Forensic Science International: Digital Investigation 44 (2023) 301484

Contents lists available at ScienceDirect

Forensic Science International: Digital Investigation

journal homepage: www.elsevier.com/locate/fsidi

Detecting the software usage on a compromised system: A triage solution for digital forensics

Somayeh Soltani*, Seyed Amin Hosseini Seno

Department of Computer Engineering, Ferdowsi University of Mashhad, Mashhad, Iran

A R T I C L E I N F O

Article history:
Received 20 June 2022
Received in revised form
9 November 2022
Accepted 20 November 2022
Available online xxx

Keywords:
Digital forensics
Triage process
Software signature
TF-IDF
Forensic differential analysis

A B S T R A C T

One of the challenges of digital forensics is the high volume of investigative cases. To address this problem, researchers have proposed various triage methods. Detecting the applications that have run on the compromised system under inspection can be an excellent triage method that gives the investigator an overview of the system. In this paper, we construct the signature of software usage on a system using file path artifacts. We propose a software signature detection engine (SSDE) to identify the usage of the software on the system under investigation. The SSDE consists of two subsystems: the signature construction subsystem, which builds the software signature using the TF-IDF weighting scheme, and the signature detection subsystem, which identifies the executed set of software on the target system. We consider several parameters with different values in the design of SSDEs, leading to more than 500 SSDE models. We test the SSDE models against 14 pseudo-real systems from the M57 Patents scenario and evaluate their performance. The experimental results show that about 38% of SSDE models achieve near-perfect Precision, and about 18% of them achieve near-perfect Recall. We introduce the top models and determine which parameter values lead to the superior models. Besides, we compare the SSDE models with some doc2vec-based models. The results show that SSDE models have higher average Precision, slightly lower average Recall, and much less computational time.

© 2022 Elsevier Ltd. All rights reserved.

1. Introduction

The rapid growth of the Internet and cyberspace has led to the growth of malicious activities. Digital forensics helps discover the events that have occurred on a compromised system. It comprises three main phases, namely acquisition, analysis, and presentation. The challenges facing digital forensic analysis are the large volume of data collected and the lack of automated analysis methods. In some cases, the analysis of the collected data may take days to weeks (Scanlon, 2016; Casey and Zehnder, 2020; Lillis et al., 2016; Du et al., 2020; Vidas et al., 2014).

Current digital forensic tools show a list of extracted artifacts from digital media (Bunting, 2012; Sammons, 2016; Ghazinour et al., 2017). However, they do not reveal the events that cause the creation of the artifacts. Usually, the investigator manually tries to find this relationship (Du et al., 2020; Shaw and Browne, 2013; Studiawan et al., 2020). Manually reconstructing the events using the artifacts creates some problems. First, the investigator can only retrieve the events that he knows. Therefore, the retrieval rate is limited to the investigator's experience. Second, manual event reconstruction is quite time-consuming, and finally, human errors may occur.

Several triage solutions have been proposed in the literature. Triage in digital forensics deals with prioritizing some pieces of evidence to start the forensic analysis quickly. In particular, some digital forensic triage methods identify devices containing valuable forensic information among the many devices found at the crime scene (Jusas et al., 2017; Gentry et al., 2019; Lim and Jones, 2020). Besides, the triage methods for multimedia forensics speed up image/video analysis (Quick and Choo, 2017; Hales and Bayne, 2019). Moreover, some triage methods detect the set of software executed on the system as a preprocessing step. Identifying the running software on the system during the incident presents an overview of the system (Nelson, 2016; Jones et al., 2016; Adegbehingbe and Jones, 2019; James and Gladyshev, 2015; Al-Sharif et al., 2019). It can also provide the examiner with a clue to the types of data that can be detected.

Running an application leaves its traces on the system. We can process these traces to build the signature of the software. The digital forensic investigator could use these signatures to determine what applications have run on the examined system. Several research efforts have built signatures for various events or applications (Nelson, 2016; Jones et al., 2016; Adegbehingbe and Jones,

* Corresponding author.
E-mail address: somayeh.soltani@mail.um.ac.ir (S. Soltani).

https://doi.org/10.1016/j.fsidi.2022.301484
2666-2817/© 2022 Elsevier Ltd. All rights reserved.

2019; James and Gladyshev, 2015; Soltani et al., 2019; Latzo, 2020; Hargreaves and Patterson, 2012; Kälber et al., 2013; Roussev and Quates, 2012; Khader et al., 2018; Jeong and Lee, 2019; Mistry and Dahiya, 2019; Song et al., 2018; Chang et al., 2019). Some of these methods use a few limited items to create a software signature (James and Gladyshev, 2015; Hargreaves and Patterson, 2012; Kälber et al., 2013). They also use a precise matching approach to check the signatures against an examined system. Therefore, they may easily be subverted by adversaries. Moreover, building signatures are not automated in some methods (Hargreaves and Patterson, 2012; Khader et al., 2018; Jeong and Lee, 2019). Also, some methods which use similarity digests are computationally time-consuming (Roussev and Quates, 2012; Chang et al., 2019; Moia and Henriques, 2017). For example, Roussev and Quates (2012) construct the hash for all executable files of the hard disk and the hash of the memory image. Finally, memory-based event reconstruction methods (Mistry and Dahiya, 2019; Song et al., 2018; Mohanta and Saldanha, 2020) are not reliable enough, as the memory content changes over time and clears totally when the system shuts down.

In this study, we present a software signature detection engine (SSDE) that consists of two subsystems. The signature construction subsystem builds the software signature using the software execution tracks on the file system. The signature detection subsystem takes the created signature database and the disk copy of the system under investigation as input and determines what software has been run on it. We have considered various design parameters with different values to build the software signature detection engine. Selecting different values leads to a variety of SSDE models. To evaluate these models and select the top ones, we have given 14 quasi-real systems from the M57 Patents scenario as the target system for each model.[1]

In our previous study (Soltani et al., 2021), we built software signature search engines (S3Es) for digital forensics purposes. In that paper, we used the word embedding model to construct the software signature vector, and by selecting different values for the design parameters, we designed 120 separate S3E models. In this work, we use the TF-IDF frequency-based weighting method to construct the software signature vectors. TF-IDF, short for term frequency-inverse document frequency, is a statistical measure that reflects how important a term (word) is to a document in a corpus (Rajaraman and Ullman, 2011). In TF-IDF, term frequency is the number of times a word occurs in a document, and IDF is inversely proportional to the number of times a word appears in the corpus. The IDF tends to reduce the weight of very frequent words in the corpus and increase the weight of rare words. An advantage of word embedding methods over frequency-based methods is that they care about the context and meaning of words. However, the downside of word embedding methods against frequency-based ones is that they are more complex and time-consuming. Section 4.3 will compare the SSDE models with the S3E models regarding Precision and Recall rates and time costs.

This research work has various contributions in the domain of digital forensics, as follows.

- Software signature detection engines that assist the digital investigator in detecting software usage on the target system have been designed.
- More than 500 SSDE models have been designed by considering eight design parameters.
- The superior models and the appropriate values of the design parameters are determined.
- Finally, the TF-IDF-based SSDE models are compared with doc2vec-based S3E models (Soltani et al., 2021).

The rest of the paper is organized as follows. Section 2 reviews some related works in digital forensic triage. Section 3 explains the proposed software signature detection engine. Section 4 describes the experiments and results, and finally, Section 5 concludes the paper and gives some recommendations for future work.

2. Related work

There are several digital forensic triage methods in the literature. Some methods visualize a timeline of events, and some find correlations among pieces of digital evidence. Also, several methods find the signatures of various actions or applications.

2.1. Digital forensics timeline

CyberForensics TimeLab (CFTL) (Olsson and Boldt, 2009) shows timing information of file system, Registry, and link files. Log2timeline (Guðjónsson, 2010; Metz and Guðjónsson, 2021) displays a timeline of timing information extracted from different file types; timing information from various parts of the hard disk such as file system, Registry, log files, prefetch files, browser history, and system memory are displayed chronologically. Although log2timeline itself cannot reconstruct high-level events, it is used in several research projects (Chabot et al., 2015; Du and Scanlon, 2019; Bhandari and Jusas, 2020; Good and Peterson, 2017). Timeline2GUI (Debinski et al., 2019) is a graphical interface that reads CSV files generated by log2timeline and performs several operations, including filtering, sorting, searching, and highlighting text. Timesketch (Google, 2021) is a timeline tool that helps investigators analyze event logs using tabular and graph-based views.

While a timeline of extracted artifacts provides an investigator with a visual view, it cannot assist in the automatic reconstruction of events. However, having a timeline can be helpful in addition to a signature-based method (such as our proposed method). For example, it can very quickly specify the time interval in which to search for software signature traces.

2.2. Ontology-based event analysis

Chabot et al. (2015) introduced an approach based on a three-layered ontology called ORD2I. They used log2timeline to populate the ontology. The Common Knowledge layer stores shared information about what happened during the incident. This layer contains a public Entity class and three derived classes, Event, Subject, and Object. The Specialized Knowledge layer has classes for many objects such as File, Account, Web, Communication, and Registry Key. Finally, the Traceability Knowledge layer stores information about the investigation process.

Arshad et al. (2019) provided a multi-layered framework for collecting and correlating evidence from various social media. The central component of the framework is a hybrid ontology. Karie and Kebande (2016) provided an ontology for digital forensics to help investigators understand various technical and non-technical terminologies. Their approach has four components: 1) digital forensic terminology database, 2) terminology semantic annotation, 3) reasoning engine, and 4) terminology semantic repository. Bhandari and Jusas (2020) proposed an ontological approach based on the abstraction concept to analyze the timeline of artifacts. The abstraction concept helps investigators resolve the meaning of new digital forensic terminologies.

[1] The M57 Patents data set is freely available at https://digitalcorpora.org/corpora/scenarios/m57-patents-scenario/.

We should mention here that ontology-based methods cannot automatically reconstruct events. These methods help the investigator find correlations between low-level pieces of evidence. An ontology-based technique alongside a signature-based one can help the inspector reconstruct events.

2.3. Creating and detecting signatures for applications or actions

Several research studies built the signature of various actions and applications. Using these signatures, they estimated what events had occurred on the system. To build application fingerprints, Kalber et al. (Kälber et al., 2013) used MACB timestamps of filesystem metadata. To determine the actions performed on the system, they used an exact matching paradigm. James and Gladyshev (2015) built the signature of user actions using the filesystem timestamps. They can detect the most recent as well as past action instance approximations. The weakness of these two methods is that they used a few modified timestamps as the fingerprint of actions, which can easily be subverted.

Hargreaves et al. (Hargreaves and Patterson, 2012) created a timeline of low-level events, including filesystem timestamps and timing information inside some complex files such as Chrome history, Skype, Link files, and Registry. They also designed specific analyzers for high-level events such as Windows installation, Google search, Skype call, and USB connection. The analyzers searched for patterns of high-level events in the low-level timeline based on some pre-determined rules. Making specific rules for each event is hard, and this method cannot be easily generalized to include analyzers for other high-level events.

To build signatures for different applications, Khan (2012) proposed a neural network and a Bayesian belief network. However, a constraint of this methodology is that for each application, a separate network needs to be trained. Khader et al. (2018) attempted to find fingerprints for the Hadoop Distributed File System (HDFS) operations, including creation, append, rename, and deletion. However, they did not reconstruct high-level events on the distributed file system.

Roussev and Quates (2012) tried to specify the applications that run on M57 computers (Woods et al., 2011) using a fuzzy hashing scheme (Roussev, 2010). They first built the hash of all executables of the disk images and then compared the memory image against the sdbf hashes of the. exe or. dll files on the disk image. This method is time-consuming and needs both hard disk image and memory image.

To identify uninstalled applications on a system, Adegbehingbe and Jones (2019) built a catalog for each application using the application's Diskprint (NIST, 2021). The application's Diskprint includes snapshots of a virtual machine while installing, running, and uninstalling the application. Adegbehingbe and Jones extracted the created and modified files between snapshots. The application's catalog includes the name of these files and the sectors' hashes containing the files.

Nelson (2016) provided a forensic search engine in which the input documents are the signatures of the software, and the query is the target disk Registry. Software signatures are obtained by processing the differences between Registry snapshots of the system. He used NIST's Diskprint project (NIST, 2021) in his work.

Jeong and Lee (2019) identified storage devices connected to a computer system. They built the connection signature of a storage device using the fingerprints found in the Registry, the main boot record (MBR), logs (Operational.evtx and diagnostic. evtx files), and NVAR variables.

M57 Patents (Woods et al., 2011) is a digital forensic training scenario developed by the US Naval Postgraduate School. This scenario captures the activities of the first four weeks of the fictitious m57. biz patents research company. At the end of each workday, the computers of four employees (named Charlie, Jo, Pat, and Terry) are captured, and their physical memory and hard disks are imaged. In this paper, we will use 14 disk images of M57 Patents computers as our test systems.

