IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received February 22, 2021, accepted April 4, 2021, date of publication April 8, 2021, date of current version April 16, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3071795

Developing Software Signature Search Engines Using Paragraph Vector Model: A Triage Approach for Digital Forensics

SOMAYEH SOLTANI[1], SEYED AMIN HOSSEINI SENO[1], AND RAHMAT BUDIARTO[2]
[1]Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad 9177948974, Iran
[2]Department of Informatics, Faculty of Science and Technology, Universitas Al-Azhar Indonesia, Jakarta 12110, Indonesia

Corresponding author: Seyed Amin Hosseini Seno (hosseini@um.ac.ir)

ABSTRACT Today, with the growth of information and communication technology, digital crimes have also spread. Advanced storage technologies and their low cost have led to a significant increase in their use. Therefore, the high volume of digital data to be analyzed is a challenge facing digital forensic investigators. Digital forensic triage solutions aim to alleviate the forensic backlog. A promising triage technique is to quickly find the software packages run on the target system to narrow down the search space. In this paper, we propose a software signature search engine (S3E) to identify software on the system under investigation. The document collection of this search engine consists of software signatures, and the query is the features extracted from the system's hard disk. We propose a forensic differential analysis model to build software signatures. Besides, we use the paragraph vector model to construct the corresponding vectors of each software signature and find similarities between the query vector and the signature vectors. Different design parameters are involved in making software signature search engines, and distinct values of these parameters lead to different models. We have measured the performance of these S3E models against several controlled systems and some pseudo-real systems. The experimental results on both datasets show that some S3E models achieve perfect recall, and many of them have a recall of more than 90%. Besides, we find that the recall rate of the S3E models in both datasets is higher than the averaged word2vec model and the TF-IDF model.

INDEX TERMS Digital forensics, triage solution, software signature, forensic differential analysis, search engine, paragraph vector.

I. INTRODUCTION

Advances in technology have led to the creation of high-volume, low-cost digital media. Therefore, in today's digital cases, a large amount of data is present. The Regional Computer Forensic Laboratory (RCFL) annual reports reveal a significant increase in the number of digital cases and the volume of data [1]. Consequently, digital forensic laboratories experience the accumulation of evidence awaiting analysis [2], [3].

In addition to the large volume of data, the lack of automated forensic analysis methods slows down the analysis process. Conventional digital forensic tools such as Encase and FTK list the artifacts extracted from digital devices. They provide keyword searching, index searching, and filtering the known files. However, these tools cannot investigate what events led to the creation of these artifacts, and this is mostly done manually by the investigator [4]–[7]. This manual process causes some problems. First, the range of the reconstructed events is entirely limited to the experience of the investigator. The examiner cannot retrieve the unknown events, and thus the retrieval rate would reduce [8]. Second, like most manual analysis, there is a possibility of human error and reduced accuracy. Third, the process of manually reconstructing the events is very time-consuming.

Over the years, researchers have presented numerous triage and data reduction solutions. Detection of the programs executed on the system is a digital forensic triage approach, which narrows the investigation scope and gives the investigator an overview of the system [9], [10]. A list of software

The associate editor coordinating the review of this manuscript and approving it for publication was Xianzhi Wang.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

executed on the target system can help the investigator make hypotheses about the incident [11]–[13].

Running any software causes changes in various parts of the system, including the hard disk and, in particular, the file system. We can use the footprints of each software to build the software signature. Then, in a post-mortem analysis, the investigator could use the software signatures to determine what software was present on the compromised system. So far, different methods have built signatures for some software or events [8]–[10], [13]–[19]. However, the methods in [14], [15], and [17] made signatures with a limited number of items, and therefore, the adversary can easily bypass the signature detection. The methods in [8], [15], and [19] did not provide an automated process for making signatures. Finally, [9], [10] used the content of disk copy to make software signatures. Detecting these signatures, unlike the metadata-based ones, is time-consuming.

In this study, we present a forensic differential analysis model. This differential model can be used to calculate the differences between two digital objects of the same type. In particular, it can be used to calculate the differences between two copies of a disk. We make the software signature using the differential analysis of disk copies before and after running the software.

Besides, we develop a software signature search engine (S3E) to detect the software on a target system. The input documents of this search engine are the signatures of various software. The query of the search engine is the disk copy of the target system. We then use the paragraph vector model [20] to build the numeric representation of signatures and queries. Finally, we measure the similarity of the query with software signatures. If the query's similarity with a software signature is more than an established threshold, we conclude that the software has run on the target system.

We consider several design parameters for building software signature search engines. Different values of these design parameters lead to different S3E models. Then, we measure the performance of these models against some experimental datasets and a pseudo-real dataset.

We can summarize our contributions as follows:

- We present a general model of forensic differential analysis that can be used to detect differences between two digital objects for forensic purposes. We then adapt this general model to disk objects to build software signatures.
- We provide 120 software signature search engine models with different design parameters. Then, we run these search engines against several controlled machines and M57 Patents machines [21] and obtain their precision and recall. We also calculate the effect of different values of design parameters on the performance of the S3E models and determine the values that result in the best models.

The organization of this paper is as follows. Section II reviews the related research on digital forensic analysis. Section III describes the word embedding and paragraph vector model. Section IV describes the proposed method for building software signature and software signature search engine. The experimental results are given in Section V, and finally, Section VI concludes the paper.

II. LITERATURE REVIEW

Digital forensic investigation is the analysis of digital evidence using scientific methods to reconstruct the happened events. Digital forensics has a history of almost three decades. While the first attempts in digital forensics were devoted to gathering valuable information from the compromised system [22]–[25], recent research focuses more on analyzing extracted evidence. After extracting evidence from the compromised system, the investigator faces a large amount of low-level raw information that should be analyzed to identify and reconstruct the events. In this section, we will review research works in digital forensic analysis and event reconstruction.

A. INDEX SEARCHING AND FILTERING

One of the first attempts in forensic analysis is to search the keyword in the collected data. Digital forensic analysis tools such as ProDiscover, EnCase, FTK, and PyFlag also provide index searching capability [26]–[28]. After mounting the disk copy, an index is created on text data and metadata of the files. While index generation is very time-consuming, index searching is fast.

Since so many files exist in a digital case, the investigator sometimes decides to filter out files related to known software to reduce the investigation time. Given a hash list of known files, popular forensic tools such as EnCase and FTK can filter known files. NIST's National Software Reference Library (NSRL) is one of the most important references of the known files [29]. It includes an extensive collection of software packages and a metadata database of the files in the software packages. A subset of the metadata for each file is published as NSRL Reference Data Set (RDS). Since 1999, the RDS dataset has been published and updated quarterly. For each file in the NSRL collection, the RDS includes 1) cryptographic hash values of the file's content, 2) information about the software package(s) containing the file, 3) the manufacturer of the package, 4) the original name, and 5) the size of the file. Many studies have used the hash list of RDS to identify and filter known benign files [3], [11], [30], and [31].

B. DIGITAL FORENSIC TIMELINE

One of the first steps taken to facilitate forensic analysis is to create a timeline of digital events. Timing information obtained from digital evidence such as file systems and log files is displayed in chronological order to get a better view of the temporal order of creating, accessing, or manipulating various digital evidence.

One of the first attempts to display the timeline of events is Zeitline [32]. The timeline includes MACB timestamps and timing information from system logs and IDS and firewall logs. Olsson and Boldt [33] developed a tool called CFTL that

extracts timestamps from FAT and NTFS file systems. It also extracts timing information from different files such as EXIF files, Link files, MBOX archives, and Registry hives. CFTL designs a special extractor for each type of file.

Log2timeline [34] is another efficient tool for extracting timing information from different file types. Log2timeline designs special extractors for 26 different input files, including Chrome, Internet Explorer and Firefox history files, event log files, McAfee log files, and Registry. Subsequent versions of log2timeline have parsers for about 100 different sources [35].

Timeline2GUI [36] is a graphical interface that reads and analyzes CSV files generated by log2timeline. Operations supported by this tool include 1) filtering data by column values, 2) sorting column values, 3) searching text anywhere in CSV data, and 4) configuring automatic highlighting. These tools do not allow for automated analysis, and the investigator needs to aggregate and correlate the evidence to create high-level events.

C. CREATING SIGNATURES FOR SOFTWARE OR ACTIVITIES

Some research efforts create signatures for applications or activities. The signatures can be used later in a post-mortem analysis to identify what events took place on the target system. PyDFT [15] is a digital forensic timeline including file system timestamps and timing information of files such as Skype, Chrome history, Registry, and Link files. It also defines some rules to detect several activities such as USB connection, Skype call, and Google search. However, PyDFT does not provide an automated process for making signatures. Kälber *et al.* [17] used file system timestamps and James *et al.* [14] used file system and registry timestamps to make signatures. These two methods make signatures with a limited number of timestamps.

Roussev and Quates [9] identified the software in M57 Patents machines [21] using fuzzy hashing (similarity digest). Unlike cryptographic digest, which checks object identity, the similarity digest finds similar objects. Roussev and Quates used sdhash [37] to generate the similarity digests for all executable files. Then, to discover software applications on the system, they used these hashes as a query against the memory image. This method is content-based and relatively slow. It also requires both the disk copy and memory image of the system to detect software.

Jones *et al.* [10] built signatures or catalogs for various software using the NIST's Diskprint project [38]. In this project, each software is installed, run, closed, and uninstalled separately on a base-controlled operating system on a virtual machine. In each of these states, a snapshot is taken from the system, and the software diskprint is created.

For each software, Jones *et al.* compared consecutive snapshots of the software diskprint and identified created and modified files. Then they created a software catalog using the names of these files and the MD5 hashes of the sectors containing these files. The hard disk of the target system is then matched to the software catalogs, and two file-weighted and sector-weighted measures are used to determine whether the software is present on the disk in question. This method calculates the hash of disk sectors, which is relatively time-consuming. Also, it does not address the issue of setting a threshold for software presence.

Khader *et al.* [19] looked for fingerprints of Hadoop Distributed File System (HDFS) operations. They recorded metadata changes after each operation and used fsimage and hdfs-audit logs to view metadata for HDFS operations. This method only attempts to find footprints for basic file operations such as create, delete, append, and rename. It does not build signatures for high-level events.

Jeong and Lee [39] tried to recognize storage devices, including HDD, SSD, and USB. They extracted the connection signature of storage devices connected to a computer system from different parts of the system, including the system registry, the master boot record (MBR), the system logs, and NVAR variables.

Park *et al.* [40] created signatures for anti-forensic tools or techniques. They created the signatures using a seven-step process: 1) creating the virtual machine test system, 2) running Process Monitor as a file system logger, 3) performing desired actions, i.e., installing, running, and uninstalling anti-forensic tool, 4) saving the output of file system logger, 5) filtering the output to remove the noises, 6) extracting unique signature, and 7) using regular expressions to generalize variables in signatures.

The work in [18] developed software signatures using file system metadata. The authors suggested a similarity measure to calculate the similarity of software signatures and the target system. Nevertheless, the authors did not consider different design parameters to build software signatures and did not use software signature search engines.

The work by Nelson [13] is the most similar work to ours, which has designed software signature searchers. The author used a frequency-based information retrieval method. Frequency-based methods consider independent vectors for words and do not take into account the syntactic and semantic similarity of the words. In this study, we use the paragraph vector model, so that the vectors of words with similar contexts are closer to each other in the vector space. Another significant difference between the two methods is that Nelson used registry artifacts, while we use file system artifacts.

