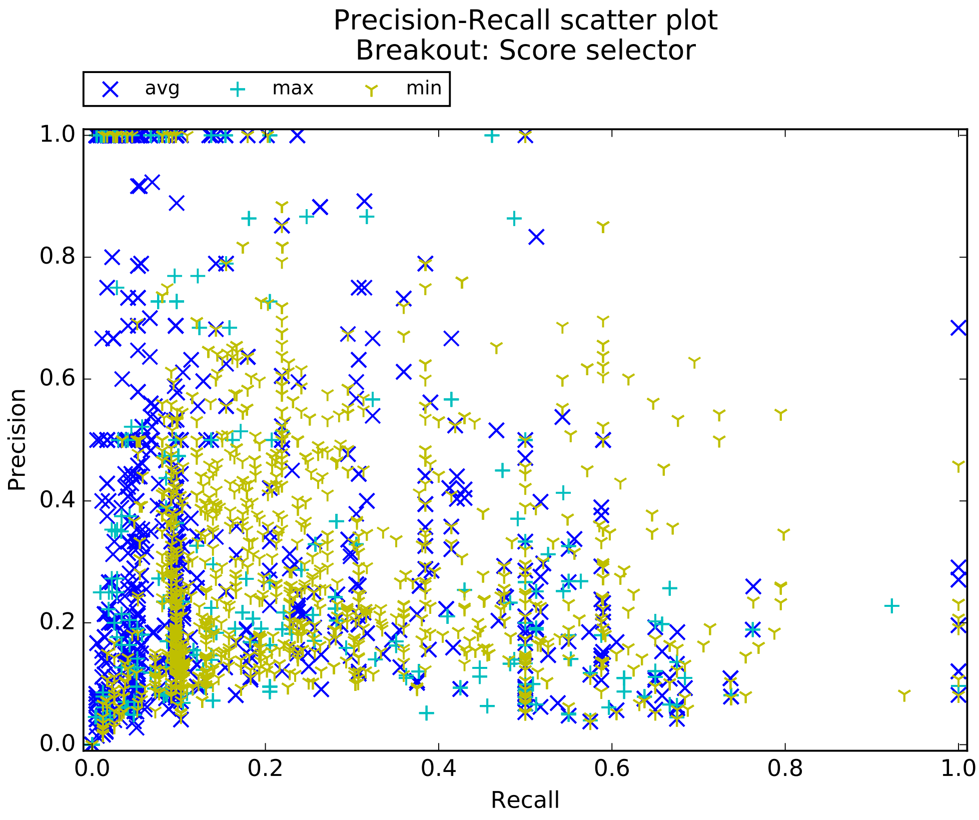


Figure A.7: Scatter plot of precision and recall of Signature Searchers measured with the evaluation subject machine states, broken out by threshold selector.

Bibliography

[1] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST)*, pages 31–45, February 2007.

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003.

[3] Nicole Lang Beebe and Lishu Liu. Ranking algorithms for digital forensic string search hits. In *Proceedings of the 14th Annual Digital Forensic Research Workshop (DFRWS)*, 2014.

[4] Steven Bird. NLTK: The Natural Language Toolkit. In *Proceedings of the COLING/ACL 2006 Interactive Presentation Sessions*, COLING-ACL '06, pages 69–72, Sydney, Australia, 2006. Association for Computational Linguistics.

[5] George E. P. Box, William Gordon Hunter, and J. Stuart Hunter. *Statistics for Experimenters.* Wiley & Suns, Inc., 1978.

[6] Brian Carrier. *File System Forensic Analysis.* Addison-Wesley, 2005.

[7] Harlan Carvey. *Windows forensic analysis.* Elsevier, 2e edition, 2009.

[8] Harlan Carvey. *Windows Registry forensics: advanced digital forensic analysis of the Windows Registry.* Elsevier, 2011.

[9] Pu-Jen Cheng, Jei-Wen Teng, Ruei-Cheng Chen, Jenq-Haur Wang, Wen-Hsiang Lu, and Lee-Feng Chien. Translating unknown queries with web corpora for cross-language information retrieval. In *Proceedings of the 27th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR '04, pages 146–153, New York, NY, USA, 2004. ACM.

[10] W. Bruce Croft, Donald Metzler, and Trevor Strohman. *Search Engines: Information Retrieval in Practice.* Addison-Wesley, 2010.

[11] Jesse Davis and Mark Goadrich. The relationship between Precision-Recall and ROC curves. In *Proceedings of the 23rd International Conference on Machine Learning*, 2006.

[12] Mark Davis, Richard Kennedy, Kristina Pyles, Amanda Strickler, and Sujeet Shenoi. Detecting data concealment programs using passive file system analysis. In Martin S. Olivier and Sujeet Shenoi, editors, *Advances in Digital Forensics II*, volume 222 of *IFIP Advances in Information and Communication*, pages 171–183. Springer New York, 2006.

[13] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the 1999 SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 59–70, May 1999.