3. The software signature detection engine

In this paper, to detect software on a system, we design a software signature detection engine. The SSDE consists of two subsystems: 1) signature construction and 2) signature detection. Creating the signature for each software is done using the traces left by running the software on the file system. In the signature detection phase, a target file system is examined to find traces of various software. Fig. 1 shows an overall view of an SSDE.

3.1. Software signature construction

As mentioned, we create the signature of each software using the effect of the software on the underlying file system. To build the software signature, we compare the disk copies before and after running the software to get a list of files and folders created, deleted, or changed during the software run. However, we should note that we build software signatures using only the created files and folders. The reason behind this decision is that we found in our experiments that most deleted or modified files were not related to the software execution but were the results of OS execution.

For example, to create a signature for Microsoft Word 2013, we compared disk copies before and after running it. In the output list, we have 485 created files and folders, 148 modified files and folders, and 542 deleted files and folders. However, most modified files and folders were not related to Microsoft Word. For example, some irrelevant files and folders that are modified include $LogFile, $MFT, $Bitmap, $Secure, ProgramData/Microsoft/Diagnosis/ETLLogs/AutoLogger, ProgramData/Microsoft/RAC/PublishedData, and Users/Somayeh/AppData/Local/Microsoft/Windows/Explorer/thumbcache_idx.db. Only 34 of the changed files and folders were related to running Microsoft Word, some of which include: Program Files/Microsoft Office/Office14/WORDIRMV.XML, Program Files/Microsoft Office/Office14/WINWORD.EXE, ProgramData/Microsoft Help/MS. WINWORD.DEV.14.1033. hxn, and Windows/assembly/tmp/5D1YG5NO/Microsoft.Office.Interop.Word.dll. The same is true for deleted files and folders.

Suppose we want to create a signature for software *SW* on operating system *O*. To do this, we first install the OS *O* on a virtual machine, then we install the software *SW* on this recently installed OS, and we convert the hard disk of this virtual machine to E01 forensic disk image format. Next, we run the software *SW* and take the E01 hard disk. Then we use Fiwalk[2] (Garfinkel, 2009) to get the list of files and folders on each disk copy. Next, we compare two Fiwalk outputs to identify files and folders affected by the software run. Then we filter this list and remove the deleted and modified files and folders. Finally, we consider the created files and folders during the software execution as the Difference-Set of the software run. Fig. 2 describes the process of creating a Difference-Set for software execution.

Simplistically, we can consider this Difference-Set as the software signature. However, different scenarios for running an application may cause partly different Difference-Sets. A piece of software can be executed in different ways. For example, it can be launched by clicking on its icon on the desktop or by using the Start

[2] Fiwalk processes a disk image and produces an XML file, which includes information such as file path, file size, MD5 and SHA1 digests, and MACB timestamps.

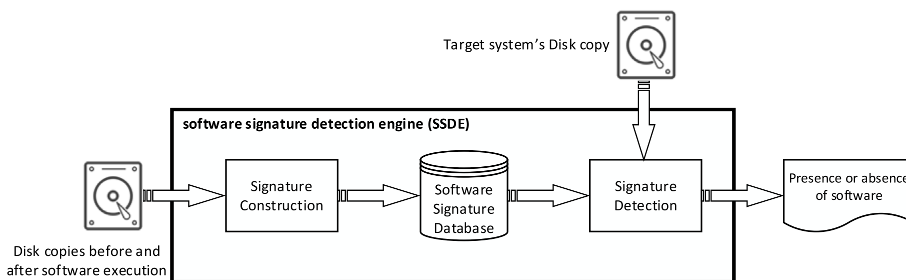

Fig. 1. An overall view of the SSDE.

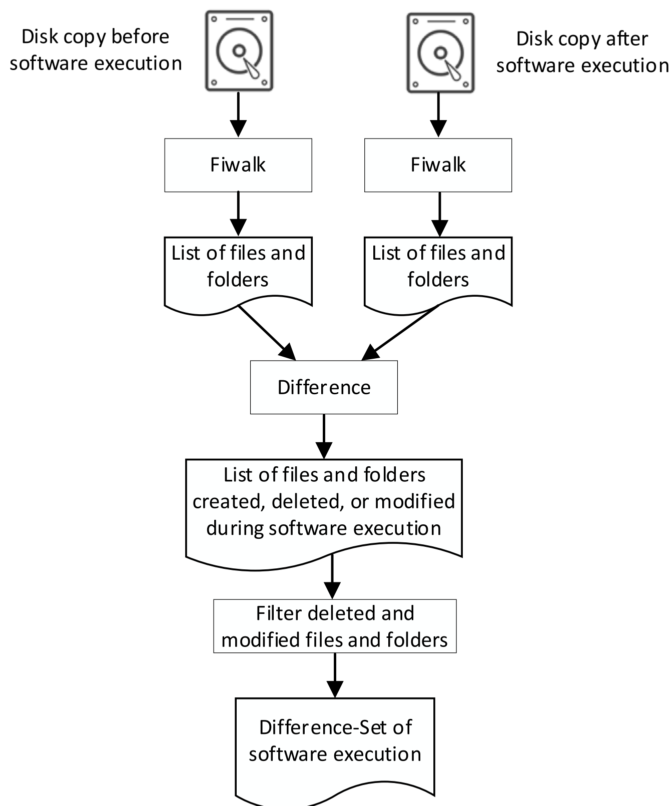

Fig. 2. The process of creating the Difference-Set for the software.

menu. After the software is opened, depending on its type, the user can have different interactions with it. For example, a user can open Firefox by clicking its icon on the desktop, taskbar, or Start menu. The user can then do various things with Firefox: visit a website and click multiple links, check his email and download an attachment, login to an online retail website, and so on. As another example, a user can open Microsoft Word by clicking its icon on the desktop, taskbar, or Start menu. Also, he can double-click a. docx file or right-click a window and select New → Microsoft Word Document. The user can do various things with Microsoft Word, and after making the desired changes on a document, save it or save it with a different name.

Each of these scenarios may cause minor changes in the list of files and folders created during software execution. As Fig. 3 shows, to build the software signature, we should consider combining various Difference-Sets obtained from different scenarios of software runs.

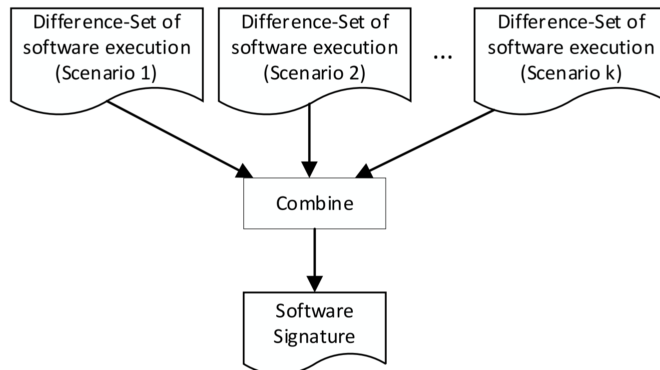

Fig. 3. Creating the software signature.

3.1.1. Design parameters of software signatures

We consider three parameters in building software signatures which are listed in Table 1. The parameters include 1) Composition of Difference-Sets, 2) n-gram types, and 3) stop list. We design two values for Difference-Set composition: Intersection (tagged with A_1) and Union (tagged with A_2). Moreover, we consider four distinct n-gram types tagged with B_1, B_2, B_3, and B_4. Also, we have two values for the stop list (C_1 and C_2). Different values of the parameters result in distinct software signature models.

To combine Difference-Sets and build signatures, we can compute either union or intersection of them. While union combination (A_2) builds software signature using all terms in all relevant Difference-Sets, intersection combination (A_1) only considers the common terms.

The second parameter determines the type of n-gram. The value B_1 of this parameter considers the entire file path as a term. For example, consider the path *Program Files/Mozilla Firefox/uninstall/shortcuts_log.ini*. B_1 considers the entire path as a term. B_2 takes four terms for this file path: *Program Files*, *Mozilla Firefox*, *uninstall*, and *shortcuts_log.ini*. B_3 considers the *shortcuts_log.ini* component as a term. Finally, B_4 introduces *uninstall/shortcuts_log.ini* as a term.

The third parameter determines whether to consider a stop list and filter the software signatures or not. The use of a stop list is inspired by natural language processing (NLP). The idea is to remove common natural language words (such as the, is, you, and for) from the text. The reason is that stop words are low-value words and do not add much meaning to a sentence. By removing them from the text, the size of the dataset is reduced, and the training time of the model is decreased. In this work, we consider the stop list equal to the files and folders in the base operating system (a newly installed OS on which no software has been installed yet).

Table 1
Design parameters of software signatures.

	Parameter	Tag	Value	Description
1	Difference-Set composition	A_1	Intersection	The Difference-Sets are combined using the intersection of their elements.
		A_2	Union	The Difference-Sets are combined using the union of their elements.
2	n-gram type	B_1	All	The full file path is used as a term.
		B_2	Unigram	The file path is broken into its components, and each component will be a term.
		B_3	Last	The last component of the file path is considered as a term.
		B_4	Two Last	The last two components of the file path constitute a term.
3	Stop list	C_1	No Stop List	There is no stop list for common terms.
		C_2	Stop List	The stop list for common terms includes the file paths in the disk copy of the base OS.

C_2 models (models that contain the C_2 value for the third parameter) define a stop list containing the file paths of the base OS. All the file paths in the stop list are removed from the software signatures. We should note that the n-gram type affects the type of stop list. It means that the stop list for B_1 models includes file paths extracted from the base OS. For B_2 models, the stop list contains all the components of these file paths. The stop list of B_3 models includes the last component of these file paths. Finally, for B_4 models, the stop list contains the last two components of the file paths of the base OS.

3.1.2. A forensic differential analysis model for building software signatures

In this section, we formally describe the software signature creation process. We propose a forensic differential analysis model to build the software signature. Suppose we have two disk copies, A (taken after software execution) and B (taken before software execution). Assume disk copies A and B have N_A and N_B files and folders, respectively. Also, suppose that each file or folder has M different forensic features. The function E extracts the features from each disk copy:

$$E : D \rightarrow F_D \tag{1}$$

where $D = \{A, B\}$, and F_D is the set of features of files and folders of disk copy D; In other terms:

$$F_D = \left\{ f_{D_{i,j}} i \in \{1, \cdots, N_D\}, j \in \{1, \cdots, M\} \right\} \tag{2}$$

where $f_{D_{i,j}}$ is the value of the j th feature of the i th file or folder of disk copy D. We consider three valuable features of files and folders, including path, modification timestamp, and hash value. The differential analysis of A and B is reduced to the differential analysis of the feature sets F_A and F_B, as defined in (3).

$$\begin{aligned} DA(F_A, F_B) = & \bigcup_{i \in \{1, \cdots, N_A\}} \left\{ (A, i, p) \neg \exists k \in \{1, \cdots, N_B\} : f_{A_{i,p}} = f_{B_{k,p}} \right\} \cup \bigcup_{i \in \{1, \cdots, N_B\}} \left\{ (B, i, p) \neg \exists k \in \{1, \cdots, N_A\} : f_{B_{i,p}} = f_{A_{k,p}} \right\} \cup \\ & \bigcup_{i \in \{1, \cdots, N_A\}} \left\{ (A, i, p) \exists k \in \{1, \cdots, N_B\} : f_{A_{i,p}} = f_{B_{k,p}} \wedge f_{A_{i,t}} \neq f_{B_{k,t}} \wedge f_{A_{i,h}} \neq f_{B_{k,h}} \right\} \cup \\ & \bigcup_{i \in \{1, \cdots, N_A\}} \left\{ (A, i, p) \exists k \in \{1, \cdots, N_B\} : f_{A_{i,p}} = f_{B_{k,p}} \wedge f_{A_{i,t}} = f_{B_{k,t}} \wedge f_{A_{i,h}} \neq f_{B_{k,h}} \right\} \cup \\ & \bigcup_{i \in \{1, \cdots, N_A\}} \left\{ (A, i, p) \exists k \in \{1, \cdots, N_B\} : f_{A_{i,p}} = f_{B_{k,p}} \wedge f_{A_{i,t}} \neq f_{B_{k,t}} \wedge f_{A_{i,h}} = f_{B_{k,h}} \right\}, \end{aligned} \tag{3}$$

where p, t, and h, are path, the modification timestamp, and the hash value, respectively.