The M57 Patents scenario [21], developed by the Naval Postgraduate School, tracks the first four weeks of the M57 Patents company. The company has four employees, Charlie, Jo, Pat, Terry. They perform some malicious and illegal acts, in addition to their usual activities. The purpose of designing this scenario is for students to find traces of various malicious activities performed by the company's employees. This scenario involves four computers. At the end of each working day, the hard disk and memory of each computer are imaged. Also, the company's network traffic is recorded every day. Besides, the contents of four USB devices have been copied. M57 Patents is one of the most important and documented forensic research collections. In this

paper, we use some of the computers of this scenario as the pseudo-real dataset.

III. PRELIMINARIES: PARAGRAPH VECTOR MODEL

To design our software signature search engine, we use the paragraph vector model. In traditional information retrieval techniques, each term/word is displayed as a one-hot encoded vector [41]. Each document or query is displayed as a vector in N-dimensional vector space, in which N is the number of terms/words in the corpus of documents. There are several ways to calculate the weight of a term in a document, and one of the most common is the term frequency-inverse document frequency (TF-IDF) [42].

In these frequency-based information retrieval methods, the vector of each word is independent of the other words, i.e., the similarity of one-hot encoded vectors is zero. We need representations for word vectors that consider the context and semantic of words, so that word vectors with similar contexts are closer together in the vector space. Mathematically speaking, the cosine of the angle between word vectors with similar context should be close to one.

Recent NLP models learn meaningful vector representations for words. These vectors are often known as word embeddings or distributed word representations [43]. There are neural network solutions for learning word embedding, which preserve the syntactic and semantic relations between words. Two of the most popular neural network models for training word embedding are continuous bag-of-words (CBOW) and skip-gram [44]. These two models are known as word2vec.

The Skip-gram model tries to predict the context words given a target word. Therefore, given a sequence of training words $w_1, w_2, \cdots, w_N$, the goal of the Skip-gram model is to maximize the log probability in (1).

$$\frac{1}{N}\sum_{t=1}^{N}\sum_{-c\le i\le c, i\neq 0} log\left(p\left(w_{t+i}|w_t\right)\right), \tag{1}$$

where N is the number of training words, and c is the size of the sliding window that determines the number of context words.

Inversely, the continuous bag-of-words model tries to predict the target word given the context words. More formally, given a sequence of training words $w_1, w_2, \cdots, w_N$, the goal of the CBOW model is to maximize the average log probability in (2).

$$\frac{1}{N}\sum_{t=1}^{N}\sum_{-c\le i\le c, i\neq 0} log\left(p\left(w_t|w_{t+i}\right)\right). \tag{2}$$

The word2vec models formulate the probability $p\left(w_O|w_I\right)$ using the softmax function represented by (3).

$$p\left(w_O|w_I\right) = \frac{exp\left(v'^{\mathrm{T}}_{w_O} v_{w_I}\right)}{\sum_{w=1}^{N} exp\left(v'^{\mathrm{T}}_{w} v_{w_I}\right)}, \tag{3}$$

where v_w and v'_w are the embedding and context vector representations of w, respectively [45].

While CBOW and skip-gram models can calculate similarities between words, they do little about similarities between documents or sentences. Paragraph vectors or doc2vec models extend CBOW and skip-gram models [20], which learn document embeddings. Here the notion of ''paragraph'' represents text with varying lengths, which can be sentences, paragraphs, or whole documents.

The CBOW model is expanded so that the input layer also comprises the paragraph's ID containing the words. Therefore, every paragraph is mapped to a unique vector that is trained using the network. Since the paragraph ID acts as a memory that wires context to (missing) words, this model is called the distributed memory model of paragraph vectors (PV-DM). The other doc2vec model is the distributed bag-of-words model of paragraph vectors (PV-DBOW) that extends the word2vec skip-gram model [20], [41]. These two models are used to find similarities between sentences or documents and are leveraged in NLP search engines.

IV. THE PROPOSED METHOD

In this study, we first create the signatures of different software in the learning phase. Then, we check the system hard drive and look for the signatures of various software applications to identify the software running on the target system. For this purpose, we design a software signature search engine whose input documents are software signatures, and its query is the hard disk of the system under investigation. Figure 1 shows the component diagram of our proposed method for detecting software. The proposed method consists of two subsystems: signature construction and signature detection. In the following, we describe each of these subsystems and their components.

A. SIGNATURE CONSTRUCTION SUBSYSTEM

To make a software signature, we need to compare the disk copies immediately before and immediately after the software execution. We first extract valuable features from these two disk copies and compare them with each other. In this way, a difference-set is obtained, which can be processed to get a software signature. The signature construction subsystem consists of four components: feature extraction, differential analysis, difference-set construction, and signature construction. In the following, we describe each of these components.

1) FEATURE EXTRACTION

To extract features from a disk copy, we use fiwalk [46]. This tool processes a disk image into an XML structure representing all of the file system and document metadata resident within a disk image. In particular, the XML block that fiwalk produces has information about each file, such as the file name, file size, MACB timestamps, and MD5 and SHA1 abstracts.

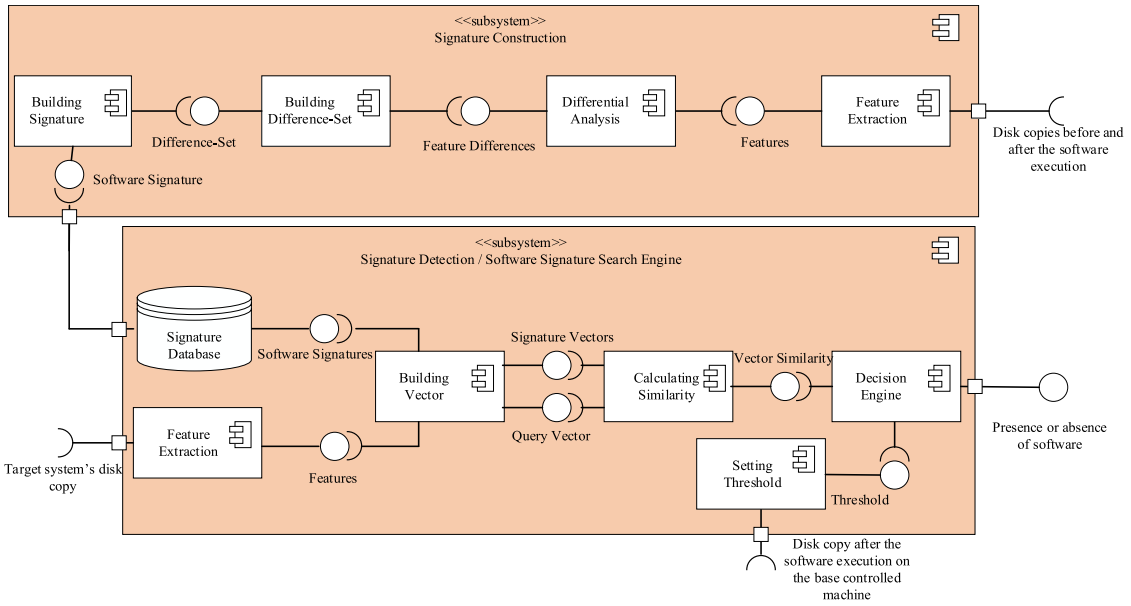

FIGURE 1. The component diagram of the proposed method for software detection.

2) THE GENERAL MODEL FOR FORENSIC DIFFERENTIAL ANALYSIS

The forensic differential analysis compares two digital artifacts and reports the differences between them. Forensic differential analysis can provide valuable information about what happened in a system. For example, comparing the memory images at two different time points gives information about created or terminated processes and established or terminated network connections. Besides, the differential analysis of two disk copies of a system explains the differences in the file system, Registry, and log files.

Artifacts or objects vary depending on the type of differential analysis. For example, to obtain a pattern of network traffic changes, the captured network packets act as objects. In this study, the disk copies are the objects that are compared to build the software signatures. Objects have different features, some of which are valuable for forensic purposes.

Objects can contain subobjects themselves. For example, if the object is a memory image, subobjects can be processes in memory; In this case, the features can be, for example, process status, number of resources allocated, and number of threads. However, the object can be a process itself, and the subobjects can be different blocks of memory allocated to it. Of course, sometimes we do not need to consider subobjects, and only the features of an object are important to us.

Suppose we have two objects A and B of the same type, collected at times T_A and T_B respectively, and $T_A < T_B$. Suppose object A has N_A subobjects, and object B has N_B subobjects. Also, assume that each subobject has M different valuable forensic features. So, we get $M \cdot N_A$ features from object A and $M \cdot N_B$ features from object B. The function E extracts features from each of these objects:

$$E : O \rightarrow F_O, \tag{4}$$

where $O = \{A, B\}$, and F_O is the set of all features extracted from subobjects of object O; In other words:

$$F_O = \left\{ f_{O_{i,j}} \mid i \in \{1, \cdots, N_O\}, j \in \{1, \cdots, M\} \right\}, \tag{5}$$

where $f_{O_{i,j}}$ is the value of jth feature extracted from the ith subobject of object O.

The differential analysis of the two objects A and B is reduced to the differential analysis of F_A and F_B. The differential analysis function is defined in (6), as shown at the bottom of the next page, where $PS(M)$ is the power set of the set $\{1, \cdots, M\}$, but it does not include the empty set and the set itself. Therefore, $PS(M)$ has 2^{M-1} elements. The function $DA(F_A, F_B)$ returns a set in which each element is an ordered triple in the form of (O, i, S_j), which shows the set of selected features (S_j) of the ith subobject of object O. If S_j has only one element j, we show the ordered triple as (O, i, j).

The differential analysis function (6) consists of four parts. The first part returns the ordered triples in the form of (A, i, j) if the value of the jth feature of none of the subobjects of B equals the value of the jth feature of the ith subobject of the object A. Similarly, the second part returns the ordered triples in the form of (B, i, j) if the value of the feature j of the subobject i of the object B equals none of the values of the feature j of the subobjects of A. The third part returns ordered triples (A, i, S_j) if for subobject i of A, there exists a subobject of B like k so that all of the k's features in S_j are equal to the corresponding feature of i, and other features of k are not equal to the corresponding feature of i. Likewise, the fourth part returns ordered triples (B, i, S_j) if for subobject i of B, there is a subobject of A like k so that all of the

k's features in S_j are equal to the corresponding feature of i, and other features of k are not equal to the corresponding feature of i.

Generally speaking, differential analysis lists features that a) exist in object A but do not exist in object B, b) do not exist in object A but exist in object B, and c) changed between two objects. We should note that, in practice, differential analysis is usually performed with a small number of features. Also, it is often not necessary to consider all features in all parts of (6), and usually, a subset of $\{1, \cdots, M\}$ or a subset of $PS(M)$ is considered.

3) FORENSIC DIFFERENTIAL ANALYSIS FOR BUILDING SOFTWARE SIGNATURE

Here, the disk copies are the objects, and files and folders in these copies are the subobjects. Each of these subobjects has many features, some of which have forensic value. For example, some metadata of files and folders, including path, size, various timestamps (modification, access, and create), and hash values, are valuable for forensic analysis.

When comparing two disk copies to build the signatures, we are interested in the following:

- Deleted files and folders.
- Created files and folders.
- Files that have changed. The change can be one of the following:
  - ✓ Files whose content has changed (the hash value has changed).
  - ✓ Files whose content has changed, but the modification timestamp has not changed. It could indicate a hardware problem, a software error, or a malicious effort. However, this could also mean that the modified files have not been saved.
  - ✓ Files whose modification timestamp has changed, but their content has not changed. In addition to the possibility of malicious manipulation, this may also happen when we make changes to a file but then undo the changes.