[14] Arthur Conan Doyle. *Sherlock Holmes: The Complete Novels and Stories*, volume I. Bantam Classic, New York, New York, September 2003.

[15] Paul Farrell, Simson L. Garfinkel, and Douglas White. Practical applications of Bloom filters to the NIST RDS and hard drive triage. In *2008 Annual Computer Security Applications Conference (ACSAC)*, pages 13–22, December 2008.

[16] Luis E. Garcia II. Bulk Extractor Windows Prefetch decoder. Technical Report NPS-CS-11-008, Naval Postgraduate School, August 2011.

[17] Simson Garfinkel. Digital Forensics XML and the DFXML toolset. *Digital Investigation*, 8(3–4):161–174, February 2012.

[18] Simson Garfinkel. Digital media triage with bulk data analysis and bulk_extractor. *Computers & Security*, 32:57–72, February 2013.

[19] Simson Garfinkel and Michael McCarrin. Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. In *Proceedings of the 15th Annual Digital Forensic Research Workshop (DFRWS)*, August 2015.

[20] Simson Garfinkel, Alex Nelson, Douglas White, and Vassil Roussev. Using purpose-built functions and block hashes to enable small block and sub-file forensics. In

Proceedings of the 10th Annual Digital Forensic Research Workshop (DFRWS), August 2010.

[21] Simson Garfinkel, Alex J. Nelson, and Joel Young. A general strategy for differential forensic analysis. In *Proceedings of the 12th Annual Digital Forensic Research Workshop (DFRWS)*, August 2012.

[22] Simson L. Garfinkel. Automating disk forensic processing with SleuthKit, XML and Python. In *Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE '09)*, pages 73–84, 2009.

[23] Simson L. Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. In *Proceedings of the 9th Annual Digital Forensic Research Workshop (DFRWS)*, Quebec, Canada, August 2009.

[24] Matthew Geiger. Evaluating commercial counter-forensic tools. In *Proceedings of the 5th Annual Digital Forensic Research Workshop (DFRWS)*, 2005.

[25] Phil Harvey. Exiftool. `http://www.sno.phy.queensu.ca/~phil/exiftool/`, 2015. Retrieved December 2, 2015.

[26] Scott Hillier. The system Registry. `https://msdn.microsoft.com/en-us/library/ms970651.aspx`, 1996. Retrieved January 28, 2016.

[27] Hongyi Hu and Chad Spensky. Live disk forensics on bare metal. Presentation at 5th Annual Open Source Digital Forensics Conference (OSDFCon 2014), 2014.

[28] Brian Jones, Syd Pleno, and Michael Wilkinson. The use of random sampling in investigations involving child abuse material. In *Proceedings of the 12th Annual Digital Forensic Research Workshop (DFRWS)*, August 2012.

[29] Jim Jones, Tahir Khan, Kathryn Laskey, Alex Nelson, Mary Laamanen, and Douglas White. Inferring previously uninstalled applications from digital traces. In *Proceedings of the Conference on Digital Forensics, Security and Law 2016*, pages 113–130, 2016.

[30] Richard W. M. Jones. hivex - Windows Registry "hive" extraction library. `http://libguestfs.org/hivex.3.html`. Retrieved February 23, 2016.

[31] Richard W. M. Jones. Windows SAM and hivex. `https://rwmj.wordpress.com/2010/06/09/windows-sam-and-hivex/`. Retrieved February 26, 2016.

[32] Sven Kälber, Andreas Dewald, and Felix C. Freiling. Forensic application-fingerprinting based on file system metadata. In *Proceedings of the 2013 Seventh International Conference on IT Security Incident Management and IT Forensics*, pages 98–112, 2013.

[33] Gene H. Kim and Eugene H. Spafford. The design and implementation of Tripwire: A file system integrity checker. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 18–29, 1994.

[34] Kibom Kim, Sangseo Park, Taejoo Chang, Cheolwon Lee, and Sungjai Baek. Lessons learned from the construction of a Korean software reference data set

for digital forensics. *Digital Investigation*, 6, Supplement:S108 – S113, 2009. The Proceedings of the Ninth Annual DFRWS Conference.

[35] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing.* MIT Press, 1999.

[36] Vico Marziale. DAMM: Differential analysis of malware in memory. Presentation at 5th Annual Open Source Digital Forensics Conference (OSDFCon 2014), 2014.

[37] Robert Mecklenburg. *Managing Projects with GNU Make.* O'Reilly Media, Inc., 2004.

[38] Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, February 2011.

[39] Arvind Narayanan and Vitaly Shmatikov. Robust de-anonymization of large sparse datasets. In *2008 IEEE Symposium on Security and Privacy (SP '08)*, pages 111–125, May 2008.

[40] Alex Nelson. RegXML: XML conversion of the Windows Registry for forensic processing and distribution. In Gilbert Peterson and Sujeet Shenoi, editors, *Advances in Digital Forensics VIII*, IFIP Advances in Information and Communication Technology, pages 51–65. Springer Berlin Heidelberg, 2012.