The first part of (3) lists the created files and folders during software execution. The second part lists the paths of deleted files and folders during software execution. The next parts list the modified files during software execution. In particular, the third part lists files whose content has changed. The fourth part lists files whose content has changed, but the modification timestamp has not changed. For example, an anti-forensic technique to hide event traces does not change the modification timestamps of some files. Also, software errors or failure to save modified files when copying the disk can be the reason for this type of modification. Finally, the fifth part lists files whose modification timestamp has changed, but their content has not changed. For example, undoing the changes in a text file may cause this modification.

While the differential function in (3) lists the created, deleted, and modified files and folders, we build software signatures using only the created files and folders. Therefore, we use a simplified version of the differential function, as defined in (4).

$$SimDA(F_A, F_B) = \bigcup_{i \in \{1, \cdots, N_A\}} \left\{ (i, p) \neg \exists k \in \{1, \cdots, N_B\} : f_{A_{i,p}} = f_{B_{k,p}} \right\}. \tag{4}$$

The differential analysis function as a result of executing the software SW is defined in (5).

$$SimDA\left(F_{Pre(SW)}, F_{Post(SW)}\right) = \bigcup_{i\in\{1,\cdots,N_{Post(SW)}\}} \left\{(i,p)\neg\exists k \in \left\{1,\cdots,N_{Pre(SW)}\right\} : f_{Post(SW)_{i,p}} = f_{Pre(SW)_{k,p}}\right\}, \quad (5)$$

where $pre(SW)$ is the disk copy taken immediately before SW execution and $post(SW)$ is the disk copy taken immediately after SW execution. The simplified differential analysis function in (5) gives ordered pairs representing the files and folders created during the software execution. The Difference-Set of software execution is composed of the second element of these ordered pairs, as represented by (6).

$$DiffSet(SW) = \left\{p(i,p) \in SimDA\left(F_{Pre(SW)}, F_{Post(SW)}\right)\right\}. \quad (6)$$

We should note that the Difference-Set in (6) is used when the whole file path is considered an n-gram. As we explained, different parts of the file path can be considered n-grams. If p consists of Q components, then p can be represented in two ways. A representation in which the order of the constituent components does not matter:

$$p = \{w_i 1 \le i \le Q\} = \{w_1, w_2, \cdots, w_Q\}, \quad (7)$$

where w_i is the i th component of p.

A form in which the order of the components is important:

$$p = w_1/w_2/\cdots/w_{Q-1}/w_Q. \quad (8)$$

When n-grams are the whole file paths (B_1 models), the Difference-Set is defined as (9).

$$DiffSet(SW_{All}) = DiffSet(SW). \quad (9)$$

When n-grams are all components of file paths (B_2 models), the Difference-Set is defined as (10).

$$DiffSet(SW_{Unigram}) = \{w_i w_i \in p = \{w_1, w_2, \cdots, w_Q\}, p \in DiffSet(SW)\}. \quad (10)$$

When n-grams are the last components of file paths (B_3 models), the Difference-Set is defined in (11).

$$eqalignDiffSet(SW_{Last}) = \{w_Q w_Q \in p, p = w_1/w_2/\cdots/w_{Q-1}/w_Q, p \in DS(SW)\}. \quad (11)$$

When n-grams are the two last components of file paths (B_4 models), the Difference-Set is defined in (12).

$$DiffSet(SW_{Two\ Last}) = \{w_{Q-1}/w_Q w_{Q-1}, w_Q \in p, p = w_1/w_2/\cdots/w_{Q-1}/w_Q, p \in DS(SW)\}. \quad (12)$$

Running software with different scenarios leads to different Difference-Sets. To build the software signature, we need to combine these Difference-Sets, for which we can use union or intersection combination. If we consider k scenarios, $S_1, S_2, \cdots, S_k$, for software SW, then we need to combine k Difference-Sets. The software signature for models with intersection composition and the whole file path as n-grams (A_1B_1 models) is defined in (13).

$$SigInt(SW_{All}) = \bigcap_{i\in\{1,\cdots,k\}} DiffSet(SW_{S_{iAll}}). \quad (13)$$

Generally, the software signature for A_1 models is defined in (14).

$$SigInt(SW_{ngram}) = \bigcap_{i\in\{1,\cdots,k\}} DiffSet\left(SW_{S_{ingram}}\right), ngram \in \{All, Unigram, Last, Two\ Last\}. \quad (14)$$

Also, the software signature for A_2 models is defined in (15).

$$SigUni(SW_{ngram}) = \bigcup_{i\in\{1,\cdots,k\}} DiffSet\left(SW_{S_{ingram}}\right), ngram \in \{All, Unigram, Last, Two\ Last\}. \quad (15)$$

3.2. Software signature detection

Using the software signatures made in the previous phase, we can discover the software that was running on the target system. To do this, we must first extract the list of files and folders of the target system's disk copy (the query). We do not use exact matching, which tries to find the entire signature in the target system. The reason behind this is that it is almost impossible to find an exact signature for a piece of software. Different execution scenarios, software updates, and different software versions may change parts of the software signature.

Instead, we try to find the similarity between the query and various software signatures. We define a threshold for each software. If the similarity of a query with each software's signature is greater than the software's threshold, we conclude the presence of the software on the target system.

To determine the threshold of a piece of software, we install and run it on a base operating system (a newly installed os with no other software). To determine the threshold, unlike making the signature, we do not compare the disk copies before and after running the software. Instead, we take a copy of the disk only after running the software. The reason behind this decision is that in the detection phase, we do not have access to a copy of the disk before running the software; we are facing a compromised system from which we can take a copy and identify what software has run on it. Now we extract the list of files and folders from this copy, calculate its similarity with the considered SSDE model and determine this similarity as the software threshold for that SSDE model. We should note that we have a separate threshold for each software per SSDE model.

In this work, to determine the threshold of a piece of software, we have only run the software with one scenario. A separate research study is needed to determine the appropriate scenario (scenarios) of software execution to set the threshold of a piece of software.

Fig. 4 shows how the software signature detection subsystem determines the presence or absence of a software package on the target system.

To calculate the similarity of a query with different signatures, we first need to convert them into numerical vectors. There are several ways to create numeric representations for text documents (here, software signatures). One of these methods is the bag-of-words model (Liu, 2013). This model first builds a dictionary of all the words in the set of signatures. Then, for each signature and each word in the dictionary, it is determined whether the word is present

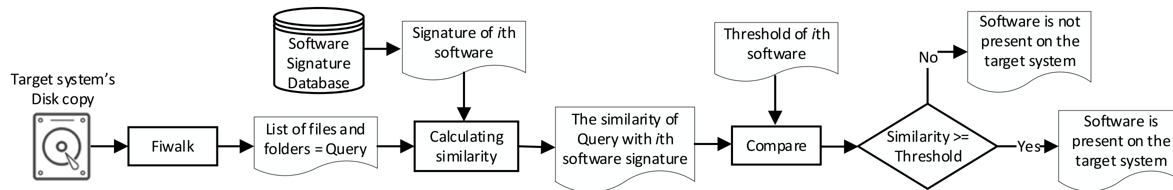

Fig. 4. Examining presence or absence of a software package.

in the signature or not. If the number of signatures is N and the number of terms (words) is T, then a T-dimensional vector space model is formed. The i th component of the signature vector S in this T-dimensional space is equal to one if the i th word of the dictionary is present in S; otherwise, this component is equal to zero.

Another way to display the signature vector is to consider the frequency of the words in the signature. Similar to the bag-of-words model, a T-dimensional vector space model is formed. The i th component of the signature vector S in this T-dimensional space is equal to the frequency of the i th word of the dictionary in S.

Another method, known as the term frequency-inverse document frequency (TF-IDF), considers both the frequency of a word and the inverse of the frequency of a word in the entire set of documents (Rajaraman and Ullman, 2011). In many cases, the TF-IDF method works better than the simple bag-of-words model or TF-only method because it considers the importance of a word within a document and within a set of documents (Ordonez et al., 2011; Akuma et al., 2022).

Some other vector representation methods don't consider the frequency of words but the context and meaning of words. These vectors are called word embeddings, and if a neural network model is used to construct these vectors, they are also called neural embeddings (Lashkari et al., 2019; Mikolov et al., 2013). While word embedding methods maintain semantic and syntactic relationships between words, they are slower than frequency-based models.

In this paper, we will use the TF-IDF weighing scheme to construct the software signature vectors. Later in Section 4.3, we will compare the performance of the TF-IDF models with some word embedding ones (Soltani et al., 2021).

We represent the collection of our signatures by a TF-IDF matrix with one row per software signature and one column per word in the corpus. Using the same TF-IDF matrix built for the collection of signatures, the query is tokenized. For a collection of N software signatures and T distinct words, the TF-IDF matrix is $N\times T$. Each of the elements of this matrix represents the weight of a word in a signature. The TF-IDF for a term t in a software signature S is computed as:

$$TF_IDF(t,S) = TF(t,S) \times IDF(t). \tag{16}$$

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

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

where $f(t,S)$ shows the frequency (count) of t in S. Another formula for TF is a logarithmic one (Aizawa, 2003) as defined in (18).

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

The IDF factor also has various possibilities (Aizawa, 2003), one of which is given by (19).

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

where N is the number of signatures in the collection, and $cf(t)$ is the number of signatures in the corpus 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 in (20).

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

Therefore, each signature in the collection is represented by a vector in T-dimensional space. Similarly, the query q is represented by a vector in this vector space. We can use a similarity measure to calculate the resemblance of the query q with signature S. The most straightforward similarity measure is the dot product of the two vectors, which cares about both angle and magnitude of two vectors:

$$sim(S,q) = \vec{S}.\vec{q} = \sum_{i=1}^{T} S_i q_i. \tag{21}$$

Another similarity measure is the cosine similarity which cares about the angle between two vectors:

$$cos_sim(S,q) = \frac{\vec{S}.\vec{q}}{\|\vec{S}\|\|\vec{q}\|} = \frac{\sum_{i=1}^{T} S_i q_i}{\sqrt{\sum_{i=1}^{T} S_i^2}\sqrt{\sum_{i=1}^{T} q_i^2}}. \tag{22}$$

To determine whether the software has run on the system, we should set a threshold for each software. If the similarity score of a software signature with a query (system under inspection) is higher than the threshold, we interpret that the software has run on the system.

3.3. Various models for SSDEs

Different design parameters are involved in building an SSDE. These parameters affect the software signature, the TF-IDF matrix, the similarity measure, and the final decision of whether an application has run on the system. The parameters include the following: 1) Composition of Difference-Sets, 2) n-gram types, 3) stop list, 4) TF-IDF formula type, 5) TF formula, 6) IDF formula, 7) Similarity measure, and 8) Threshold values. Table 2 shows these parameters and their different values.

Different values of these parameters lead to different models of SSDEs. The total number of SSDE models is equal to $2 \times 4 \times 2 \times 2 \times 2 \times 2 \times 2 \times 3 = 768$. To name each SSDE model, an eight-character string is used, with each position in the string assigned to a parameter. For example, the string $A_2B_3C_1D_2E_1F_2G_2H_3$ represents an SSDE model in which we have 1) union composition for combining Difference-Sets, 2) the last components of the file paths as terms, 3) no stop list, 4) the standard TF-IDF formula, 5) the simple TF formula, 6) the smooth IDF formula, 7) the cosine similarity measure, and 8) the small threshold. We should note that out of 768 SSDE models, 192 are duplicates, as models that use only TF (D_1 models) do not care about the IDF formula type (F_1 or F_2). So, we have 576 distinct SSDE models.

The first three parameters affect the construction of software signatures, which are explained in 3.1.1. Actually, we can create $2 \times$

Table 2
The design parameters of the SSDE.

	Parameter	Tag	Value	Description
A	Difference-Set composition	A_1	Intersection	The Difference-Sets are combined using the intersection of their elements.
		A_2	Union	The Difference-Sets are combined using the union of their elements.
B	n-gram type	B_1	All	The full file path is used as a term.
		B_2	Unigram	The file path is broken into its components, and each component will be a term.
		B_3	Last	The last component of the file path is considered as a term.
		B_4	Two Last	The last two components of the file path constitute a term.
C	Stop list	C_1	No Stop List	There is no stop list for common terms.
		C_2	Stop List	The stop list for common terms includes the file paths in the disk copy of the base OS.
D	TF-IDF formula type	D_1	TF-only	The weight matrix is computed using only term frequency.
		D_2	TF-IDF	The weight matrix is computed using the standard formula, as stated in Equation (16).
E	TF formula	E_1	Simple	The TF is computed using Equation (17).
		E_2	Logarithmic	The TF is computed using Equation (18).
F	IDF formula	F_1	Logarithmic	The IDF is computed using Equation (19).
		F_2	Smooth	The IDF is computed using Equation (20).
G	Similarity measure	G_1	Simple	The similarity presented in Equation (21) is used.
		G_2	Cosine	The similarity presented in Equation (22) is used.
H	Threshold value	H_1	Big	The similarity obtained from installing and running the software on the base OS.
		H_2	Medium	1/2 Big threshold
		H_3	Small	1/4 Big threshold

$4 \times 2 = 16$ different signatures for each software. The signatures for software *SW* in A_1 models consist of the intersection of Difference-Sets of several runs of *SW*. The signatures of *SW* in A_2 models include the union of Difference-Sets of different runs of *SW*. The signatures of *SW* in B_1 models contain the full file paths in the intersection or union of Difference-sets as n-grams. Similarly, the signatures of *SW* in B_2, B_3, and B_4 models include respectively every, the last, and the two-last components of the file paths in the intersection or union of Difference-Sets as n-grams. The signatures of *SW* in C_1 models do not pay attention to the stop list. Software signatures in C_2 models consider the stop list and remove the n-grams in the stop list from the software signature.