Therefore, for differential analysis of two disk copies, we consider three features: 1) the file path, 2) the modification timestamp, and 3) the hash value.

To formally describe the software signature construction, we use the proposed differential analysis model. To follow the terminology of the differential analysis model, we call the two disk copies taken before and after the software execution A and B, respectively. We also display our three features, i.e., the file path, the modification timestamp, and the hash value with p, t, and c, respectively.

The formal representation of deleted files and folders is defined in (7).

$$\begin{aligned} &Del\,(F_A, F_B) \\ &\quad = \bigcup_{i\in\{1,\cdots,N_A\}, j\in\{p\}} \{(A,i,j) \mid \neg\exists k \in \{1,\cdots,N_B\} : \\ &\qquad \times f_{A_{i,j}} = f_{B_{k,j}}\}. \end{aligned} \tag{7}$$

Equation (7) lists files and folders whose file path value is present in object A but not in object B.

The formal representation of created files and folders is defined in (8).

$$\begin{aligned} &Cre\,(F_A, F_B) \\ &\quad = \bigcup_{i\in\{1,\cdots,N_B\}, j\in\{p\}} \{(B,i,j) \mid \neg\exists k \in \{1,\cdots,N_A\} : \\ &\qquad \times f_{B_{i,j}} = f_{A_{k,j}}\}. \end{aligned} \tag{8}$$

Equation (8) lists files and folders whose file path value is present in object B but not in object A.

The formal representation of files whose content has changed is defined in (9).

$$\begin{aligned} &Mod\,l(F_A, F_B) \\ &= \bigcup_{i\in\{1,\cdots,N_B\}, S_j\in\{p\}} \left\{ (B,i,S_j) \;\middle|\; \begin{array}{l} \exists k \in \{1,\cdots,N_A\} : \forall j \in S_j : \\ f_{B_{i,j}} = f_{A_{k,j}} \wedge \\ \left(\forall h \in \{t,c\} : f_{B_{i,h}} \neq f_{A_{k,h}}\right) \end{array} \right\}. \end{aligned} \tag{9}$$

Equation (9) lists the files in object B whose hash value and timestamp have changed, but the file path has not changed. Since we are interested in the changes made to the disk's copy after running the software, we use this form of the equation and return the ordered triples that list the files and features in object B. Another form of the equation that lists the ordered

$$\begin{aligned} &DA\,(F_A, F_B) \\ &= \bigcup_{i\in\{1,\cdots,N_A\}, j\in\{1,\cdots,M\}} \left\{(A,i,j) \mid \neg\exists k \in \{1,\cdots,N_B\} : f_{A_{i,j}} = f_{B_{k,j}}\right\} \\ &\quad \cup \bigcup_{i\in\{1,\cdots,N_B\}, j\in\{1,\cdots,M\}} \left\{(B,i,j) \mid \neg\exists k \in \{1,\cdots,N_A\} : f_{B_{i,j}} = f_{A_{k,j}}\right\} \\ &\quad \cup \bigcup_{i\in\{1,\cdots,N_A\}, S_j\in PS(M)} \left\{\left(A,i,S_j\right) \mid \exists k \in \{1,\cdots,N_B\} : \forall j \in S_j : f_{A_{i,j}} = f_{B_{k,j}} \wedge \left(\forall h \in \{1,\cdots,M\}, h \notin S_j : f_{A_{i,h}} \neq f_{B_{k,h}}\right)\right\} \\ &\quad \cup \bigcup_{i\in\{1,\cdots,N_B\}, S_j\in PS(M)} \left\{\left(B,i,S_j\right) \mid \exists k \in \{1,\cdots,N_A\} : \forall j \in S_j : f_{B_{i,j}} = f_{A_{k,j}} \wedge \left(\forall h \in \{1,\cdots,M\}, h \notin S_j : f_{B_{i,h}} \neq f_{A_{k,h}}\right)\right\}, \end{aligned} \tag{6}$$

triples of object A is given in (10). However, there is no need for it because the modified items are the same and do not need to be repeated.

$$Mod1'(F_A, F_B) = \bigcup_{i\in\{1,\cdots,N_A\}, S_j\in\{P\}} \left\{ (A, i, S_j) \left| \begin{array}{l} \exists k \in \{1,\cdots,N_B\} : \forall j \in S_j : \\ f_{A_{i,j}} = f_{B_{k,j}} \wedge \\ \left(\forall h \in \{t, c\} : f_{A_{i,h}} \neq f_{B_{k,h}}\right) \end{array} \right. \right\}. \tag{10}$$

The formal representation of files whose content has changed, but the modification timestamp has not changed is defined in (11).

$$Mod2(F_A, F_B) = \bigcup_{i\in\{1,\cdots,N_B\}, S_j\in\{p,t\}} \left\{ (B, i, S_j) \left| \begin{array}{l} \exists k \in \{1,\cdots,N_A\} : \forall j \in S_j : \\ f_{B_{i,j}} = f_{A_{k,j}} \wedge \\ \left(\forall h \in \{c\} : f_{B_{i,h}} \neq f_{A_{k,h}}\right) \end{array} \right. \right\}. \tag{11}$$

Equation (11) lists the files whose hash value has changed, but the file path and modification timestamp have not changed.

The formal representation of files whose modification timestamp has changed, but the content has not changed is defined in (12).

$$Mod3(F_A, F_B) = \bigcup_{i\in\{1,\cdots,N_B\}, S_j\in\{p,c\}} \left\{ (B, i, S_j) \left| \begin{array}{l} \exists k \in \{1,\ldots,N_A\} : \forall j \in S_j : \\ f_{B_{i,j}} = f_{A_{k,j}} \wedge \\ \left(\forall h \in \{t\} : f_{B_{i,h}} \neq f_{A_{k,h}}\right) \end{array} \right. \right\}. \tag{12}$$

Finally, the differential analysis of two disk copies before and after the software execution is the union of triples obtained from (7), (8), (9), (11), and (12).

In our experiments, we found that most of the deleted or modified files were not due to software execution, but to the underlying operating system. Therefore, to create a software signature, we only focus on the files and folders created during the software execution and consider only the file path feature, p. The differential analysis function for finding the mentioned differences between two disk copies, before and after running the software SW, is defined by (13).

$$Cre\left(F_{Pre(SW)}, F_{Post(SW)}\right) = \bigcup_{i\in\{1,\cdots,N_{Post(SW)}\}} \left\{ (i, p) \left| \begin{array}{l} \neg\exists k \in \{1,\cdots,N_{Pre(SW)}\} : \\ f_{Post(SW)_{i,p}} = f_{Pre(SW)_{k,p}} \end{array} \right. \right\}. \tag{13}$$

In this way, the differential analysis of two disk copies before and after the software execution creates ordered pairs representing the files and folders created during the software run. We get the difference-set of software execution by extracting the second element from these ordered pairs, as represented by (14).

$$DS(SW) = \left\{ p \mid (i, p) \in Cre\left(F_{Pre(SW)}, F_{Post(SW)}\right) \right\}. \tag{14}$$

For the software signature not to rely on just running the software once, we run each software with different scenarios, and after each run, we get the difference-set. Then, we combine these difference-sets and get the software signature. This combination, unlike the union operation of sets, does not remove duplicate elements. If the file path p exists in r difference-sets, p appears in the final signature r times. If the software SW runs with K different scenarios $s_1, \cdots, s_K$, we will have K difference-sets that should be combined to get the software signature, as represented by (15).

$$Sig(SW) = \bigcup_{i\in\{1,\cdots,K\}} DS\left(SW_{s_i}\right). \tag{15}$$

4) BUILDING DIFFERENCE-SET AND SIGNATURE OF THE SOFTWARE

To build the signature for each software, we proceed as follows. On a virtual machine, we install the operating system X and uninstall the default OS programs and take a snapshot of the system called *Snapshot_Base*. Then we install the desired software on the system and run it with the relevant scenarios, and after each run, we take a copy of the disk. In the following, we will explain how to make the signature for the software *app*.

We install the software *app* on the system, suspend the system, convert the virtual machine disk (.vmdk file) to E01 format using a forensic tool (such as FTK Imager or EnCase Forensic Imager) to get the *app_install.E01*. Then we resume the system and make a series of initial settings of the software *app* and take a snapshot of the system called *Snapshot_app_initial_setting*. We run the software *app* with the first scenario, suspend the system, and convert the resulting.vmdk file to *app_run_1.E01*. Then we restore the system to the *snapshot_app_initial_setting*. Similarly, we run *app* with all the scenarios and take a copy of the disk.

We should note that before each scenario execution of the *app*, we must restore to the *snapshot_app_initial_setting*. This way, every time, the software *app* runs on a system on which the software is recently installed and configured. Also, we need to go back to the *snapshot_Base* to install any software.

After installing and running all *apps* and getting the disk copies, we process the copies with fiwalk [46] to get the files and folders in the disk copy as a DFXML[1] output [47]. Now we use the differential analysis model to find files and folders created, deleted, or changed while running software with a specific scenario. For this purpose, we compare the two DFXML files after installing and after running the *app* and get the differences using the *make_differential_dfxml* command. However, as mentioned in the previous section, due to the high false-positive rate of deleted and modified items, we focus only on the added items to build the difference-set. Then we combine the difference-sets of software and create

[1]Digital Forensic XML (DFXML) is an XML language that allows the exchange of structured forensic information.

Algorithm 1 Create software signature

1: Install operating system X on a virtual machine
2: Take a snapshot of the system called *Snapshot_Base*
3: *apps* ← The set of applications
5: **for** *app* in *apps* **do**
6: Install *app*
7: Create *app_install.E01*
8: Perform initial settings
9: Take a snapshot of the system called *Snapshot_app_initial_setting*
10: **for***i* in related_scenarios **do**
11: Run app with scenario *i*
12: Create *app_run_i.E01*
13: Restore virtual machine to *Snapshot_app_initial_setting*
14: **end for**
15: Restore virtual machine to *Snapshot_Base*
16: **end for**
17: **for all**disk copies as *d.E01* **do**
18: fiwalk *d.E01 d.dfxml*
19: **end for**
20: **for***app* in *apps* **do**
21: *signature_app* = {}
22: **for***i* in related_scenarios **do**
23: make_differential_dfxml *app_install.dfxml app_run_i.dfxml* ≫ *delta_app_run_i.dfxml*
24: *difference-set_app_run_i* = added items in *delta_app_run_i.dfxml*
25: *signature_app* = *signature_app* ∪ *difference-set_app_run_i*
26: **end for**
27: **end for**

the software signature. Algorithm 1 describes how to build a software signature.

B. SIGNATURE DETECTION SUBSYSTEM

In this section, we provide a solution to detect the presence of software on the target system. The software detection subsystem or software signature search engine consists of six components:

1) *software signature database:* The software signatures created by the software construction subsystem make the signature database. These software signatures, after processing, make the input documents of our S3E.
2) *feature extraction:* The hard disk of the target system is processed, and valuable forensic features (the file path attribute of all files and folders) are extracted. These extracted features make up our query.
3) *vector construction:* We need a method to compare the query with the signatures of different software. Therefore, we should find a numerical representation of this textual data so that each signature or query is represented as a vector in a multidimensional space. As described in Section III, we use the paragraph vector model to construct the signature vectors and the query vector.
4) *similarity calculation:* If a piece of software has run on the target machine, the queried hard disk must be similar to the software signature. This module calculates the cosine similarity of the query vector and various signature vectors.
5) *threshold setting:* We should note that unlike conventional search engines, which return a list of related documents in response to a query, the software signature search engine must determine whether the software has run on the target system. Therefore, we need to define a threshold for the presence of any software. If the similarity score is greater than the threshold, we conclude that the software has run on the system.
   To set the software threshold, we install and run the software on a base-controlled system. The base-controlled system is a newly installed system on which no software has been installed yet, and even the default Windows programs have been uninstalled. This system acts as a query, and the similarity score against the corresponding software signature makes the software threshold. Suppose the search engine has n software signatures. To determine the threshold of the ith software, we install and run it on the base-controlled system. Then we process the disk copy and give it to the search engine as a query. The similarity score of this query with the ith document of the search engine is the threshold of the ith software.
6) *decision engine:* If the similarity of the query vector with a software signature vector is greater than the

software threshold, the decision engine concludes that the software has run on the system.