[41] Alex J. Nelson. RegXML Extractor. `https://github.com/ajnelson/regxml_extractor`, 2012. Retrieved February 27, 2016.

[42] Alex J. Nelson. Diskprint database. `https://github.com/ajnelson/diskprint_database`, 2013. Retrieved February 27, 2016.

[43] Alex J. Nelson, Mary T. Laamanen, John Tebbutt, and Darrell D. E. Long. Indexing the Windows Registry for software detection. In *Proceedings of the American Academy of Forensic Sciences 66th Annual Scientific Meeting*, page 156, Seattle, WA, 2014. American Academy of Forensic Sciences.

[44] Peter Norris. The internal structure of the Windows Registry. Master's thesis, Cranfield University, 2009.

[45] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program annual report fiscal year 2011.

[46] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program fiscal year 2004 annual report, 2004.

[47] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program fiscal year 2005 annual report, 2005.

[48] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program annual report for fiscal year 2005, May 2006.

[49] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program annual report for fiscal year 2006, 2007.

[50] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program annual report for fiscal year 2007, 2008.

[51] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program annual report for fiscal year 2008, 2009.

[52] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program annual report for fiscal year 2009, May 2010.

[53] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program annual report for fiscal year 2010, May 2011.

[54] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program annual report fiscal year 2012, 2013.

[55] Federal Bureau of Investigation. Regional Computer Forensics Laboratory Program annual report fiscal year 2012, 2014.

[56] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[57] Python Software Foundation. 20.5. `xml.etree.ElementTree` – the ElementTree XML API. https://docs.python.org/3.4/library/xml.etree.elementtree.html. Retrieved Feb. 21, 2016.

[58] Vassil Roussev. Data fingerprinting with similarity digests. In Kam-Pui Chow and Sujeet Shenoi, editors, *Advances in Digital Forensics VI*, volume 337 of *IFIP Advances in Information and Communication Technology*, pages 207–226. Springer Berlin Heidelberg, 2010.

[59] Vassil Roussev. Managing terabyte-scale investigations with similarity digests. In Gilbert Peterson and Sujeet Shenoi, editors, *Advances in Digital Forensics VIII*,

volume 383 of *IFIP Advances in Information and Communication Technology*, pages 19–34. Springer Berlin Heidelberg, 2012.

[60] Vassil Roussev and Candice Quates. Content triage with similarity digests: The M57 case study. In *Proceedings of the 12th Annual Digital Forensic Research Workshop (DFRWS)*, August 2012.

[61] Neil C. Rowe. Testing the National Software Reference Library. In *Proceedings of the 12th Annual Digital Forensic Research Workshop (DFRWS)*, August 2012.

[62] Robert C. Russell. United States patent 198,458, 1917. Issued 1918.

[63] Margo Seltzer and Nicholas Murphy. Hierarchical file systems are dead. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS-XII)*, 2009.

[64] Ole Tange. GNU Parallel - the command-line power tool. *;login: The USENIX Magazine*, 36(1):42–47, 2011.

[65] John Tebbutt, Mary T. Laamanen, and Alex J. Nelson. Tracking the effects of software on systems: A forensic metadata collection. In *Proceedings of the American Academy of Forensic Sciences 66th Annual Scientific Meeting*, page 155, Seattle, WA, 2014. American Academy of Forensic Sciences.

[66] Jolanta Thomassen. Forensic analysis of unallocated space in Windows Registry Hive files. M.sc. thesis, University of Liverpool, April 2008.

[67] Emma Tonkin. Searching the long tail: Hidden structure in social tagging. *Advances in Classification Research Online*, 17(1):1–10, 2006.

[68] Daniel Veillard. The XML C parser and toolkit of Gnome. `http://www.xmlsoft.org/`. Retrieved Feb. 21, 2016.

[69] David Waltermire, Brant A. Cheikes, Larry Feldman, and Greg Witte. Guidelines for the creation of interoperable software identification (SWID) tags. Technical Report NISTIR 8060, National Institute of Standards and Technology, 2016.

[70] Eric W. Weisstein. Symmetric difference. `http://mathworld.wolfram.com/SymmetricDifference.html`. From MathWorld – A Wolfram Web Resource.

[71] Douglas R. White. The National Software Reference Library: Applications in digital forensics. In *Proceedings of the American Academy of Forensic Sciences 66th Annual Scientific Meeting*, pages 155–156, 2014. Presentation with abstract in proceedings.

[72] Kam Woods, Christopher Lee, Simson Garfinkel, David Dittrich, Adam Russel, and Kris Kearton. Creating realistic corpora for forensic and security education. In *2011 ADFSL Conference on Digital Forensics, Security and Law*, Richmond, VA, 2011. Elsevier.

[73] Yuandong Zhu, Pavel Gladyshev, and Joshua James. Using shellbag information to reconstruct user activities. In *Proceedings of the 9th Annual Digital Forensic Research Workshop (DFRWS)*, 2009.

[74] Justin Zobel and Alistair Moffat. Exploring the similarity space. *ACM SIGIR Forum*, 32(1):18–34, 1998.