The fourth to seventh parameters are described in the previous section. The D_1 models use only the TF factor to build vectors of software signatures. The D_2 models use both TF and IDF factors for building signature vectors. The E_1 models use the TF formula in (17), and the E_2 models use the TF formula in (18). The F_1 models use the IDF formula in (19), and the F_2 models use the IDF formula in (20). The G_1 models use the simple similarity measure in (21), and the G_2 models use the cosine similarity measure in (22).

The eighth parameter determines the threshold value for different signatures. After creating an SSDE model, we set the threshold for each software. In other words, the threshold of a piece of software is calculated after making the software signatures, constructing the TF-IDF matrix, and determining the type of similarity measure for an SSDE model. To define the threshold for

Table 3
The description of our experiments.

Software	Scenario	Scenario Description
Adobe Reader 11	$S_{1,1}$	We opened Adobe Reader by clicking on its icon on the desktop, then we opened *file1.pdf*, and finally, we closed it.
	$S_{1,2}$	We opened Adobe Reader by clicking on its icon in the Start menu, then we opened *file1.pdf*, and finally, we closed the application.
	$S_{1,3}$	We opened *file2.pdf* by double-clicking it, then we made changes to it, saved it, and finally closed it.
	$S_{1,4}$	We opened *file1.pdf* by double-clicking it, then we made changes to it, saved it to another file, and finally closed it.
Firefox 67	$S_{2,1}$	We opened Firefox by clicking on its icon on the desktop, then we visited *yahoo.com*, and finally, we closed it.
	$S_{2,2}$	We opened Firefox by clicking on its icon in the Start menu, then we visited *google.com*, we searched for Olympic, we opened *www.olympic.org* in a new tab, and finally, we closed it.
	$S_{2,3}$	We clicked on the Firefox icon on the desktop, then we visited *gmail.com*, logged in, opened an email, downloaded the email attachment, which was a pdf file, and finally closed it.
	$S_{2,4}$	We clicked on the Firefox icon in the Start menu, opened the *en.um.ac.ir*, and then reviewed several successive links on this site and finally closed it.
Python 2.7	$S_{3,1}$	We selected Run, typed python, opened python.exe, typed the command *print "hello world"*, and finally closed it.
	$S_{3,2}$	We clicked Start menu, All Programs, Python 2.7.17, and IDLE. Then we ran several commands in IDLE and finally closed it.
Word 2013	$S_{4,1}$	We clicked on the Word icon on the desktop, created a new document, saved it as *doc1.docx* on the desktop, and finally closed it.
	$S_{4,2}$	We clicked on the Word icon in the Start menu, opened *doc2.docx*, made changes to it, saved it, and finally closed the application.
	$S_{4,3}$	We double-clicked on the *doc2.docx* file to open it, then we made changes to it, saved the changes to the *doc3.docx* file, and closed it.
	$S_{4,4}$	We right-clicked on the desktop and selected New Microsoft Word Document, and named it *doc1.docx*. Then we double-clicked on doc1 and opened it. We typed a text and saved it, and finally, we closed it.
7-Zip 19.0	$S_{5,1}$	We opened 7-Zip by clicking on its icon in the Start menu, then we compressed *music1.mp3*, and finally, we closed it.
	$S_{5,2}$	We opened 7-Zip by clicking on its icon in the Start menu, then we compressed *pic1.jpg*. Next, we decompressed the *pic1.7z* and write it to a new file, and finally, we closed it.
Cygnus Hex Editor 1.0	$S_{6,1}$	We clicked on the Cygnus Hex Editor icon on the desktop, then we opened *pic2.jpg* and viewed its hex values. Finally, we closed the application.
	$S_{6,2}$	We opened the Cygnus Hex Editor by clicking on its icon in the Start menu, then we opened *pic3.jpg* and made some changes in its hex representation. Then, we saved the changes and finally closed the application.
Invisible Secrets 4.6.2	$S_{7,1}$	We clicked on the Invisible Secrets icon in the Start menu and selected the "Hide Files" option. Then, we hid *data1.txt* in the *pic.png* carrier and named the new file as *hidden_data1*. Finally, we closed the application.
	$S_{7,2}$	We clicked on the Invisible Secrets icon in the Start menu and selected the "Encrypt Files" option. Then, we encrypted *data2.txt* into *data2.txt.isc*. Finally, we closed the application.
	$S_{7,3}$	We clicked on the Invisible Secrets icon in the Start menu and selected the "Encrypt Files" option. Then, we encrypted *data1.txt* into *data1.txt.isc*. Next, we selected the "Decrypt Files" option and decrypted the newly encrypted file. Finally, we closed the application.

Table 4
Ground truth of applications that have run on 14 M57 machines.

	Applications	Nov. 16	Nov. 17	Nov. 18	Nov. 19	Nov. 20	Nov. 23	Nov. 24
Pat's computer	Adobe Reader	−	−	−	−	+	+	+
	Firefox	+	+	−	+	+	+	+
	Python	−	+	−	+	+	−	+
	7-Zip	−	−	−	−	−	−	−
	Cygnus	−	−	−	−	−	−	−
	Invisible Secrets	−	−	−	−	−	−	−
Charlie's computer	Adobe Reader	−	−	−	−	−	−	−
	Firefox	+	+	+	−	+	+	+
	Python	−	+	+	+	+	+	+
	7-Zip	−	−	−	−	−	−	+
	Cygnus	−	−	−	−	−	−	+
	Invisible Secrets	−	−	−	+	−	−	+

software *SW*, we install and then run the *SW* on a base OS. We then query this machine against the SSDE and set the corresponding similarity value as the threshold. Therefore, each SSDE model has a separate threshold for each software. The three values in Table 2 for the threshold parameter are Big (H_1), Medium (H_2), and Small (H_3). The Big value for this parameter is the value calculated according to the above description. The H_2 value is half of the H_1, and the H_3 value is one-quarter of the H_1.

4. Experiments

Our experiments build the signatures for seven software, i.e., Adobe Reader, Firefox, Python, Word, 7-Zip, Cygnus Hex Editor, and Invisible Secrets on Windows 7 32-bit, with several different scenarios. The description of different running scenarios of these applications is given in Table 3.

As Table 3 shows, we have run Adobe Reader 11 four times with four different scenarios. Other pieces of software, including Firefox 67, Python 2.7, Word 2013, 7-Zip 19.0, Cygnus Hex Editor 1.0, and Invisible Secrets 4.6.2, have been run with four, two, four, two, two, and three different scenarios, respectively.

As explained in section 3.1.1, for each SSDE model, we create a software signature based on the corresponding parameter values. For example, in our experiments, we construct the signature of Adobe Reader 11 in an $A_1B_1C_1$ model as follows: we calculate the intersection of Difference-Sets obtained from four executions of Adobe Reader 11 (listed in Table 3). We consider all file paths in the resulting set as n-grams and do not consider a stop list.

As another example, we construct the signature of Adobe Reader 11 in an $A_2B_1C_2$ model as follows: we obtain the union of Difference-Sets obtained from four executions of Adobe Reader 11. We consider all the file paths in the resulting set as n-grams, and this time we select a stop list equal to all the file paths in the disk copy of the base operating system. If one of the file paths in the union of Difference-Sets is in the stop list, we remove it from the software signature.

As a final example, we construct the signature of Adobe Reader 11 in an $A_2B_3C_2$ model as follows: We calculate the union of Difference-Sets obtained from four executions of Adobe Reader 11. From the file paths in the resulting set, we consider the last components of the paths as n-grams. Here the stop list is equal to the last components of the file paths in the disk copy of the base OS.

To evaluate the SSDE models and determine the best ones, we examine them against M57 Patents machines. Six of the software packages listed in Tables 3 and i.e., Adobe Reader, Firefox, Python, 7-Zip, Cygnus, and Invisible Secrets, have run on M57 machines. In our experiments, we evaluate each SSDE model against disk images of Pat and Charlie computers over seven working days. Therefore, we have 14 test machines (queries). Table 4 shows the ground truth about the application execution on these machines found in (Nelson, 2016; Roussev and Quates, 2012; Corpora, 2009). The 14 machines listed in Table 4 make up our 14 queries.

We examine the 14 queries mentioned above on each SSDE model. For each SSDE model, we calculate the true-positive, true-negative, false-positive, and false-negative values for all queries. We have:

- True-Positive is the number of applications that the SSDE model correctly detects run on the controlled system.
- True-Negative is the number of applications that the SSDE model correctly detects not run on the controlled system.
- False-Positive is the number of applications that the SSDE model falsely detects run on the controlled system.
- False-Negative is the number of applications that the SSDE model falsely detects not run on the controlled system.

As said before, we have a threshold value for each software in each SSDE model. If a software package is executed on a test machine, the similarity of the query with the software signature should be greater than or equal to the corresponding threshold. Otherwise, if the software is not executed on the test machine, the similarity should be less than the corresponding threshold. This way, we can calculate true-positive, true-negative, false-positive, and false-negative. Algorithm 1 describes how to calculate these values.

Algorithm 1. Calculate evaluation metrics for an SSDE model

Algorithm 1 Calculate evaluation metrics for an SSDE model

```
1: signatures ← list of software signatures of the SSDE model
2: machines ← fourteen M57 machines (listed in Table 4)
3: TP, FP, TN, FN ← 0
4: for c in machines do
5:     softs ← list of software packages in c
6:     for i in signatures do
7:         if i in softs then
8:             if similarity(i) >= threshold(i) then
9:                 TP ← TP + 1
10:            else
11:                FN ← FN + 1
12:            end if
13:        else
14:            if similarity(i) >= threshold(i) then
15:                FP ← FP + 1
16:            else
17:                TN ← TN + 1
18:            end if
19:        end if
20:    end for
21: end for
22: Precision ← TP / (TP + FP )
23: Recall ← TP / (TP + FN )
```

As Algorithm 1 shows, TP, FP, TN, and FN are calculated cumulatively. The initial value of these four parameters is zero. Then, in two nested loops, the following checks are performed for all 14 machines (mentioned in Table 4) and for the signatures of 7 pieces of software (in Table 3):

If software i is executed in machine c according to Table 4, then if the similarity of machine c (query c) with the signature of software i is greater than or equal to the threshold of software *i*, TP is increased by one; otherwise, if the similarity is less than the threshold, FN is increased by one. Otherwise, if software i is not executed in machine c according to Table 4, then if the similarity of query c with the signature of software i is greater than or equal to the threshold of software *i*, one unit is added to FP; otherwise, if the similarity is less than the threshold, TN is added by one.

For example, in Table 4, we see that 7-Zip was run on Charlie's computer on November 24th. If the similarity of this machine with the 7-Zip signature is greater than or equal to the 7-Zip threshold, one unit is added to TP, and otherwise, one unit is added to FN. On the other hand, 7-Zip was not run on Charlie's computer on November 20th. If the similarity of this machine with the 7-Zip signature is greater than or equal to the 7-Zip threshold, FP is added by one, and otherwise, TN is added by one.

4.1. Experimental results

The evaluation of SSDE models can start with visualizing Precision and Recall. Fig. 5 draws a point in the Precision-Recall space for each SSDE model against 14 test machines. Because some SSDE models have the same Precision and Recall, some points represent more than one model. The darker blue circles represent more models. This inspection shows that some SSDE models have reached perfect Recall (1), and a few of them have reached perfect Precision, but no model has reached both. In fact, all models with perfect Recall have a Precision of less than 0.4, and all models with perfect Precision have a Recall of less than 0.1. Also, we see that most high Recall models have low Precision and most high Precision models have low Recall.

We divide the range [0, 1] into ten bins and determine the number of models with Precision or Recall belonging to each bin. Histograms of Fig. 6 show how many SSDE models fit within each bin. As we can see, while about 100 models have perfect Recall, only about 25 models have perfect Precision. However, if we consider the Recall and Precision greater than 0.8 to be appropriate, more than 240 models have the appropriate Precision, and also more than 130 models have good Recall.