C. DESIGN PARAMETERS OF SOFTWARE SIGNATURE SEARCH ENGINES

We consider several design parameters in developing software signature search engines. Our first design choice is to choose the doc2vec model type (PV-DM or PV-DBOW). The second and third design parameters consider different values for the vector size and window size, respectively. Finally, the fourth design parameter sets the threshold value for each software. Table 1 shows these parameters and their different values. Different values of design parameters lead to different S3E models. The total number of S3E models is $2 \times 4 \times 5 \times 3 = 120$. We have used some tags to make it easier to name the models. For example, the S3E model, which uses PV-DM and has a vector size of 80 and a window size of 10 and uses a medium threshold, is known as P1-V3-W5-T2.

TABLE 1. Design parameters of our software signature search engines.

Parameter	Value	Tag
paragraph model	PV-DM	P1
	PV-DBOW	P2
vector size	20	V1
	40	V2
	80	V3
	160	V4
window size	2	W1
	4	W2
	6	W3
	8	W4
	10	W5
Threshold value	Big	T1
	Medium	T2
	Small	T3

The first parameter of the software signature search engine determines the representation model for software signatures. As stated in Section III, the PV-DM model predicts a word using the context words and the paragraph containing it. For this purpose, a three-layer neural network is used, which has context word vectors and the paragraph vector in its first layer. These vectors are averaged or concatenated to predict the target word. Once the neural network is trained, the trained vectors for the paragraphs can be used as paragraph signatures.

The second model, PV-DBOW, predicts words in a paragraph using only the paragraph (without using other context words). Therefore, the first layer of the neural network has only one paragraph vector. This model is conceptually simpler than PV-DM and requires less storage space; Unlike the PV-DM method, which requires storing softmax weights and word vectors, this method only needs to store softmax weights [20].

The second and third parameters adjust vector dimensions and window size. For vector dimensions, we have 20, 40, 80, and 160, and for window size, we have 2, 4, 6, 8, and 10. The fourth parameter determines the threshold value for different signatures. As mentioned earlier, we need to set a threshold for the presence of any software. Every search engine model has a separate threshold per software.

Note that the first three design parameters affect the threshold value. After developing the S3E model using these parameter values, we set the threshold for each software. The Big value for the threshold is calculated according to the description in the previous section. The Medium value is half of the Big one, and the Small value is one-quarter of the Big.

V. EXPERIMENTS, EVALUATION AND RESULTS

A. EXPERIMENTS

In our experiments, we run two versions of several software applications with different scenarios on two operating systems (Windows 7 x32 and Windows XP x32). We choose the 32-bit version of Windows XP to have the same OS as the M57 computers [21]. Table 2 describes the scenarios of these software packages. The first versions (Adobe Reader 9.2, Firefox 3.5.10, and Python 2.6.1) are the same as the applications executed on the M57 computers.

As stated in Section IV, we combine the difference-sets of various software execution scenarios to get the software signature. Therefore, our S3E models have 16 input documents, each with an average of 954 sentences and 4942 words. However, we should note that the meaning of words and sentences in software signatures is different from their meaning in natural languages. Here the sentences are complete file paths, and the words are the names of each of the folders and files in the path. Unlike common documents, where words are separated by spaces, signature words are separated by / character. We need to do the necessary preprocessing on software signatures to introduce these words to PV-DM and PV-DBOW algorithms.

As mentioned, the word2vec and doc2vec models pay attention to word context and place word vectors with similar contexts close together. To display word vectors used in software signature, we first need to convert the high dimensional word vectors into two-dimensional points. We use the t-distributed stochastic neighbor embedding (t-SNE), which is a nonlinear dimensionality reduction technique [48].

Figure 2 shows only a small part of this two-dimensional space. Each point in this figure represents one of the components of the file paths in the software signature of Adobe Reader. A small piece of these file paths is shown in Figure 3. Comparing Figure 2 and Figure 3, we find that the vectors of words with similar contexts are almost close. For example, most of the last components of the file paths are located in the middle of Figure 2. Also, the second components *administrator* and *all users* are located close to each other. Similarly, the third components, including *application data*, *local settings*, and *desktop*, are located close to each other in the vector space. Moreover, the two components *9.0* and *11.0*, which represent the Adobe versions, are close together.

TABLE 2. Description of performed experiments.

Software	Scenario	Scenario Description
Adobe Reader 9.2 & Adobe Reader 11.0.3	S1_1	Clicking on Adobe Reader icon on the desktop, opening file1.pdf, and finally, closing it
	S1_2	Clicking on Adobe Reader icon in the Start menu, opening file1.pdf, and finally, closing it
	S1_3	Clicking on Adobe Reader icon on the desktop, then opening file2.pdf, saving it to another file, and finally, closing it
	S1_4	Opening file2.pdf by double-clicking it and then closing it
Adobe Reader 11.0.3	S1_5	Opening file2.pdf by double-clicking it, then making changes to it, saving it, and finally closing it
	S1_6	Opening file1.pdf by double-clicking it, then making changes to it, saving it to another file, and finally closing it
Firefox 3.5.10 & Firefox 40.0.3	S2_1	Clicking on Firefox icon on the desktop, visiting yahoo.com, and finally, closing it
	S2_2	Clicking on Firefox icon in the Start menu, then visiting google.com, searching for Olympic, opening www.olympic.org in a new tab, and finally, closing it
	S2_3	Clicking on the Firefox icon on the desktop, then visiting gmail.com, logging in, opening an email, downloading the email attachment, which was a pdf file, and finally closing it
	S2_4	Clicking on the Firefox icon in the Start menu, opening the en.um.ac.ir, and then reviewing several successive links on this site and finally closing it
Python 2.6.1 & Python 2.7.17	S3_1	Opening Python by selecting Run and typing python, then typing the command *print "hello world"*, and finally closing it
Word 2003 pro & Word 2010 pro	S4_1	Opening the Word by clicking on its icon on the desktop, creating a new document, saving it as doc1 on the desktop, and finally closing it
	S4_2	Opening the Word by clicking on its icon in the Start menu, opening doc2, making changes to it, saving it, and closing it
	S4_3	Double-clicking on the doc2, making changes to it, saving the changes to the doc3 file, and finally closing it
	S4_4	Right-clicking on the desktop and selecting New Microsoft Word Document and naming it doc1, then double-clicking it, typing a text and saving it, and finally, closing it

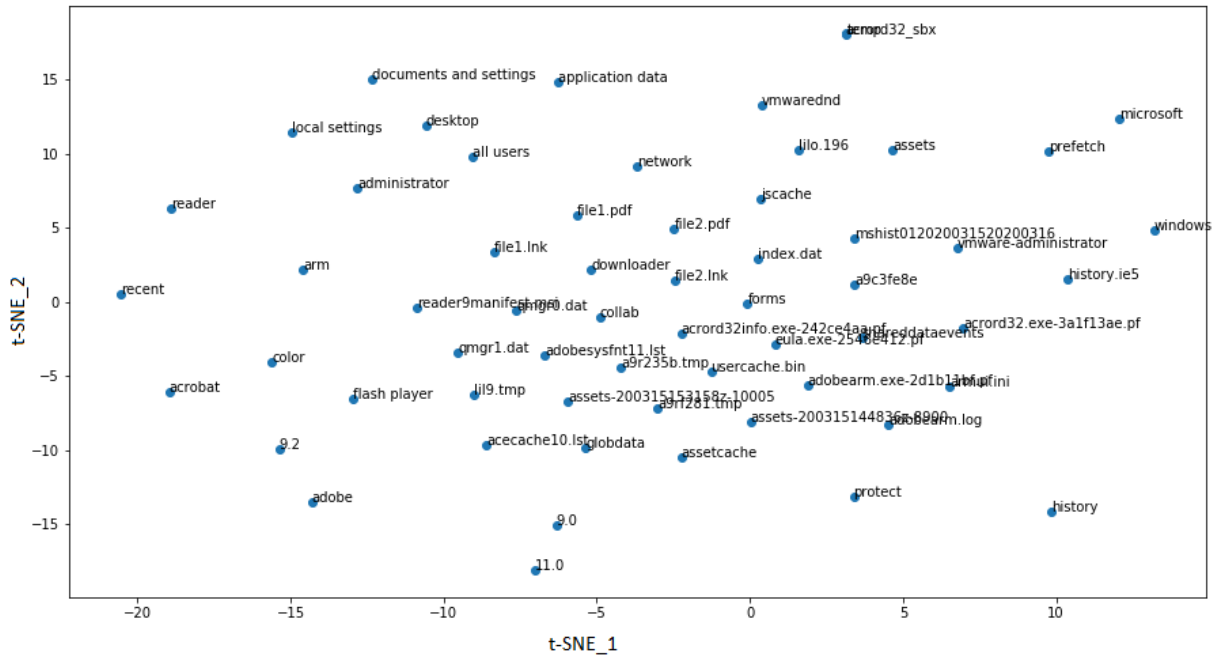

FIGURE 2. Two-dimensional representation of software signature word vectors.

B. CONTROLLED MACHINES

To evaluate each S3E model, we design controlled machines on which we already know what applications run. We design 20 controlled machines: 10 machines with the specification of Intel Core i5 processor with 2.50 GHz 8 GB RAM, running Windows 7 and 10 machines with the same specification, running Windows XP. How to run the applications on both Windows is the same. Table 3 describes the controlled machines.

We query the controlled machines one by one against each S3E model and calculate its similarity with different input signatures. For the software x that has a signature in the search engine, if x is present on the controlled machine, the similarity of this queried machine with the signature of x should be

```
Documents and Settings/Administrator/Application Data/Adobe/Acrobat/9.0/Collab/
Documents and Settings/All Users/Application Data/Microsoft/Network/Downloader/qmgr0.dat/
Documents and Settings/All Users/Application Data/Microsoft/Network/Downloader/qmgr1.dat/
Documents and Settings/Administrator/Local Settings/History/History.IE5/MSHist012020031520200316/index.dat/
Documents and Settings/Administrator/Recent/file2.lnk/
Documents and Settings/Administrator/Local Settings/Application Data/Adobe/Color/
Documents and Settings/Administrator/Local Settings/Temp/AdobeARM.log/
Documents and Settings/Administrator/Local Settings/Temp/vmware-Administrator/VMwareDnD/a9c3fe8e/file1.pdf/
Documents and Settings/Administrator/Desktop/file1.pdf/
Documents and Settings/All Users/Application Data/Microsoft/Network/Downloader/
Documents and Settings/Administrator/Desktop/file2.pdf/
WINDOWS/Prefetch/EULA.EXE-2546E412.pf/
Documents and Settings/Administrator/Application Data/Adobe/Acrobat/9.0/
Documents and Settings/Administrator/Local Settings/Application Data/Adobe/
WINDOWS/Prefetch/ACRORD32.EXE-3A1F13AE.pf/
WINDOWS/Prefetch/ADOBEARM.EXE-2D1B11BF.pf/
Documents and Settings/Administrator/Application Data/Adobe/Acrobat/9.0/SharedDataEvents/
WINDOWS/Prefetch/ACRORD32INFO.EXE-242CE4AA.pf/
Documents and Settings/Administrator/Application Data/Adobe/Acrobat/
Documents and Settings/All Users/Application Data/Adobe/Reader/9.2/ARM/Reader9Manifest.msi/
Documents and Settings/Administrator/Application Data/Adobe/Acrobat/9.0/UserCache.bin/
Documents and Settings/All Users/Application Data/Adobe/Reader/
Documents and Settings/All Users/Application Data/Adobe/Reader/9.2/ARM/
```

FIGURE 3. Part of file paths in software signature of Adobe Reader.