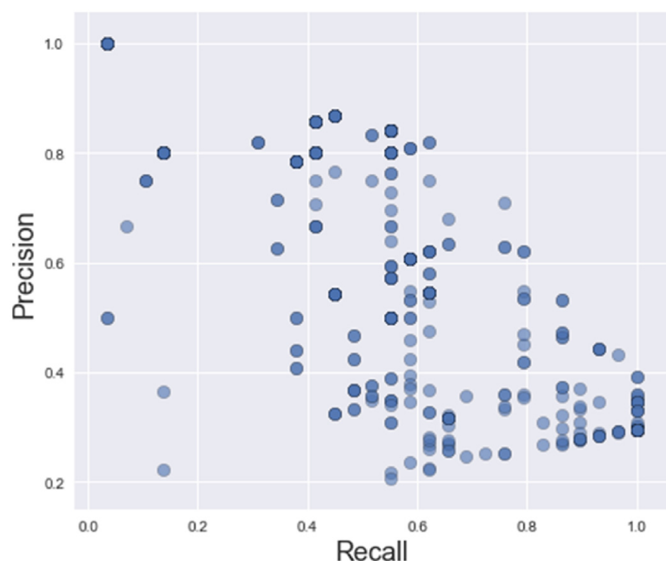

Fig. 5. Scatter plot of Precision-Recall scores of SSDE models against 14 M57 machines.

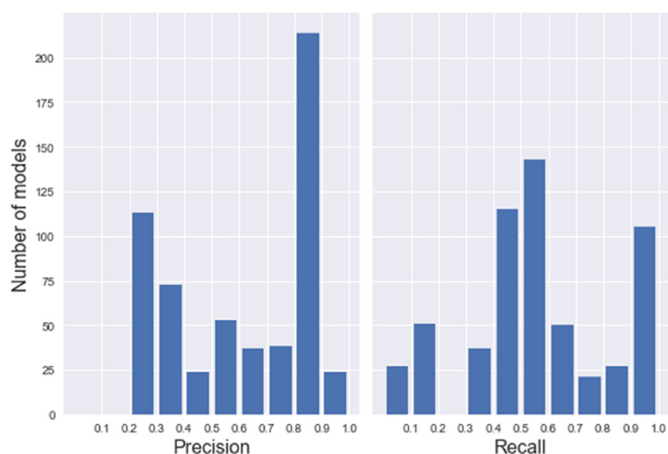

Fig. 6. Distribution of Precision and Recall of SSDE models.

Now we want to measure the effects of different parameter values on the performance of SSDE models. To see which parameter values lead to the top models, we can look at the bar charts of Fig. 7. The left diagram of this figure shows the percentage of SSDE models that have perfect Precision for different values of design parameters. The right diagram of Fig. 7 shows the percentage of SSDE models with perfect Recall for different values of design parameters.

As Fig. 7 shows, while the union or intersection of DifferenceSets has the same effect in models with perfect Precision, the union of sets has performed a little better than their intersection in models with perfect Recall. Regarding the second parameter (n-gram type), we see that only B_1 models have perfect Precision. On the other hand, only B_2 and B_3 models (with the majority of B_2 models) have perfect Recall. Regarding the third parameter, the presence or absence of the stop list has the same effect on the models with perfect Precision. However, not specifying the stop list has performed better in the models with perfect Recall.

Including or not including the IDF formula did not make a difference in models with perfect Precision, but not including the IDF formula performed better in models with perfect Recall. Regarding the fifth parameter, the logarithmic TF formula works better than the simple formula in models with perfect Precision, and it is the opposite in models with perfect Recall. Moreover, while there is no difference between the two IDF formulas in models with perfect Precision, the smooth IDF formula has performed better in models with perfect Recall.

In both perfect Precision and perfect Recall models, cosine similarity works better. In particular, all models with perfect Precision have used cosine similarity. In the models with perfect Recall, the small threshold is better than the medium, and the medium is better than the Big one. The threshold values in the models with perfect Precision have worked in reverse.

As Fig. 7 shows, many of the parameter values have a value of 0, especially in models with perfect Precision. This is because only a few models have perfect Precision. Moreover, as Fig. 5 shows, none of the models have a Precision greater than 0.9 and less than one. Therefore, we considered the models with Precision or Recall above 0.8 as appropriate models, and we examined the effect of the parameter values on these models. Fig. 8 shows the percentage of SSDE models with a Precision (left diagram) or Recall (right

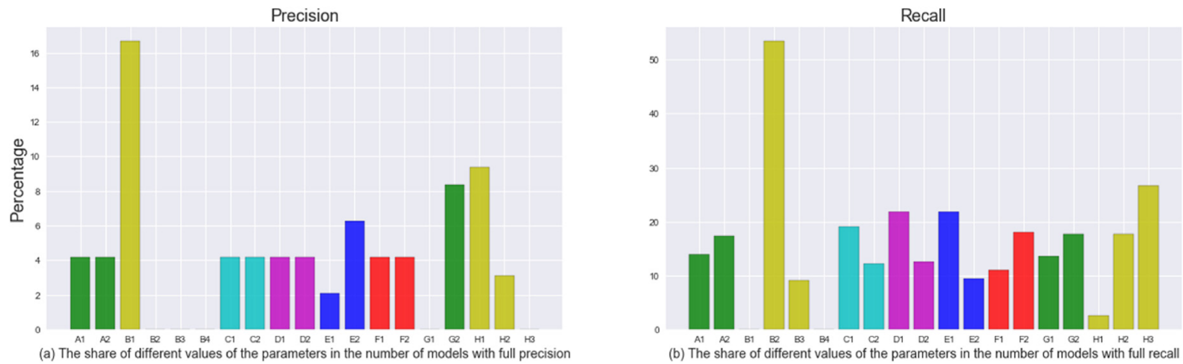

Fig. 7. Effects of different parameter values on SSDE models with perfect Precision or Recall.

diagram) greater than or equal to 0.8 for different values of design parameters.

As we can see in Fig. 8, the percentage of high Recall models with union composition (A_2) is more than the intersection composition (A_1). That is, union composition leads to more models with a Recall greater than 0.8. However, union and intersection composition work almost identically in the number of high Precision models. Regarding the second parameter, we see that B_1 models have the highest percentage among the high Precision models. Then there are B_4, B_3, and B_2 models, respectively. This is not true in high Recall models. B_1 models have a small share of high Recall models, and B_2 models have the largest share of high Recall models.

In the third parameter, we see that the existence of a stop list leads to more high Precision models, which is the opposite in models with high Recall. TF-IDF models have a higher percentage of high Precision SSDE models than TF-only models. Conversely, in high Recall models, TF-only models have a more prominent presence than TF-IDF models.

Different values of the fifth and sixth design parameters also act differently in terms of Precision and Recall. We see that the logarithmic TF formula performs better in terms of Precision and the simple TF acts better in terms of Recall. Also, the logarithmic IDF performs better in terms of Precision, and the smooth IDF works better in terms of Recall. We also see that the cosine similarity in both diagrams works better than the simple similarity. Regarding the threshold value, we observe that larger threshold values perform better in terms of Precision, and smaller threshold values work better in terms of Recall.

Finally, by comparing the diagrams of Figs. 7 and 8, we see that the effect of parameter values in models with Recall $\geq$ 0.8 is very similar to models with perfect Recall. It is roughly true for models with Precision $\geq$ 0.8 and models with perfect Precision.

4.2. Experiments based on the lasting effect of software execution

Since the effects of running a software package on a hard disk usually last for a while (Soltani et al., 2019; Grier, 2011), in our new experiments, we assume that if a software package is run on the system in one day, its effects on the system in the following days are visible. There is a limit to the stability of the effect of a software package, and we have considered two days here.

Table 5 is based on Table 4, considering this two-day period. The red asterisk signs in this table indicate these effects. For example, as Table 4 shows, Firefox has run on Pat's computer on November 17th but not on November 18th. Considering the two-days effect of the software's execution, in Table 5, Firefox is also considered executed on November 18. Since Firefox was executed on November 19th (as mentioned in Table 4), there is no need to change the table entry for this day. As another example, in Table 4, we see that Invisible Secrets was run on Charlie's computer on November 19 but not on November 20 and 23. In Table 5, we have marked November 20 and 23 for Invisible Secrets, considering the two-days effect of software execution.

The experiments in this phase are performed on 14 test machines based on the ground truth in Table 5. The difference between the experiments in this phase and the previous phase is in the number of days that a piece of software is considered executed. This difference affects the calculation of TP, FP, TN, and FN values in Algorithm 1. For example, in the previous experiments, Firefox was

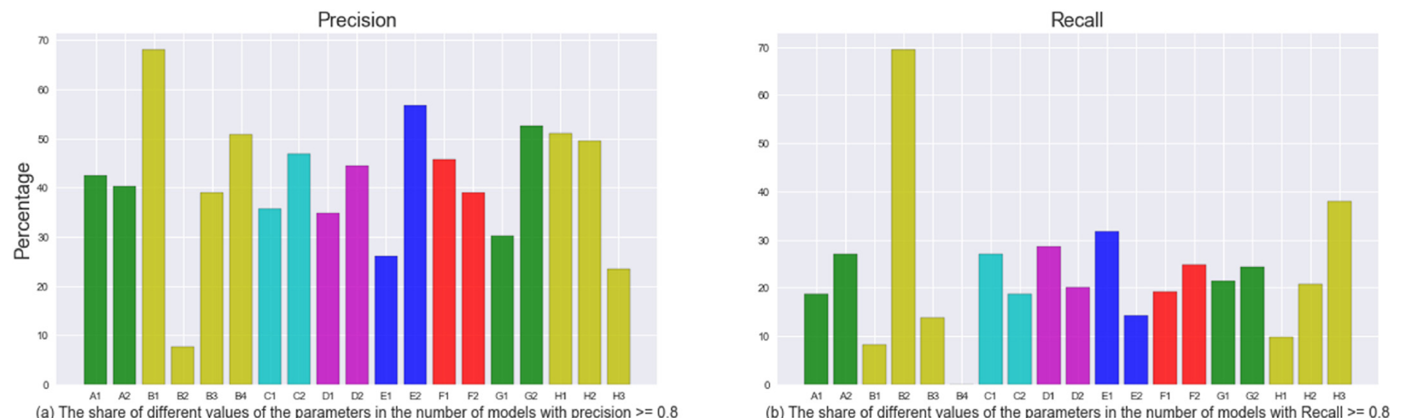

Fig. 8. Effects of different parameter values on appropriate SSDE models in terms of Precision and Recall.

considered not executed on November 18. Therefore, according to Algorithm 1, if the similarity of the query belonging to this day with the Firefox signature is greater than or equal to the Firefox threshold, FP is added by one; otherwise, if the similarity is less than the threshold, TN is added by one. In the experiments of this phase, Firefox is considered executed on November 18. Therefore, if the similarity of this day's query with the Firefox signature is greater than or equal to the Firefox threshold, TP is added by one; otherwise, if the similarity is less than the threshold, FN is increased by one.

Fig. 9 shows the Precision and Recall values of different SSDE models against 14 test machines based on the ground truth presented in Table 5.

To compare the performance of SSDE models in our two experiments, we can look at the histograms in Fig. 10. In this figure, Experiments1 are the experiments of the previous section (with ground truth in Table 4), and Experiments2 are the experiments of this section (with ground truth in Table 5). We see that the SSDE models achieve better Precision in Experiments2. However, there is no significant improvement in the Recall rate of Experiments2 compared to Experiments1.

We see that in Experiments2, about 38% of the models achieve near-perfect Precision, and about 18% of the models achieve near-perfect Recall, and this is while the operating system version of the M57 machines (Windows XP) is different from our experiment machines (Windows 7). Besides, the software versions used in our experiments are different from the software versions of the M57 machines.

In this set of experiments, we have also measured the effect of parameter values on models with perfect Precision and models with perfect Recall. Fig. 11 shows the influence of parameter values on these models. By comparing Fig. 11 (Experiments2) and Fig. 7 (Experiments1), we conclude that the parameter values have a relatively similar effect in both experiments. Besides, Fig. 12 shows the influence of parameter values on models with Precision ≥ 0.8 and models with Recall ≥ 0.8 in Experiments2. Comparing Figs. 12 and 8, we see that parameter values have a similar influence in the two experiments.

Tables I and II in Appendix 1 list the SSDE models with perfect Precision and perfect Recall in Experiments2, respectively. To see the share of each of the design parameter values in these superior models, we can look at Fig. 13.

In models with perfect Recall, we have:

- The union composition performs better than the intersection composition.
- B_2 Models have the maximum presence.
- Not setting a stop list works better.
- The standard TF-IDF formula works a little better.