TABLE 3. Description of controlled machines.

name	Description
Controlled machine 1	We execute the first version of all four applications, namely Adobe Reader 9.2, Firefox 3.5.10, Python 2.6.1, and Word 2003 pro one after the other, without closing them.
Controlled machine 2	We execute the first version of all four applications, namely Adobe Reader 11.0.3, Firefox 40.0.3, Python 2.7.17 and Word 2010 pro one after the other, without closing them.
Controlled machine 3	We execute three applications Adobe Reader 9.2, Firefox 3.5.10, and Python 2.6.1 one after the other, without closing them.
Controlled machine 4	We execute three applications Firefox 3.5.10, Python 2.6.1, and Word 2003 pro one after the other, without closing them.
Controlled machine 5	We execute three applications Adobe Reader 9.2, Python 2.6.1, and Word 2003 pro one after the other, without closing them.
Controlled machine 6	We execute three applications Adobe Reader 9.2, Firefox 3.5.10, and Word 2003 pro one after the other, without closing them.
Controlled machine 7	We execute three applications Adobe Reader 11.0.3, Firefox 40.0.3, and Python 2.7.17 one after the other, without closing them.
Controlled machine 8	We execute three applications Firefox 40.0.3, Python 2.7.17, and Word 2010 pro one after the other, without closing them.
Controlled machine 9	We execute three applications Adobe Reader 11.0.3, Python 2.7.17, and Word 2010 pro one after the other, without closing them.
Controlled machine 10	We execute three applications Adobe Reader 11.0.3, Firefox 40.0.3, and Word 2010 pro one after the other, without closing them.

greater than or equal to the threshold of x. Conversely, if x is not present on the controlled machine, the similarity of the query to x's signature should be less than the x's threshold.

C. EVALUATION AND RESULTS

To evaluate software signature search engines, we first need to define evaluation criteria. The efficiency of information retrieval methods and search engines is usually measured by precision and recall [49]–[51]. Precision is the number of related results retrieved by the search engine divided by the total number of retrieved items. Recall is the number of related results retrieved by the search engine divided by the total number of related items in the collection. To calculate the precision and recall, it is necessary to determine the values of true-positive, true-negative, false-positive, and false-negative, each of which is defined as follows:

- True-positive (TP) is the number of software packages that the search engine correctly detects them running on the system.
- True-negative (TN) is the number of software packages that the search engine correctly detects them not running on the system.
- False-positive (FP) is the number of software packages that the search engine falsely detects them running on the system.
- False-negative (FN) is the number of software packages that the search engine falsely detects them not running on the system.

We calculate true-positive, true-negative, false-positive, and false-negative values cumulatively for each controlled machine. In other words, for each controlled machine, we get these values and add them together, and finally, we calculate the precision and recall using these cumulative values.

Algorithm 2 Calculate performance metrics for a software signature search engine

```
Apps ← The set of input applications of the software signature search engine
machines ← 20 controlled machines
for m in machines do
   machine_apps ← The set of applications in m
   for i in Apps do
      if i in machine_apps then
         if similarity(i) >= threshold(i) then
            TP ← TP + 1
         else
            FN ← FN + 1
         end if
      else
         ifsimilarity(i) >= threshold(i) then
            FP ← FP + 1
         else
            TN ← TN + 1
         end if
      end if
   end for
end for
Precision ← TP / (TP + FP )
Recall ← TP / (TP + FN )
```

Algorithm 2 describes how to calculate the performance metrics for a software signature search engine.

We should note that we run each S3E model 10-times and average the precision and recall values of these 10-runs. The reason is that our software signature search engine is based on neural networks, and the document vectors are learned using a probabilistic modeling approach. Unlike frequency-based vector representations such as TF-IDF, in which the vector of a document or a query is non-variable, the probability-based vectors are different each time the neural network is trained or tested.

As mentioned earlier, we have designed 120 different models of software signature search engines. In the following, we examine the precision and recall of different search engine models and examine the effect of different design parameters on these performance metrics.

1) EXPERIMENTAL RESULTS ON CONTROLLED MACHINES

Figure 4 shows the precision and recall distribution of our 120 S3E models against 20 controlled machines. We see that the minimum precision of these models is more than 0.795. However, none of the models have reached perfect precision. The minimum recall of these models is about 0.75, and some models have perfect recall. We can divide these points into two categories: the first category (A) has a recall of less than 0.85 and a precision of more than 0.8, and the second group (B) has a recall of more than 0.85 and a precision of less than 0.805. We can see that models in category A have higher precision and lower recall, and models in category B have higher recall and lower precision. To see which models fall into each of these two categories, we can look at the diagrams in Figure 5.

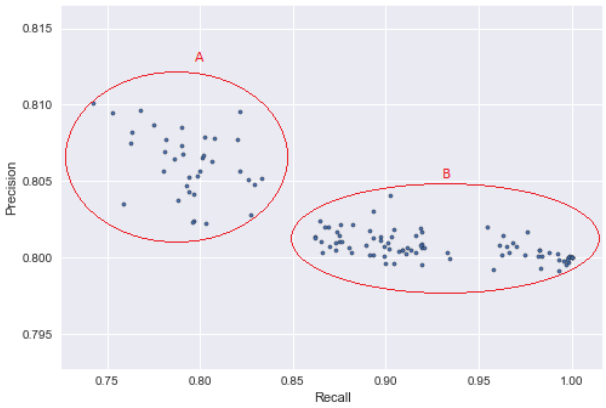

FIGURE 4. The precision and recall of 120 S3E models against controlled machines.

Each of the diagrams in Figure 5 shows the precision and recall for one of the design parameters. Each of the two doc2vec models, PV-DM and PV-DBOW, has relatively the same share in two categories, A and B. However, in category B, PV-DM models have better performance in terms of recall than PV-DBOW models, so that most models with a recall of more than 0.9 and all models with a recall of more than 0.95 use the PV-DM algorithm. Also, in Category A, the PV-DM models perform better and achieve better precision values.

Different values of vector dimensions do not operate distinctly in the precision-recall space. In other words, in both categories, all four values have a relatively equal share. It is

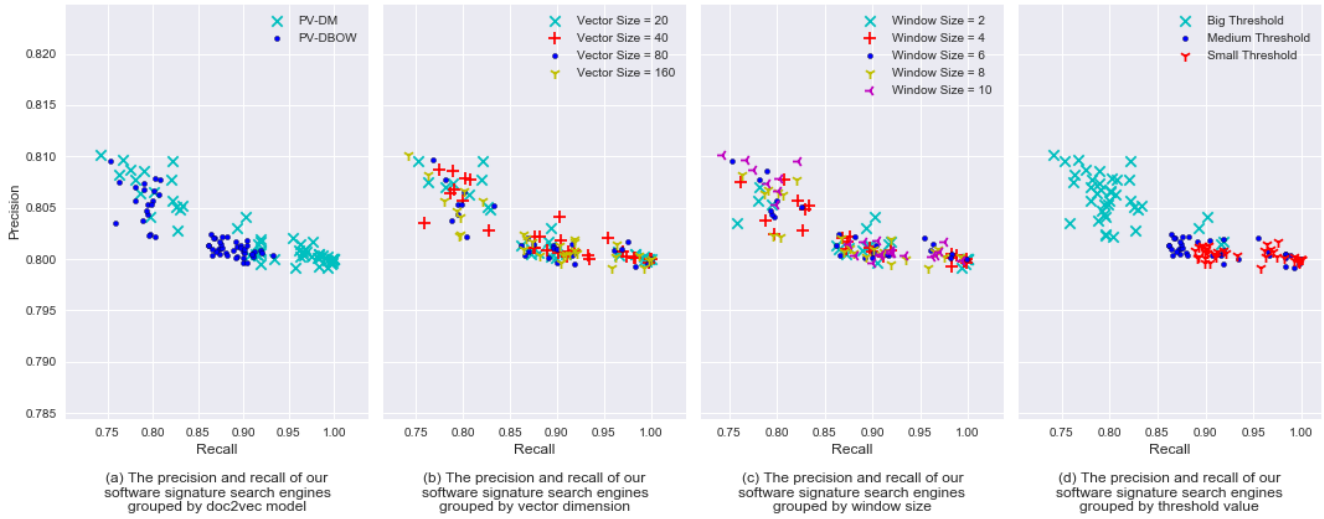

FIGURE 5. The precision and recall of our S3E models against controlled machines grouped by design parameters.

also true for the window size parameter. Different threshold values behave separately in the precision-recall space. All members of category A are Big threshold models. Also, almost all models with Big threshold are in category A. Medium, and Small threshold models are in category B. However, Small threshold models account for a larger share of models with higher recall.

In short, the PV-DM algorithm performs better in our software signature search engine, but the vector length and the sliding window size have no noticeable effect on the performance of our search engine. Moreover, we can say that larger threshold values lead to greater precision, and smaller values lead to better recall.

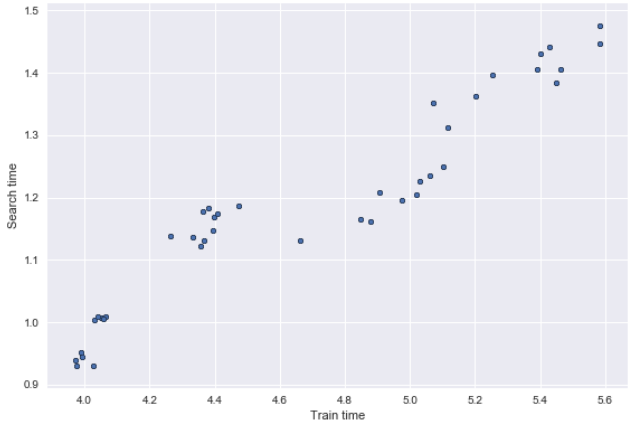

FIGURE 6. The training and searching times (in seconds) for different S3E models.

As mentioned, the software signature search engine first uses one of the doc2vec algorithms to train software signature vectors. Then, when processing the query, it calculates the cosine similarity of the query vector with each of the software signature vectors. Figure 6 shows the training time and the query time for each of the S3E models. As can be seen, the total training and query time in these models is not more than 7 seconds. Therefore, in a short time, an S3E model can detect software on a system.

Of course, before model training, it is necessary to convert the target disk to DFXML format. For example, converting a 4GB disk copy to DFXML format on a BitCurator virtual machine with 8GB RAM takes about 2 minutes. It can be estimated that this conversion takes about 40 minutes for an 80GB disk copy. This conversion is done much faster on a powerful physical machine. Therefore, the presence of any software by an S3E model can be checked in less than an hour.

However, working with conventional digital forensic tools takes more time. For example, loading and indexing an 80GB disk copy takes about 4 hours with AccessData FTK 4.2 and about 10 hours with Encase Forensic 7 on a PC with 8GB RAM [52]. Besides, after loading and indexing the copy, the investigator needs to look for traces of various applications among a large number of files in the disk copy. For example, the disk copy of Controlled Machine 1 in Table 3 on Windows 7 has more than 100,000 files and folders. This manual process is time-consuming and depends entirely on the investigator's experience.

Figure 7 shows the average neural network training time for the search engine and the average query time for the controlled machines for different design parameters.

In this figure, we see that the average training and searching times for PV-DM models are longer than PV-DBOW models. We also see that with increasing vector size and window size, these two times increase. However, there is only one exception in the size of the vector equal to 20. Moreover, we see no difference in the average training and query times for different threshold values. The reason is that the threshold value does not affect training or testing the model.

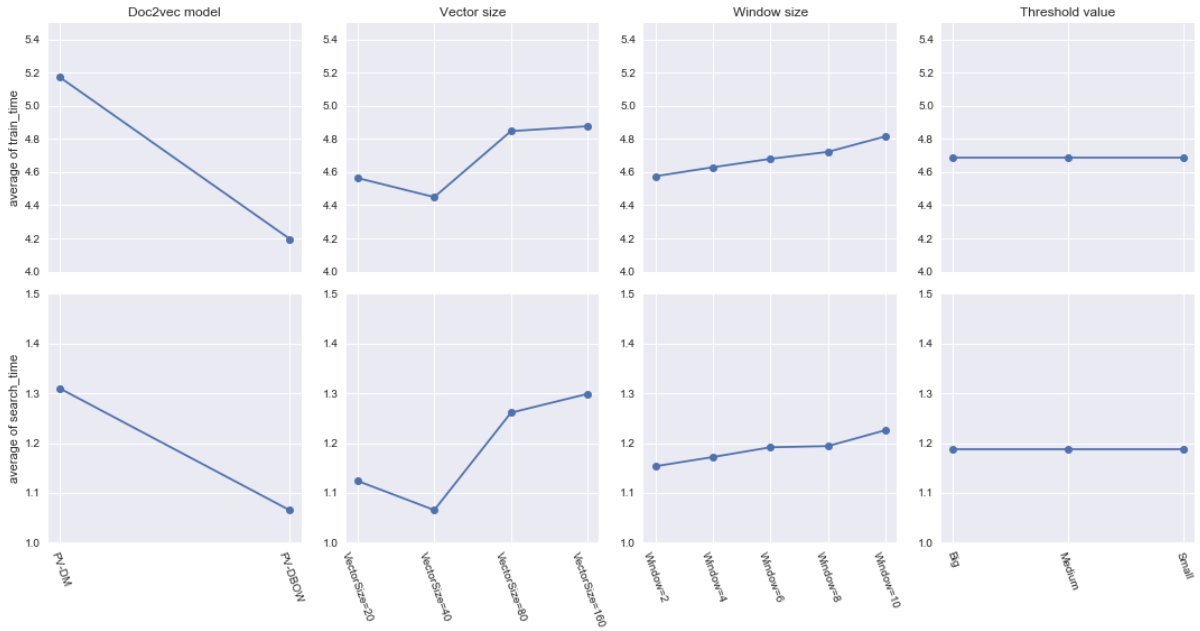

FIGURE 7. The average training and searching times (in seconds) for different design parameters.

2) EXPERIMENTAL RESULTS ON M57 MACHINES

Roussev and Quates [9] report a list of software in the M57 Patents corpus [21] that includes three of our test software (Adobe Reader, Firefox, and Python). Firefox and Python were running on Pat's computer on November 16, and Adobe Reader was running on Pat's computer on November 19. In the following, we will examine Pat's disk copy in these two days to discover these software packages.

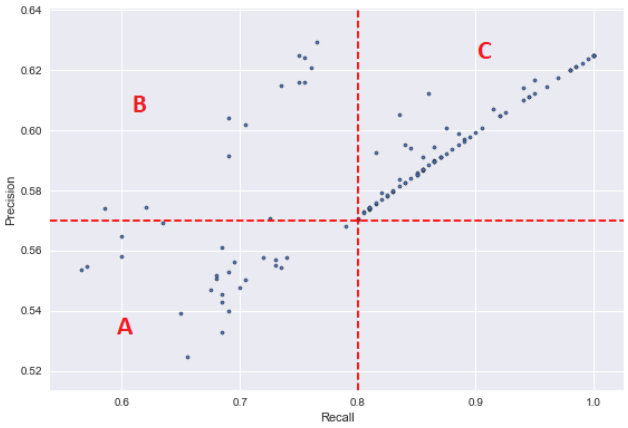

FIGURE 8. The precision and recall of our software signature search engines against M57 machines.

Figure 8 shows the precision and recall distribution of our S3E models against M57 machines. The precision of S3E models is in the approximate range of (0.52, 0.63), and the recall is in the range of (0.55, 1). None of the models reach high precision. Nevertheless, some models have perfect recall, and 67.5% of them have a recall greater than 0.8. High recall means that these models have been relatively successful in detecting applications running on M57 machines.

We can divide the points on the precision-recall space into three categories: The models that have a precision of less than 0.57 and a recall of less than 0.8 (Category A), the models with a precision of more than 0.57, and a recall of less than 0.8 (Category B), and the models with a precision of more than 0.57 and a recall of more than 0.8 (Category C).

The diagrams in Figure 9 show the precision and recall values for the various design parameters. We see that category A consists mostly of PV-DBOW models, category B mostly consists of PV-DM models, and category C consists of both models equally. However, almost all of the top models in this category are PV-DMs. So, the PV-DM models perform better than the PV-DBOW models against the M57 in terms of precision and recall, the same thing we had in the controlled machines.

Although there is no clear distinction between different vector size values in the above classification, category A mostly includes models with smaller vector size, and category B mostly contains models with larger vector size. In category C, the number of models with different vector sizes is almost the same. There is no significant difference between the different values of window size in this categorization.

Regarding the threshold, we see that almost all category A elements and all category B elements are big threshold models. Category C includes models with medium and small thresholds. However, most of the top models in this category have small thresholds. So, we see that some models with a big threshold have achieved relatively high precision and that the small threshold has performed well in terms of both precision and recall.

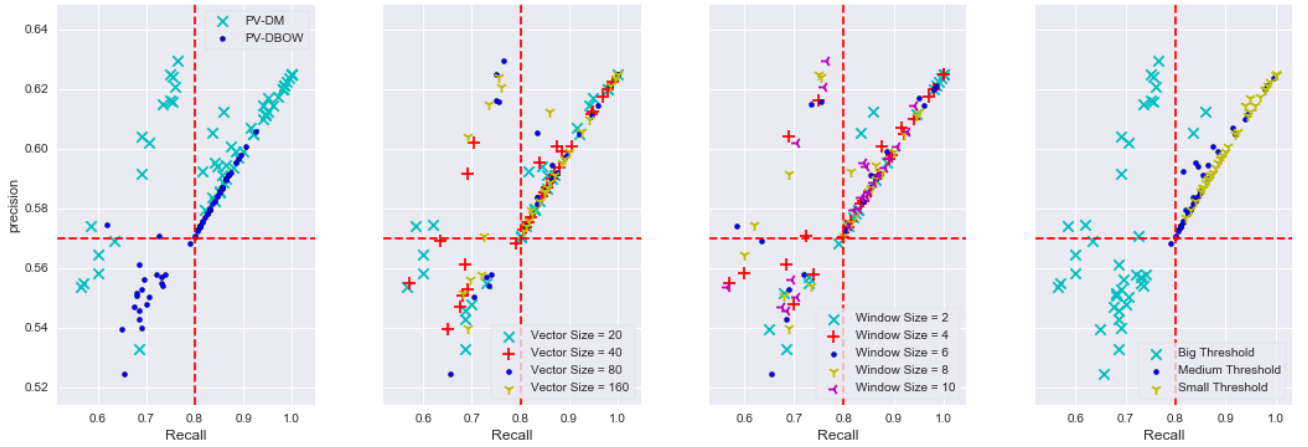

FIGURE 9. The precision and recall of our S3E models against M57 machines grouped by design parameters.

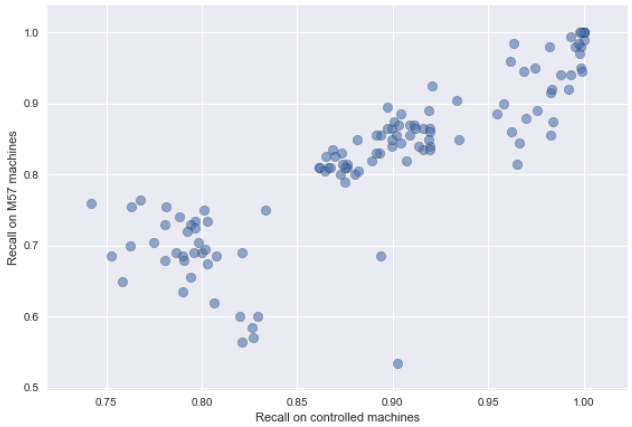

FIGURE 10. The recall rate of S3E models in both datasets.

3) EXPERIMENTAL RESULTS ON BOTH DATASETS

Figure 10 shows the recall rate of the S3E models against controlled machines and M57 machines. We see that some models perform well in both datasets. To see which design values resulted in superior models for both datasets, we can look at Table 4 and Table 5. Table 4 lists the top 20 S3E models based on recall on controlled machines, and Table 5 lists the high recall models on M57 machines.

All of the top models in Table 4 are PV-DM and have a medium or small threshold. Also, the small size of the vector is more prominent than other sizes. Besides, the smaller sizes of the sliding window are more prominent among the top models. In Table 5, we see that all the top models are PV-DM and have a medium or small threshold. However, the number of models with a small threshold is more than the medium threshold. Also, the small size of the vector is more prominent than other sizes. Nevertheless, models with different window sizes have the same presence. We also see that 15 out of the 20 models in Table 4 are also present in Table 5 and are among the top models in both datasets.

TABLE 4. Top 20 S3E models based on recall on controlled machines.

	Name	Recall on controlled machines		Name	Recall on controlled machines
1	P1-V1-W1-T3	1	11	P1-V3-W1-T2	0.997
2	P1-V2-W1-T3	1	12	P1-V1-W2-T3	0.996
3	P1-V3-W1-T3	1	13	P1-V1-W5-T3	0.993
4	P1-V4-W1-T3	1	14	P1-V4-W1-T2	0.993
5	P1-V2-W1-T2	0.999	15	P1-V1-W4-T3	0.992
6	P1-V1-W1-T2	0.998	16	P1-V4-W2-T2	0.988
7	P1-V1-W3-T3	0.998	17	P1-V2-W2-T2	0.984
8	P1-V3-W2-T3	0.998	18	P1-V3-W2-T2	0.983
9	P1-V2-W2-T3	0.998	19	P1-V1-W2-T2	0.983
10	P1-V4-W2-T3	0.998	20	P1-V1-W3-T2	0.983

TABLE 5. Top 20 S3E models based on recall on M57 machines.