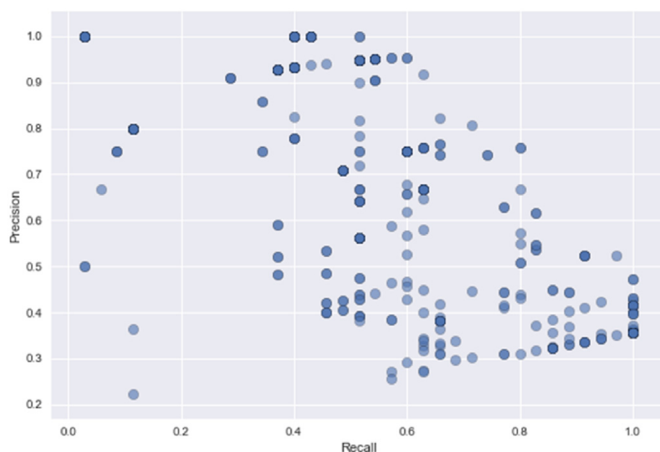

Fig. 9. Scatter plot of Precision-Recall scores of SSDE models against 14 M57 machines (ground truth in Table 5).

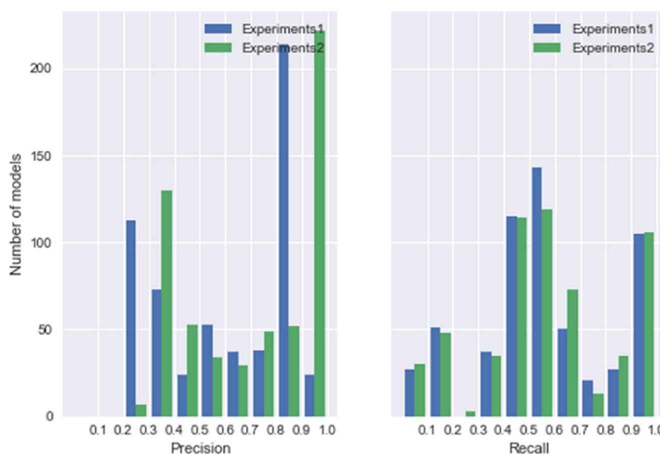

Fig. 10. Distribution of Precision and Recall of SSDE models in Experiments1 and Experiments2.

- The simple TF formula acts better.
- The smooth IDF formula has a little more presence.
- The cosine similarity works better.
- The small threshold has the best performance.

In models with perfect Precision, we see that:

Table 5
Ground truth of applications that have run on 14 M57 machines, considering the lasting effect of software run.

	Applications	Nov. 16	Nov. 17	Nov. 18	Nov. 19	Nov. 20	Nov. 23	Nov. 24
Pat's machine	Adobe Reader	−	−	−	−	+	+	+
	Firefox	+	+	*	+	+	+	+
	Python	−	+	*	+	+	*	+
	7-Zip	−	−	−	−	−	−	−
	Cygnus	−	−	−	−	−	−	−
	Invisible Secrets	−	−	−	−	−	−	−
Charlie's machine	Adobe Reader	−	−	−	−	−	−	−
	Firefox	+	+	+	*	+	+	+
	Python	−	+	+	+	+	+	+
	7-Zip	−	−	−	−	−	−	+
	Cygnus	−	−	−	−	−	−	+
	Invisible Secrets	−	−	−	+	*	*	+

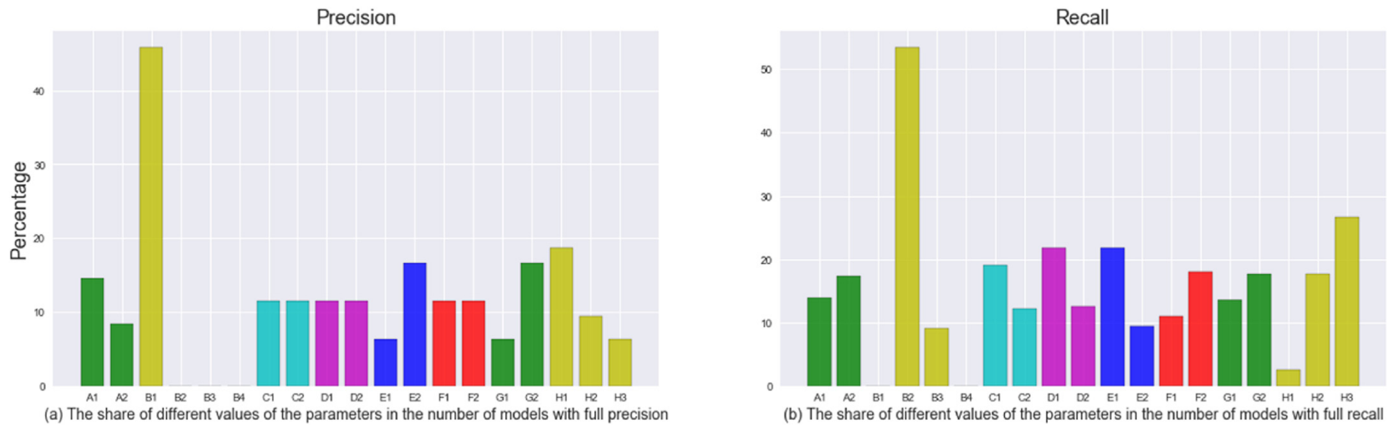

Fig. 11. Effects of different parameter values on SSDE models with perfect Precision or Recall (Experiments2).

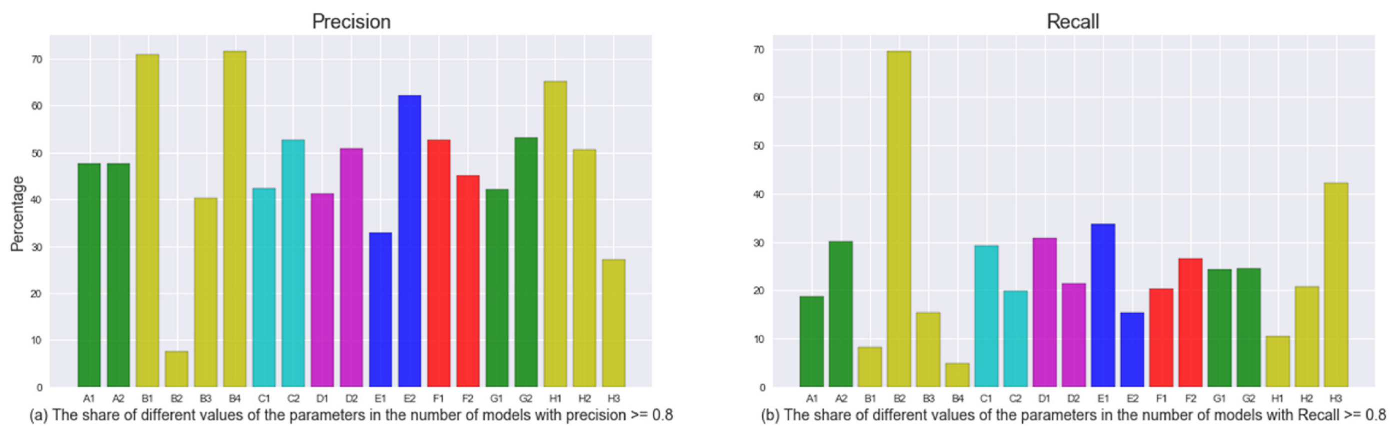

Fig. 12. Effects of different parameter values on appropriate SSDE models in terms of Precision and Recall (Experiments2).

- The intersection composition acts better than the union composition.
- The full file path is the only n-gram type.
- 50% of the models have a stop list, and 50% do not.
- The standard TF-IDF formula is more prominent.
- The logarithmic TF formula works better.
- The smooth IDF (F_2 models) is twice as present as logarithmic IDF (F_1 models), but given that the total number of F_2 models is twice the number of F_1 models, we do not differ much in terms of the IDF formula.
- Cosine similarity works much better than simple similarity.
- Finally, larger threshold values perform better.

In short, the different values of the design parameters have worked very differently in high Precision models and high Recall models. To select the superior SSDE models, we must determine whether to prioritize high Recall or high Precision. In investigating a case, all pieces of evidence must be collected. The lack of a piece of evidence may affect the final decision of the court. Therefore, in digital forensic investigation, Recall is more significant than Precision (Du and Scanlon, 2019; Mashhadani et al., 2019; Lillis and Scanlon, 2016; Porter and Petrovic, 2018; Beebe and Liu, 2014). Although we may have to increase the false positive rate to increase Recall, this is tolerable because any pertinent inculpatory or exculpatory artifact cannot be ignored (Du and Scanlon, 2019).

Therefore, we introduce the models in Table II of the attachment as top models. Finally, if we want to choose only one model among these models as the final superior model, we will look at the ratios

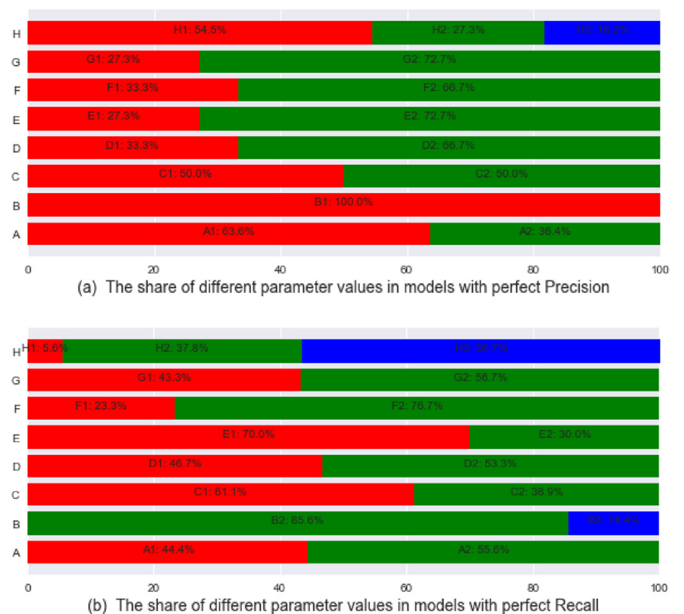

Fig. 13. The share of different parameter values in models with perfect Precision and perfect Recall.

in Table 6. Among the models in Table II of the attachment, we choose A_2 models because the share of A_2 in perfect Recall models is

Table 6
The average Precision and Recall of SSDE models and S3E models (Soltani et al., 2021).

	Average Precision (ground truth in Table 4)	Average Recall (ground truth in Table 4)	Average Precision (ground truth in Table 5)	Average Recall (ground truth in Table 5)	Time Cost (ms) = train + search time
SSDE models	0.607	0.566	0.693	0.555	6.9 + 119.6 = 126.5
S3E models (Soltani et al., 2021)	0.25	0.646	0.307	0.667	1343.4 + 927.1 = 2270.5

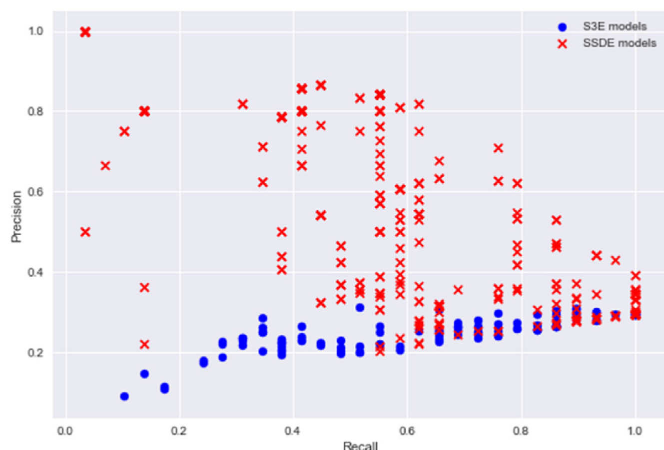

Fig. 14. Precision-Recall scores of SSDE models and S3E models (Soltani et al., 2021) against 14 M57 machines (ground truth in Table 4).

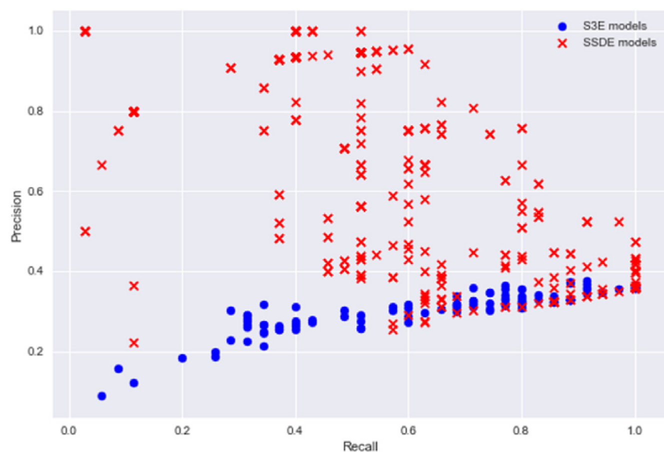

Fig. 15. Precision-Recall scores of SSDE models and S3E models (Soltani et al., 2021) against 14 M57 machines (ground truth in Table 5).

higher than A_1. Among the A_2 models in this table, we choose the B_2 models. Then, among the A_2B_2 models in this table, we choose the C_1 models. If we continue, we end up with $A_2B_2C_1D_2E_1F_2G_2H_3$ as the final top model.