	Name	Recall on M57 machines		Name	Recall on M57 machines
1	P1-V1-W1-T3	1	11	P1-V1-W2-T3	0.98
2	P1-V3-W1-T3	1	12	P1-V2-W3-T3	0.98
3	P1-V3-W2-T3	1	13	P1-V2-W2-T3	0.97
4	P1-V4-W1-T3	1	14	P1-V3-W3-T3	0.96
5	P1-V4-W2-T3	1	15	P1-V1-W3-T3	0.95
6	P1-V4-W1-T2	0.995	16	P1-V2-W4-T3	0.95
7	P1-V2-W1-T3	0.99	17	P1-V2-W1-T2	0.945
8	P1-V3-W1-T2	0.985	18	P1-V3-W4-T3	0.945
9	P1-V4-W3-T3	0.985	19	P1-V1-W5-T3	0.94
10	P1-V1-W1-T2	0.98	20	P1-V4-W2-T2	0.94

Figure 11 shows the precision rate of the S3E models against controlled machines and M57 machines. Since the precision changes of S3E models in controlled machines are about 0.02%, we focus only on the precision of the S3E models against the M57 machines. Table 6 lists the top 20 S3E models based on precision on M57 machines. All the top models in Table 6 are of the PV-DM type. All three threshold values are among the top models, but the small threshold has a more prominent presence. Also, all four

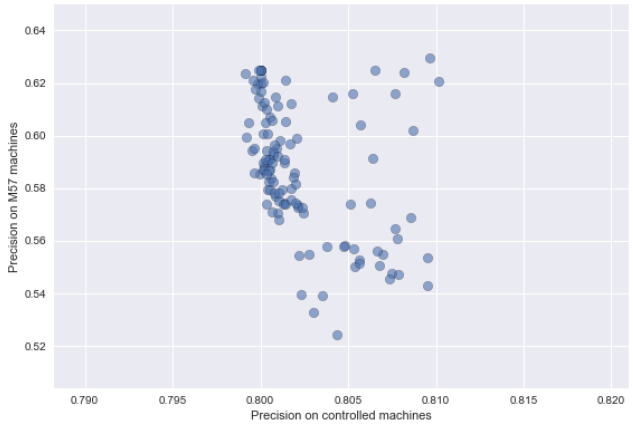

FIGURE 11. The precision rate of S3E models in both datasets.

TABLE 6. Top 20 S3E models based on precision on M57 machines.

	Name	Prec. on M57 machines		Name	Prec. on M57 machines
1	P1-V3-W5-T1	0.629	11	P1-V3-W1-T2	0.621
2	P1-V1-W1-T3	0.625	12	P1-V4-W3-T3	0.621
3	P1-V3-W1-T3	0.625	13	P1-V4-W5-T1	0.621
4	P1-V3-W2-T3	0.625	14	P1-V2-W3-T3	0.62
5	P1-V4-W1-T3	0.625	15	P1-V1-W1-T2	0.62
6	P1-V4-W2-T3	0.625	16	P1-V1-W2-T3	0.62
7	P1-V3-W4-T1	0.625	17	P1-V2-W2-T3	0.618
8	P1-V4-W4-T1	0.624	18	P1-V1-W3-T3	0.617
9	P1-V4-W1-T2	0.624	19	P1-V3-W2-T1	0.616
10	P1-V2-W1-T3	0.623	20	P1-V3-W3-T1	0.616

vector sizes and all five window sizes are among the top models. However, smaller window sizes are more prominent. As can be seen, 14 of these models are similar to the models in Table 5.

Since achieving at or near 100% recall rates is often a requirement in the digital forensics context [53], [54], we consider the 15 common top models in Table 4 and Table 5 as the best models. Table 7 lists these top models. In short, all the top models are PV-DM. Also, most top models have small thresholds and smaller window sizes.

TABLE 7. The top S3E models.

	Name		Name		Name
1	P1-V1-W1-T3	6	P1-V1-W1-T2	11	P1-V3-W1-T2
2	P1-V2-W1-T3	7	P1-V1-W3-T3	12	P1-V1-W2-T3
3	P1-V3-W1-T3	8	P1-V3-W2-T3	13	P1-V1-W5-T3
4	P1-V4-W1-T3	9	P1-V2-W2-T3	14	P1-V4-W1-T2
5	P1-V2-W1-T2	10	P1-V4-W2-T3	15	P1-V4-W2-T2

The source code of our search engines, along with the relevant data are available at https://github.com/Somayeh-Soltani/Software-Signature-Search-Engine.

4) PERFORMANCE COMPARISON OF VECTOR REPRESENTATIONS

As we described, our S3E models use the doc2vec model, which is a shallow neural network method. However, there are some other methods for the numerical representation of textual data, such as frequency-based methods, similarity hashes, and other neural network methods. In the following, we compare the performance of the doc2vec model with two other methods: the averaged word2vec model [45] and the TF-IDF method [42]. A comprehensive comparison of vector representations for software signature detection is left to future work.

As mentioned, in our S3E models, we used the doc2vec method to build signature vectors. Another solution is to use the word2vec method to get the word vectors inside each signature and then average them together. We design some software signature search engines using the averaged word2vec model and call them A_S3Es. To design A_S3Es, we only need to change the vector construction component of the signature detection subsystem.

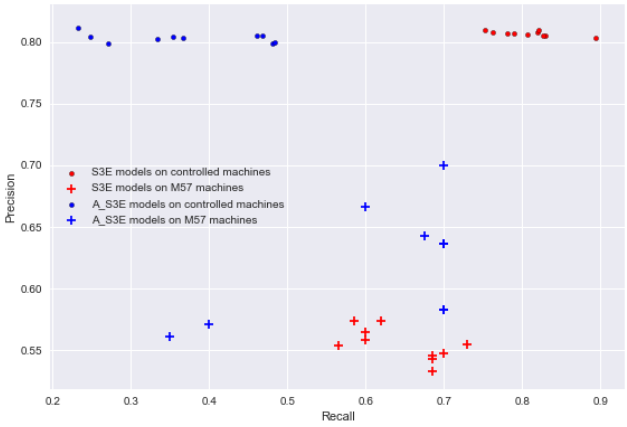

FIGURE 12. The precision and recall of S3E and A_S3E models in both datasets.

We consider five CBOW models and five skip-gram models, with a vector size of 20, the Big threshold, and five window sizes of 2, 4, 6, 8, and 10. We compare these ten A_S3E models with ten equivalent S3E models. Figure 12 shows the precision and recall of these models against both datasets, and Table 8 shows the average precision and recall of these models. On controlled machines, while the average precision of S3E models and A_S3E models is almost equal, the average recall of S3E models is much higher than A_S3E models. On M57 machines, the average recall of S3E models is slightly higher than A_S3E models, and the average precision of S3E models is less than A_S3E models. As stated, we pay more attention to the high recall rate in event reconstruction, and therefore we can say that S3E models performed slightly better than A_S3E models.

Table 8 also shows the average training time and query time for both models. While the average training time of S3E models is longer than A_S3E models, the search time of S3E models on both datasets is less than A_S3E models. Since the model is trained only once and then queried many times, we can say that the S3E models perform faster than the A_S3E models.

TABLE 8. The average precision, recall, train time, and search time of S3E and A_S3E models in both datasets.

Name	Avg. train time	Controlled machines			M57 machines		
		Avg. recall	Avg. prec.	Avg. search time	Avg. recall	Avg. prec.	Avg. search time
S3E	4.564	0.808	0.807	1.124	0.646	0.555	1.116
A_S3E	1.88	0.37	0.803	7.3	0.622	0.621	5.05

Moreover, we design some software signature search engines using the TF-IDF model and call them F_S3Es. Generally, the TF-IDF for a term t in a document d is computed as:

$$TF\_IDF\ (t, d) = TF\ (t, d) \times IDF\ (t)\,. \tag{16}$$

There are various ways of determining the TF factor. The simplest form of TF is defined in (17).

$$TF\ (t, d) = f\ (t, d)\,, \tag{17}$$

where $f(t, d)$ shows the frequency of t in d. Another formula for TF is a logarithmic one [55], which is represented by (18).

$$TF\ (t, d) = 1 + \log\left(f\ (t, d)\right). \tag{18}$$

The IDF factor also has various possibilities [55], one of them is represented by (19).

$$IDF\ (t) = \log\left(\frac{N}{df\,(t)}\right) + 1, \tag{19}$$

where N is the number of documents in the collection, and $df\,(t)$ is the document frequency of t, which is the number of documents that contain the term t. Another variation of the IDF formula, called smooth IDF, adds the constant 1 to the numerator and denominator of the IDF, as defined by (20).

$$IDF\ (t) = \log\left(\frac{N+1}{df\ (t)+1}\right) + 1. \tag{20}$$

We consider four F_S3E models. The first model uses the simple TF in (17) and the IDF in (19). The second one uses the TF in (17) and the smooth IDF in (20). The third one uses the logarithmic TF in (18) and the IDF in (19). Finally, the fourth one uses the logarithmic TF in (18) and the smooth IDF in (20). We perform similar experiments on these F_S3E models and compare their performance with S3E models. Table 9 shows the average precision and recall of S3E and F_S3E models on both datasets.

TABLE 9. The average precision and recall of S3E and F_S3E models in both datasets.

Name	Controlled machines		M57 machines	
	Avg. recall	Avg. prec.	Avg. recall	Avg. prec.
S3E	0.888	0.802	0.818	0.586
F_S3E	0.53	0.813	0.613	0.782

We can see that the average precision of the S3E and F_S3E models on the controlled machines is almost equal. The average precision of the S3E models is lower than that of the F_S3E on M57 machines. However, S3E models have a better average recall than F_S3E models in both datasets, and therefore, we can conclude that S3E models are better at detecting software on the system.

VI. CONCLUSION AND FUTURE WORK

In this paper, we proposed a method to identify the software executed on the target system. This method is a digital forensic triage solution, which narrows down the inspection scope. It consists of two subsystems: software signature construction and software signature search engine. We proposed a forensic differential analysis model to build the software signature. Besides, we used the idea of semantic search engines to build our software signature search engines. We used paragraph vector models to calculate the corresponding vectors of software signatures and queries. We designed 120 different S3E models and measured their performance against some controlled and pseudo-real datasets. We also determined the parameter values that lead to better performance models. For both datasets, PV-DM led to better models in terms of both precision and recall by spending more time and allocating more storage space. Also, the small threshold worked well for software signature search engines.

In this paper, we used the doc2vec model to represent the software signatures and showed its superiority over TF-IDF and averaged word2vec models. However, there are other word or sentence embeddings. In future work, we can design software signature search engines using different embeddings and compare the performance of them.

Creating the software signature is a time-consuming process because we need to take a copy of the disk before and after running each software scenario. So, we use a limited number of software signatures. As future work, we should create a large database of software signatures. This database should include multiple versions of various software on common operating systems.

In this work, we considered the signature of any software as an input document. Therefore, the number of input documents is relatively small, and the number of words in each document is large. In later works, we can consider different granularities for paragraph vectors. For instance, we can consider file paths inside the signatures as paragraphs. This way, the number of input documents will be extensive, and the length of each document will be short.

While we used the Windows file system metadata to build the software signature, there are no restrictions on using this method on other operating systems and other metadata. In this work, we used file paths to create the software signature. However, other file system metadata can also be considered. For instance, incorporating timestamps in creating the

software signature highlights the chronological order of the files created or modified.

REFERENCES

[1] *The RCFL Program's Annual Report for Fiscal Year 2015*. Accessed: 2015. [Online]. Available: https://www.rcfl.gov/file-repository/rcfl-annual-2015-160817-sc.pdf/view

[2] D. Quick and K.-K. R. Choo, "Impacts of increasing volume of digital forensic data: A survey and future research challenges," *Digit. Invest.*, vol. 11, no. 4, pp. 273–294, 2014.

[3] P. Joseph and J. Norman, "Forensic corpus data reduction techniques for faster analysis by eliminating tedious files," *Inf. Secur. J. A, Global Perspective*, vol. 28, nos. 4–5, pp. 136–147, 2019.

[4] H. Studiawan, F. Sohel, and C. Payne, "Sentiment analysis in a forensic timeline with deep learning," *IEEE Access*, vol. 8, pp. 60664–60675, 2020.

[5] P. H. Rughani, "Artificial intelligence based digital forensics framework," *Int. J. Adv. Res. Comput. Sci.*, vol. 8, no. 8, pp. 10–14, 2017.

[6] N. A. Adderley, "Graph-based temporal analysis in digital forensics," Air Force Inst. Technol., Wright-Patterson Air Force Base, OH, USA, Tech. Rep. AD1073875, 2019.

[7] J. S. Okolica, "Temporal event abstraction and reconstruction," Air Force Inst. Technol., Wright-Patterson Air Force Base, OH, USA, Tech. Rep. AD1055588, 2017.

[8] J. Kang, S. Lee, and H. Lee, "A digital forensic framework for automated user activity reconstruction," in *Proc. Int. Conf. Inf. Secur. Pract. Exper.* Berlin, Germany: Springer, 2013, pp. 263–277.

[9] V. Roussev and C. Quates, "Content triage with similarity digests: The M57 case study," *Digit. Invest.*, vol. 9, pp. S60–S68, Aug. 2012.

[10] J. Jones, T. Khan, K. Laskey, A. Nelson, M. Laamanen, and D. White, "Inferring previously uninstalled applications from residual partial artifacts," in *Proc. ADFSL*, 2017, pp. 113–130.

[11] S. Mead, "Unique file identification in the national software reference library," *Digit. Invest.*, vol. 3, no. 3, pp. 138–150, 2006.

[12] Y. Chabot, A. Bertaux, C. Nicolle, and T. Kechadi, "An ontology-based approach for the reconstruction and analysis of digital incidents timelines," *Digit. Invest.*, vol. 15, pp. 83–100, Dec. 2015.

[13] A. J. Nelson, "Software signature derivation from sequential digital forensic analysis," Ph.D. dissertation, Dept. Comput. Sci., Univ. California Santa Cruz, Santa Cruz, CA, USA, 2016.

[14] J. I. James, P. Gladyshev, and Y. Zhu, "Signature based detection of user events for post-mortem forensic analysis," in *Proc. Int. Conf. Digit. Forensics Cyber Crime*. Berlin, Germany: Springer, 2010, pp. 96–109.

[15] C. Hargreaves and J. Patterson, "An automated timeline reconstruction approach for digital forensic investigations," *Digit. Invest.*, vol. 9, pp. S69–S79, Aug. 2012.

[16] M. N. A. Khan, "Performance analysis of Bayesian networks and neural networks in classification of file system activities," *Comput. Secur.*, vol. 31, no. 4, pp. 391–401, 2012.

[17] S. Kälber, A. Dewald, and F. C. Freiling, "Forensic application-fingerprinting based on file system metadata," in *Proc. 7th Int. Conf. IT Secur. Incident Manage. IT Forensics*, Mar. 2013, pp. 98–112.

[18] S. Soltani, S. A. H. Seno, and H. S. Yazdi, "Event reconstruction using temporal pattern of file system modification," *IET Inf. Secur.*, vol. 13, no. 3, pp. 201–212, 2019.

[19] M. Khader, A. Hadi, and G. Al-Naymat, "HDFS file operation fingerprints for forensic investigations," *Digit. Invest.*, vol. 24, pp. 50–61, Mar. 2018.

[20] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196.

[21] K. Woods, C. A. Lee, S. Garfinkel, D. Dittrich, A. Russell, and K. Kearton, "Creating realistic corpora for security and forensic education," Dept. Comput. Sci., Nav. Postgraduate School, Monterey, CA, USA, 2011. [Online]. Available: https://calhoun.nps.edu/handle/10945/41311

[22] N. Kanome, "Computer system capable of restarting system using disk image of arbitrary snapshot," Google Patents 6 205 450, Mar. 20, 2001.

[23] R. M. Jenevein, H. S. Kramer, D. S. Shadel, A. V. Lawrence, and V. A. Arbon, "Storing a computer disk image within an imaged partition," Google Patents 6 615 365, Sep. 2, 2003.

[24] B. D. Carrier and J. Grand, "A hardware-based memory acquisition procedure for digital investigations," *Digit. Invest.*, vol. 1, no. 1, pp. 50–60, 2004.

[25] L. Wang, "Providing access to physical memory allocated to a process by selectively mapping pages of the physical memory with virtual memory allocated to the process," Google Patents 6 477 612, Nov. 5, 2002.

[26] M. Cohen, "PyFlag—An advanced network forensic framework," *Digit. Invest.*, vol. 5, pp. S112–S120, Sep. 2008.

[27] F. Carbone, *Computer Forensics With FTK*. Birmingham, U.K: Packt, 2014.

[28] S. Widup, *Computer Forensics and Digital Investigation With EnCase Forensic V7*. New York, NY, USA: McGraw-Hill, 2014.

[29] NIST. *National Software Reference Library (NSRL)*. Accessed: Apr. 8, 2021. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/national-software-reference-library-nsrl

[30] X. Lin, "File signature searching forensics," in *Introductory Computer Forensics*. Cham, Switzerland: Springer, 2018, pp. 235–244.

[31] N. C. Rowe, "Identifying forensically uninteresting files in a large corpus," *ICST Trans. Secur. Saf.*, vol. 3, no. 7, Dec. 2016, Art. no. 151725.

[32] F. P. Buchholz and C. Falk, "Design and implementation of Zeitline: A forensic timeline editor," in *Proc. DFRWS*, 2005, pp. 1–7.

[33] J. Olsson and M. Boldt, "Computer forensic timeline visualization tool," *Digit. Invest.*, vol. 6, pp. S78–S87, Sep. 2009.

[34] K. Guðjónsson, "Mastering the super timeline with log2timeline," SANS Inst., Bethesda, MD, USA, Tech. Rep., 2010.

[35] J. Metz and K. Guðjónsson. *Plaso—Super Timeline all the Things*. Accessed: Apr. 8, 2021. [Online]. Available: https://github.com/log2timeline/plaso

[36] M. Debinski, F. Breitinger, and P. Mohan, "Timeline2GUI: A Log2Timeline CSV parser and training scenarios," *Digit. Invest.*, vol. 28, pp. 34–43, Mar. 2019.

[37] V. Roussev, "Data fingerprinting with similarity digests," in *Proc. IFIP Int. Conf. Digit. Forensics*. Berlin, Germany: Springer, 2010, pp. 207–226.

[38] NIST. *Diskprints*. Accessed: Apr. 8, 2021. [Online]. Available: https://www.nist.gov/itl/ssd/software-quality-group/national-software-reference-library-nsrl/nsrl-subprojects/diskprints

[39] D. Jeong and S. Lee, "Forensic signature for tracking storage devices: Analysis of UEFI firmware image, disk signature and windows artifacts," *Digit. Invest.*, vol. 29, pp. 21–27, Jun. 2019.

[40] K. Park, J.-M. Park, E.-J. Kim, C. Cheon, and J. James, "Anti-forensic trace detection in digital forensic triage investigations," *J. Digit. Forensics, Secur. Law*, vol. 12, no. 1, p. 8, 2017.

[41] T. Teofili, "Deep learning for search," in *Manning Early Access Program*. Shelter Island, NY, USA: Manning Publications, 2018.

[42] A. Rajaraman and J. D. Ullman, *Mining of Massive Datasets*. Cambridge, U.K.: Cambridge Univ. Press, 2011.

[43] F. Lashkari, E. Bagheri, and A. A. Ghorbani, "Neural embedding-based indices for semantic search," *Inf. Process. Manage.*, vol. 56, no. 3, pp. 733–755, 2019.

[44] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," 2013, *arXiv:1301.3781*. [Online]. Available: http://arxiv.org/abs/1301.3781

[45] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.

[46] S. L. Garfinkel, "Automating disk forensic processing with SleuthKit, XML and Python," in *Proc. 4th Int. IEEE Workshop Systematic Approaches Digit. Forensic Eng.*, May 2009, pp. 73–84.

[47] S. Garfinkel, "Digital forensics XML and the DFXML toolset," *Digit. Invest.*, vol. 8, nos. 3–4, pp. 161–174, 2012.

[48] L. Van der Maaten and G. Hinton, "Visualizing data using t-SNE," *J. Mach. Learn. Res.*, vol. 9, no. 11, pp. 2579–2605, 2008.

[49] B. S. Kumar and J. Prakash, "Precision and relative recall of search engines: A comparative study of Google and Yahoo," *Singap. J. Library Inf. Manage.*, vol. 38, no. 1, pp. 124–137, 2009.

[50] D. Tümer, M. A. Shah, and Y. Bitirim, "An empirical evaluation on semantic search performance of keyword-based and semantic search engines: Google, Yahoo, Msn and Hakia," in *Proc. 4th Int. Conf. Internet Monitor. Protection*, May 2009, pp. 51–55.

[51] N. L. Beebe and L. Liu, "Clustering digital forensic string search output," *Digit. Invest.*, vol. 11, no. 4, pp. 314–322, Dec. 2014.

[52] K. Kröger and R. Creutzburg, "A practical overview and comparison of certain commercial forensic software tools for processing large-scale digital investigations," in *Mobile Multimedia/Image Processing, Security, and Applications*, vol. 8755. Bellingham, WA, USA: International Society for Optics and Photonics, 2013, Art. no. 875519.

[53] N. L. Beebe, J. G. Clark, G. B. Dietrich, M. S. Ko, and D. Ko, "Post-retrieval search hit clustering to improve information retrieval effectiveness: Two digital forensics case studies," *Decis. Support Syst.*, vol. 51, no. 4, pp. 732–744, 2011.

[54] V. K. Kota, "An ontological approach for digital evidence search," *Int. J. Sci. Res. Publications*, vol. 2, no. 12, pp. 409–414, 2012.
[55] A. Aizawa, "An information-theoretic perspective of tf–idf measures," *Inf. Process. Manage.*, vol. 39, no. 1, pp. 45–65, 2003.

SOMAYEH SOLTANI received the B.S. degree in computer science from the Ferdowsi University of Mashhad, Iran, in 2006, and the M.Sc. degree in computer science from Islamic Azad University, Science and Research Branch, Tehran, Iran, in 2010. She is currently pursuing the Ph.D. degree with the Ferdowsi University of Mashhad. Her research interests include digital forensics, malware detection, and formal verification methods.

SEYED AMIN HOSSEINI SENO received the B.Sc. and M.Sc. degrees in computer engineering from the Ferdowsi University of Mashhad, Mashhad, Iran, in 1990 and 1998, respectively, and the Ph.D. degree from Universiti Sains Malaysia, Malaysia, in 2010. He is currently an Associate Professor with the Department of Computer Engineering, Ferdowsi University of Mashhad. His research interests include computer networks, QoS, the IoT, and network security.

RAHMAT BUDIARTO received the B.Sc. degree in mathematics from the Bandung Institute of Technology, Indonesia, in 1986, and the M.Eng. and Dr.Eng. degrees in computer science from the Nagoya Institute of Technology, Japan, in 1995 and 1998, respectively. He is currently a Full Professor with the Department of Informatics, Universitas Al-Azhar Indonesia. His research interests include intelligent systems, brain modeling, IPv6, network security, wireless sensor networks, and MANETs.