4.3. Performance comparison

In this section, we will compare this work with our previous work. In (Soltani et al., 2021), to detect the presence of software, we designed software signature search engines (S3Es). We used the doc2vec paragraph vector method to construct the software signature vector and presented 120 different S3E models. We examine these S3E models against 14 M57 machines described in Table 4. Fig. 14 shows the Precision-Recall values of 120 S3E models and the Precision-Recall values of 576 SSDE models. As can be seen, the Recall rate of S3E models, similar to SSDE models, is scattered in the whole range (0,1). This means that we have some Recall values between (0,0.2) (0.2,0.4) (0.4,0.6) (0.6,0.8), and (0.8,1) in both models. However, the Recall dispersion is not the same between these two models. However, the Precision rate of S3E models is lower than SSDE models. We also run S3E models against 14 M57 machines that consider the lasting effect of software implementation (Table 5). Fig. 15 shows the Precision-Recall values of S3E models and the Precision-Recall values of SSDE models against these machines. We see similar results here.

Table 6 shows the average Precision and Recall of S3E and SSDE models against the two test sets in Tables 4 and 5. As can be seen, in both test sets, the S3E models have lower average Precision and a slightly higher average Recall. Table 6 also shows the time complexity of SSDE and S3E models, which includes the time needed to train the software signatures and the time needed to calculate the similarity of a query with software signatures. We see that the train time and the average search time of S3E models are much longer than frequency-based SSDE models.

5. Conclusion and future work

In this paper, for digital forensic triage purposes, we proposed a software signature detection engine, which includes two subsystems: software signature construction and detection. To build the software signature, we presented a differential analysis model that calculates the difference between the file system information before and after running the software in isolated conditions. We considered eight design parameters with different values, which resulted in 576 distinct SSDE models. We tested each of these SSDE models against 14 pseudo-real machines from the M57 Patents scenario. We introduced the top SSDE models based on Precision and Recall, and we identified the values of the design parameters that lead to these superior models.

As future work, we can consider other design parameters for the software signature detection engine. For instance, to determine the effect of the operating system version and the software version on the software signature, we can specify two parameters: grouping/non-grouping based on the software version and grouping/non-grouping based on the OS version. In OS version grouping, the software signature is the same on different operating systems. However, in non-grouping, the software signatures are different on different OS versions. Also, in software version grouping, a single signature is considered for various versions of a software package. While in non-grouping, each version of the software will have its signature. Besides, in future work, we should construct an extensive database of software signatures. For each software package, we should consider various versions on different operating systems.

In this paper, we used the file path to build the software signature. In the future, we can utilize other file system metadata such as MACB timestamps, file size, and hash values. Besides, we can consider other artifacts from disk copy, such as the Registry keys, to enhance our software signature detection engines.

In this paper, we used the TF-IDF weighting scheme to plot each software signature as a vector in a multidimensional space, and we calculated the TF and IDF factors based on two formulas. Of course, there are different methods in the literature for calculating these two factors that we can examine in the future. Also, other similarity criteria can be used to calculate the similarity between the two signature and query vectors.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Appendix 1

Table I
SSDE models with perfect Precision

	Name		Name		Name		Name		Name
1	$A_1B_1C_1D_1E_1F_2G_1H_1$	2	$A_1B_1C_1D_1E_1F_2G_2H_1$	3	$A_1B_1C_1D_1E_1F_2G_2H_2$	4	$A_1B_1C_1D_1E_2F_2G_1H_1$	5	$A_1B_1C_1D_1E_2F_2G_2H_1$
6	$A_1B_1C_1D_1E_2F_2G_2H_2$	7	$A_1B_1C_1D_1E_2F_2G_2H_3$	8	$A_1B_1C_1D_2E_1F_1G_1H_1$	9	$A_1B_1C_1D_2E_1F_1G_2H_1$	10	$A_1B_1C_1D_2E_1F_1G_2H_2$
11	$A_1B_1C_1D_2E_1F_2G_1H_1$	12	$A_1B_1C_1D_2E_1F_2G_2H_1$	13	$A_1B_1C_1D_2E_1F_2G_2H_2$	14	$A_1B_1C_1D_2E_2F_1G_1H_1$	15	$A_1B_1C_1D_2E_2F_1G_2H_1$
16	$A_1B_1C_1D_2E_2F_1G_2H_2$	17	$A_1B_1C_1D_2E_2F_1G_2H_3$	18	$A_1B_1C_1D_2E_2F_2G_1H_1$	19	$A_1B_1C_1D_2E_2F_2G_2H_1$	20	$A_1B_1C_1D_2E_2F_2G_2H_2$
21	$A_1B_1C_1D_2E_2F_2G_2H_3$	22	$A_1B_1C_2D_1E_1F_2G_1H_1$	23	$A_1B_1C_2D_1E_1F_2G_2H_1$	24	$A_1B_1C_2D_1E_1F_2G_2H_2$	25	$A_1B_1C_2D_1E_2F_2G_1H_1$
26	$A_1B_1C_2D_1E_2F_2G_2H_1$	27	$A_1B_1C_2D_1E_2F_2G_2H_2$	28	$A_1B_1C_2D_1E_2F_2G_2H_3$	29	$A_1B_1C_2D_2E_1F_1G_1H_1$	30	$A_1B_1C_2D_2E_1F_1G_2H_1$
31	$A_1B_1C_2D_2E_1F_1G_2H_2$	32	$A_1B_1C_2D_2E_1F_2G_1H_1$	33	$A_1B_1C_2D_2E_1F_2G_2H_1$	34	$A_1B_1C_2D_2E_1F_2G_2H_2$	35	$A_1B_1C_2D_2E_2F_1G_1H_1$
36	$A_1B_1C_2D_2E_2F_1G_2H_1$	37	$A_1B_1C_2D_2E_2F_1G_2H_2$	38	$A_1B_1C_2D_2E_2F_1G_2H_3$	39	$A_1B_1C_2D_2E_2F_2G_1H_1$	40	$A_1B_1C_2D_2E_2F_2G_2H_1$
41	$A_1B_1C_2D_2E_2F_2G_2H_2$	42	$A_1B_1C_2D_2E_2F_2G_2H_3$	43	$A_2B_1C_1D_1E_2F_2G_1H_1$	44	$A_2B_1C_1D_1E_2F_2G_2H_1$	45	$A_2B_1C_1D_1E_2F_2G_2H_2$
46	$A_2B_1C_1D_1E_2F_2G_2H_3$	47	$A_2B_1C_1D_2E_2F_1G_1H_1$	48	$A_2B_1C_1D_2E_2F_1G_2H_1$	49	$A_2B_1C_1D_2E_2F_1G_2H_2$	50	$A_2B_1C_1D_2E_2F_1G_2H_3$
51	$A_2B_1C_1D_2E_2F_2G_1H_1$	52	$A_2B_1C_1D_2E_2F_2G_2H_1$	53	$A_2B_1C_1D_2E_2F_2G_2H_2$	54	$A_2B_1C_1D_2E_2F_2G_2H_3$	55	$A_2B_1C_2D_1E_2F_2G_1H_1$
56	$A_2B_1C_2D_1E_2F_2G_2H_1$	57	$A_2B_1C_2D_1E_2F_2G_2H_2$	58	$A_2B_1C_2D_1E_2F_2G_2H_3$	59	$A_2B_1C_2D_2E_2F_1G_1H_1$	60	$A_2B_1C_2D_2E_2F_1G_2H_1$
61	$A_2B_1C_2D_2E_2F_1G_2H_2$	62	$A_2B_1C_2D_2E_2F_1G_2H_3$	63	$A_2B_1C_2D_2E_2F_2G_1H_1$	64	$A_2B_1C_2D_2E_2F_2G_2H_1$	65	$A_2B_1C_2D_2E_2F_2G_2H_2$
66	$A_2B_1C_2D_2E_2F_2G_2H_3$								

Table II
SSDE models with perfect Recall

	Name		Name		Name		Name		Name
1	$A_1B_2C_1D_1E_1F_2G_1H_2$	2	$A_1B_2C_1D_1E_1F_2G_1H_3$	3	$A_1B_2C_1D_1E_1F_2G_2H_1$	4	$A_1B_2C_1D_1E_1F_2G_2H_2$	5	$A_1B_2C_1D_1E_1F_2G_2H_3$
6	$A_1B_2C_1D_1E_2F_2G_1H_2$	7	$A_1B_2C_1D_1E_2F_2G_1H_3$	8	$A_1B_2C_1D_1E_2F_2G_2H_2$	9	$A_1B_2C_1D_1E_2F_2G_2H_3$	10	$A_1B_2C_1D_2E_1F_1G_1H_2$
11	$A_1B_2C_1D_2E_1F_1G_1H_3$	12	$A_1B_2C_1D_2E_1F_1G_2H_2$	13	$A_1B_2C_1D_2E_1F_1G_2H_3$	14	$A_1B_2C_1D_2E_1F_2G_1H_2$	15	$A_1B_2C_1D_2E_1F_2G_1H_3$
16	$A_1B_2C_1D_2E_1F_2G_2H_2$	17	$A_1B_2C_1D_2E_1F_2G_2H_3$	18	$A_1B_2C_1D_2E_2F_1G_1H_3$	19	$A_1B_2C_1D_2E_2F_1G_2H_3$	20	$A_1B_2C_1D_2E_2F_2G_1H_3$
21	$A_1B_2C_1D_2E_2F_2G_2H_3$	22	$A_1B_2C_2D_1E_1F_2G_1H_2$	23	$A_1B_2C_2D_1E_1F_2G_1H_3$	24	$A_1B_2C_2D_1E_1F_2G_2H_1$	25	$A_1B_2C_2D_1E_1F_2G_2H_2$
26	$A_1B_2C_2D_1E_1F_2G_2H_3$	27	$A_1B_2C_2D_1E_2F_2G_1H_2$	28	$A_1B_2C_2D_1E_2F_2G_1H_3$	29	$A_1B_2C_2D_1E_2F_2G_2H_2$	30	$A_1B_2C_2D_1E_2F_2G_2H_3$
31	$A_1B_2C_2D_2E_1F_1G_1H_3$	32	$A_1B_2C_2D_2E_1F_1G_2H_2$	33	$A_1B_2C_2D_2E_1F_1G_2H_3$	34	$A_1B_2C_2D_2E_1F_2G_1H_2$	35	$A_1B_2C_2D_2E_1F_2G_1H_3$
36	$A_1B_2C_2D_2E_1F_2G_2H_2$	37	$A_1B_2C_2D_2E_1F_2G_2H_3$	38	$A_1B_2C_2D_2E_2F_2G_2H_3$	39	$A_1B_3C_1D_1E_1F_2G_2H_3$	40	$A_1B_3C_1D_2E_1F_2G_2H_3$
41	$A_2B_2C_1D_1E_1F_2G_1H_2$	42	$A_2B_2C_1D_1E_1F_2G_1H_3$	43	$A_2B_2C_1D_1E_1F_2G_2H_1$	44	$A_2B_2C_1D_1E_1F_2G_2H_2$	45	$A_2B_2C_1D_1E_1F_2G_2H_3$
46	$A_2B_2C_1D_1E_2F_2G_1H_2$	47	$A_2B_2C_1D_1E_2F_2G_1H_3$	48	$A_2B_2C_1D_1E_2F_2G_2H_2$	49	$A_2B_2C_1D_1E_2F_2G_2H_3$	50	$A_2B_2C_1D_2E_1F_1G_1H_2$
51	$A_2B_2C_1D_2E_1F_1G_1H_3$	52	$A_2B_2C_1D_2E_1F_1G_2H_2$	53	$A_2B_2C_1D_2E_1F_1G_2H_3$	54	$A_2B_2C_1D_2E_1F_2G_1H_2$	55	$A_2B_2C_1D_2E_1F_2G_1H_3$
56	$A_2B_2C_1D_2E_1F_2G_2H_2$	57	$A_2B_2C_1D_2E_1F_2G_2H_3$	58	$A_2B_2C_1D_2E_2F_1G_1H_3$	59	$A_2B_2C_1D_2E_2F_1G_2H_3$	60	$A_2B_2C_1D_2E_2F_2G_1H_3$
61	$A_2B_2C_1D_2E_2F_2G_2H_3$	62	$A_2B_2C_2D_1E_1F_2G_1H_2$	63	$A_2B_2C_2D_1E_1F_2G_1H_3$	64	$A_2B_2C_2D_1E_1F_2G_2H_1$	65	$A_2B_2C_2D_1E_1F_2G_2H_2$
66	$A_2B_2C_2D_1E_1F_2G_2H_3$	67	$A_2B_2C_2D_1E_2F_2G_1H_2$	68	$A_2B_2C_2D_1E_2F_2G_1H_3$	69	$A_2B_2C_2D_1E_2F_2G_2H_2$	70	$A_2B_2C_2D_1E_2F_2G_2H_3$
71	$A_2B_2C_2D_2E_1F_1G_1H_3$	72	$A_2B_2C_2D_2E_1F_1G_2H_2$	73	$A_2B_2C_2D_2E_1F_1G_2H_3$	74	$A_2B_2C_2D_2E_1F_2G_1H_2$	75	$A_2B_2C_2D_2E_1F_2G_1H_3$
76	$A_2B_2C_2D_2E_1F_2G_2H_2$	77	$A_2B_2C_2D_2E_1F_2G_2H_3$	78	$A_2B_2C_2D_2E_2F_2G_1H_3$	79	$A_2B_2C_2D_2E_2F_2G_2H_3$	80	$A_2B_3C_1D_1E_1F_2G_1H_2$
81	$A_2B_3C_1D_1E_1F_2G_1H_3$	82	$A_2B_3C_1D_1E_1F_2G_2H_1$	83	$A_2B_3C_1D_1E_1F_2G_2H_2$	84	$A_2B_3C_1D_1E_1F_2G_2H_3$	85	$A_2B_3C_1D_2E_1F_1G_1H_3$
86	$A_2B_3C_1D_2E_1F_1G_2H_2$	87	$A_2B_3C_1D_2E_1F_1G_2H_3$	88	$A_2B_3C_1D_2E_1F_2G_1H_3$	89	$A_2B_3C_1D_2E_1F_2G_2H_2$	90	$A_2B_3C_1D_2E_1F_2G_2H_3$

References

Adegbehingbe, O., Jones Jr., J.H., 2019. Improved Decay Tolerant Inference of Previously Uninstalled Computer Applications.

Aizawa, A., 2003. An information-theoretic perspective of tf-idf measures. Inf. Process. Manag. 39 (1), 45–65.

Akuma, S., Lubem, T., Adom, I.T., 2022. Comparing Bag of Words and TF-IDF with different models for hate speech detection from live tweets. Int. J. Inf. Technol. 1–7.

Al-Sharif, Z.A., Al-Saleh, M.I., Jararweh, Y., Alawneh, L., Shatnawi, A.S., 2019. The effects of platforms and languages on the memory footprint of the executable Program: a memory forensic approach. J. UCS 25 (9), 1174–1198.

Arshad, H., Jantan, A., Hoon, G.K., Butt, A.S., 2019. A multilayered semantic framework for integrated forensic acquisition on social media. Digit. Invest. 29, 147–158.

Beebe, N.L., Liu, L., 2014. Clustering digital forensic string search output. Digit. Invest. 11 (4), 314–322.

Bhandari, S., Jusas, V., 2020. An ontology based on the timeline of log2timeline and psort using abstraction approach in digital forensics. Symmetry 12 (4), 642.

Bunting, S., 2012. EnCase Computer Forensics – the Official EnCE: EnCase Certified Examiner Study Guide. Wiley, p. 752.

Casey, E., Zehnder, A., 2020. Inter-regional digital forensic knowledge management: needs, challenges, and solutions. J. Forensic Sci. 66 (2), 619–629.

Chabot, Y., Bertaux, A., Nicolle, C., Kechadi, T., 2015. An ontology-based approach for the reconstruction and analysis of digital incidents timelines. Digit. Invest. 15, 83–100.

Chang, D., Ghosh, M., Sanadhya, S.K., Singh, M., White, D.R., 2019. FbHash: a new similarity hashing scheme for digital forensics. Digit. Invest. 29, S113–S123.

Corpora, D., 2009. M57-Patents scenario. https://digitalcorpora.org/corpora/scenarios/m57-patents-scenario (accessed 2021).

Debinski, M., Breitinger, F., Mohan, P., 2019. Timeline2GUI: a Log2timeline CSV parser and training scenarios. Digit. Invest. 28, 34–43.

Du, X., Scanlon, M., 2019. Methodology for the automated metadata-based classification of incriminating digital forensic artefacts. In: Proceedings of the 14th International Conference on Availability. Reliability and Security, pp. 1–8.

Du, X., et al., 2020. SoK: exploring the state of the art and the future potential of artificial intelligence in digital forensic investigation. In: Proceedings of the 15th International Conference on Availability. Reliability and Security, pp. 1–10.

Garfinkel, S.L., 2009. Automating disk forensic processing with SleuthKit, XML and Python. In: 2009 Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering. IEEE, pp. 73–84.

Gentry, E., McIntyre, R., Soltys, M., Lyu, F., 2019. SEAKER: a tool for fast digital forensic triage. In: Future of Information and Communication Conference. Springer, pp. 1227–1243.

Ghazinour, K., Vakharia, D.M., Kannaji, K.C., Satyakumar, R., 2017. A study on digital forensic tools. In: 2017 IEEE International Conference on Power, Control, Signals and Instrumentation Engineering (ICPCSI). IEEE, pp. 3136–3142.

Good, R., Peterson, G., 2017. Automated collection and correlation of file provenance information. In: IFIP International Conference on Digital Forensics. Springer, pp. 269–284.

Google. Timesketch: collaborative forensic timeline analysis. https://github.com/google/timesketch (accessed 2021).

Grier, J., 2011. Detecting data theft using stochastic forensics. Digit. Invest. 8, S71–S77.

Guðjónsson, K., 2010. Mastering the Super Timeline with Log2timeline. SANS Institute.

Hales, G., Bayne, E., 2019. Investigating visualisation techniques for rapid triage of digital forensic evidence. In: International Conference on Human-Computer Interaction. Springer, pp. 277–293.

Hargreaves, C., Patterson, J., 2012. An automated timeline reconstruction approach for digital forensic investigations. Digit. Invest. 9, S69–S79.

James, J.I., Gladyshev, P., 2015. Automated inference of past action instances in digital investigations. Int. J. Inf. Secur. 14 (3), 249–261.

Jeong, D., Lee, S., 2019. Forensic signature for tracking storage devices: analysis of UEFI firmware image, disk signature and windows artifacts. Digit. Invest. 29, 21–27.

Jones, J., Khan, T., Laskey, K., Nelson, A., Laamanen, M., White, D., 2016. Inferring Previously Uninstalled Applications from Residual Partial Artifacts.

Jusas, V., Birvinskas, D., Gahramanov, E., 2017. Methods and tools of digital triage in forensic context: survey and future directions. Symmetry 9 (4), 49.

Kälber, S., Dewald, A., Freiling, F.C., 2013. Forensic application-fingerprinting based on file system metadata. In: 2013 Seventh International Conference on it Security Incident Management and it Forensics. IEEE, pp. 98–112.

Karie, N.M., Kebande, V.R., 2016. Building ontologies for digital forensic terminologies. Int. J. Cyber-Secur. Digital Forensics 5 (2), 75–83.

Khader, M., Hadi, A., Al-Naymat, G., 2018. HDFS file operation fingerprints for forensic investigations. Digit. Invest. 24, 50–61.

Khan, M.N.A., 2012. Performance analysis of Bayesian networks and neural networks in classification of file system activities. Comput. Secur. 31 (4), 391–401.

Lashkari, F., Bagheri, E., Ghorbani, A.A., 2019. Neural embedding-based indices for semantic search. Inf. Process. Manag. 56 (3), 733–755.

Latzo, T., 2020. Efficient fingerprint matching for forensic event reconstruction. In: International Conference on Digital Forensics and Cyber Crime. Springer, pp. 98–120.

Lillis, D., Scanlon, M., 2016. On the benefits of information retrieval and information extraction techniques applied to digital forensics. In: Advanced Multimedia and Ubiquitous Engineering. Springer, pp. 641–647.

Lillis, D., Becker, B., O'Sullivan, T., Scanlon, M., 2016. Current Challenges and Future Research Areas for Digital Forensic Investigation arXiv preprint arXiv:1604.03850.

Lim, M., Jones, J., 2020. A digital media similarity measure for triage of digital forensic evidence. In: IFIP International Conference on Digital Forensics. Springer, pp. 111–135.

Liu, J., 2013. Image Retrieval Based on Bag-Of-Words Model arXiv preprint arXiv:1304.5168.

Mashhadani, S., Clarke, N.L., Li, F., 2019. Identification and Extraction of Digital Forensic Evidence from Multimedia Data Sources Using Multi-Algorithmic Fusion. ICISSP, pp. 438–448.

Metz, J., Guðjónsson, K.. Plaso - super timeline all the things. https://github.com/log2timeline/plaso (accessed 2021).

Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J., 2013. Distributed representations of words and phrases and their compositionality. In: Advances in Neural Information Processing Systems, pp. 3111–3119.

Mistry, N.R., Dahiya, M., 2019. Signature based volatile memory forensics: a detection based approach for analyzing sophisticated cyber attacks. Int. J. Inf. Technol. 11 (3), 583–589.

Mohanta, A., Saldanha, A., 2020. Memory forensics with volatility. In: Malware Analysis and Detection Engineering. Springer, pp. 433–476.

Moia, V.H.G., Henriques, M.A.A., 2017. Similarity digest search: a survey and comparative analysis of strategies to perform known file filtering using approximate matching. Secur. Commun. Network. 2017.

Nelson, A.J., 2016. Software Signature Derivation from Sequential Digital Forensic Analysis. UC Santa Cruz.

NIST. Diskprints. https://www.nist.gov/itl/ssd/software-quality-group/national-software-reference-library-nsrl/nsrl-subprojects/diskprints (accessed 2021).

Olsson, J., Boldt, M., 2009. Computer forensic timeline visualization tool. Digit. Invest. 6, S78–S87.

Ordonez, P., Armstrong, T., Oates, T., Fackler, J., 2011. Using modified multivariate bag-of-words models to classify physiological data. In: 2011 IEEE 11th International Conference on Data Mining Workshops. IEEE, pp. 534–539.

Porter, K., Petrovic, S., 2018. Obtaining precision-recall trade-offs in fuzzy searches of large email corpora. In: IFIP International Conference on Digital Forensics. Springer, pp. 67–85.

Quick, D., Choo, K.K.R., 2017. Big forensic data management in heterogeneous distributed systems: quick analysis of multimedia forensic data. Software Pract. Ex. 47 (8), 1095–1109.

Rajaraman, A., Ullman, J.D., 2011. Mining of Massive Datasets. Cambridge University Press.

Roussev, V., 2010. Data fingerprinting with similarity digests. In: IFIP International Conference on Digital Forensics. Springer, pp. 207–226.

Roussev, V., Quates, C., 2012. Content triage with similarity digests: the M57 case study. Digit. Invest. 9, S60–S68.

Sammons, J., 2016. Digital Forensics with the AccessData Forensic Toolkit (FTK). McGraw-Hill Education, p. 416.

Scanlon, M., 2016. Battling the digital forensic backlog through data deduplication. In: 2016 Sixth International Conference on Innovative Computing Technology (INTECH). IEEE, pp. 10–14.

Shaw, A., Browne, A., 2013. A practical and robust approach to coping with large volumes of data submitted for digital forensic examination. Digit. Invest. 10 (2), 116–128.

Soltani, S., Seno, S.A.H., Yazdi, H.S., 2019. Event reconstruction using temporal pattern of file system modification. IET Inf. Secur. 13 (3), 201–212.

Soltani, S., Seno, S.A.H., Budiarto, R., 2021. Developing software signature search engines using paragraph vector model: a triage approach for digital forensics. IEEE Access 9, 55814–55832.

Song, W., Yin, H., Liu, C., Song, D., 2018. Deepmem: learning graph neural network models for fast and robust memory forensic analysis. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 606–618.

Studiawan, H., Sohel, F., Payne, C., 2020. Sentiment analysis in a forensic timeline with deep learning. IEEE Access 8, 60664–60675.

Vidas, T., Kaplan, B., Geiger, M., 2014. OpenLV: empowering investigators and first-responders in the digital forensics process. Digit. Invest. 11, S45–S53.

Woods, K., Lee, C.A., Garfinkel, S., Dittrich, D., Russell, A., Kearton, K., 2011. Creating Realistic Corpora for Security and Forensic Education. Naval Postgraduate School Monterey Ca Dept of Computer Science.