3-23-2018

Digital Forensics Event Graph Reconstruction

Daniel J. Schelkoph

DIGITAL FORENSICS EVENT GRAPH RECONSTRUCTION

THESIS

Daniel J. Schelkoph, Capt, USAF

AFIT-ENG-MS-18-M-058

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT-ENG-MS-18-M-058

DIGITAL FORENSICS EVENT GRAPH RECONSTRUCTION

THESIS

Presented to the Faculty

Department of Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Computer Science

Daniel J. Schelkoph, B.S

Capt, USAF

March 2018

AFIT-ENG-MS-18-M-058

DIGITAL FORENSICS EVENT GRAPH RECONSTRUCTION

THESIS

Daniel J. Schelkoph, B.S
Capt, USAF

AFIT-ENG-MS-18-M-058

Abstract

Ontological data representation and data normalization can provide a structured way to correlate digital artifacts. This can reduce the amount of data that a forensics examiner needs to process in order to understand the sequence of events that happened on the system. However, ontology processing suffers from large disk consumption and a high computational cost. This paper presents Property Graph Event Reconstruction (PGER), a novel data normalization and event correlation system that leverages a native graph database to improve the speed of queries common in ontological data. PGER reduces the processing time of event correlation grammars and maintains accuracy over a relational database storage format.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank Dr. Peterson for his advice and guidance on this project. Additionally, I would like to thank Dr. Okolica for his efforts in creating the TEAR dataset and assistance in incorporating machine-generated abstraction into PGER. Finally, I would like to thank my family who provided support for me in order to complete the project.

Daniel J. Schelkoph

DIGITAL FORENSICS EVENT GRAPH RECONSTRUCTION

I. Introduction

With society's ever-increasing reliance on technology, the demand for digital forensics has risen significantly. This is noted in the 2017 Bureau of Labor Statistics ten-year job outlook figures for the related fields of Forensic Technicians (up 17%) [6] and Information Security Analysts (up 28%) [7]. This need is, in part, driven by the time-consuming task of manual data correlation required for digital forensics investigations [12]. Forensics research tries to provide data correlation tools for examiners that reduce the time needed to process each case. Any forensics tools must focus on solutions for three main challenges: data volume, heterogeneous data, and legal requirements [12].

Data volume is a serious problem for examiners [12]. Not only have storage capacities risen, but consumers are now also likely to own multiple digital devices. A study from the Pew Research Center states that smartphone ownership has risen from 35% in 2011 to 77% in 2016, and tablet usage, typically used as a secondary device, has increased from 8% in 2011 to 51% in 2016 [36]. The large amounts of forensic data provided by multiple devices presents an overwhelming amount of data for examiners to process [12]. Automated tools are needed to combine events together or eliminate unimportant events.

Heterogeneous data complicates data processing [12]. Many types of forensic artifacts contain unique data that is important for a forensics investigation. For example, a New Technology File System (NTFS) file table contains creation, access, and modification times where a registry key contains a value. This can lead to disparate formats for different types of data, requiring more work from examiners during correlation [12]. Developed tools need to normalize heterogeneous data into a standard format, allowing easy correlation for either examiners or computers.

The last major challenge is meeting legal requirements. Any results from an automated tool must be verifiable by digital forensics experts and explainable to a court. This means the automation must provide transparency into how correlations were established. Further, the methods used to create the correlations should be credible and rely on a formal model [12].

1.1 The Digital Forensic Process

The National Institute of Standards and Technology (NIST) highlights the four main phases of digital forensics investigations: Collection, Examination, Analysis, Reporting [25]. Figure 1 presents these phases and indicates important topics in a specific phase to the right. Each step is explained in further detail below.

The collection phase deals with the physical acquisition of digital forensics artifacts like computers, cell phones, or logs. Steps should be taken to preserve and ensure the integrity of digital data. The National Institue of Justice provides a good starting point for digital forensics collection [15].

The Examination phase processes forensic artifacts. This involves manual or automated methods to extract data of particular interest [25]. Both methods can represent data in many different formats that affect how data is analyzed. As with all steps, integrity of the data must be preserved [25].

The third phase, analysis, uses legally justifiable methods to extract useful information from the data obtained in the previous step [25]. This analysis provides answers to questions posed by the investigation [25]. Event reconstruction is one way of accomplishing this step and attempts to understand the sequence of events on a

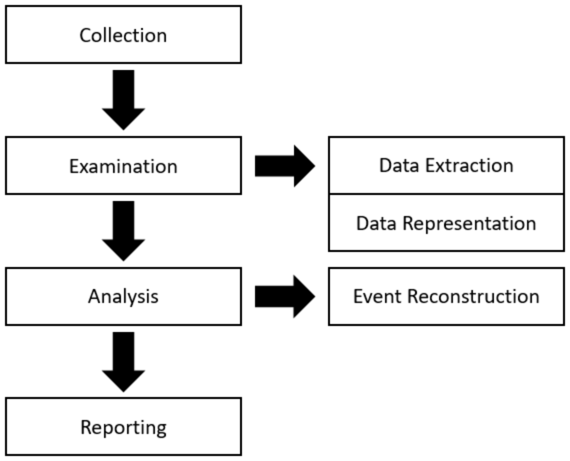

Figure 1. NIST Forensics Phases.

digital device.

The final phase, reporting, can have varying requirements, but it generally includes the following: investigative actions performed, reasoning behind tool section and procedures, the results of the analysis, and further actions that can be performed [25].

1.2 Problem Statement

Property Graph Event Reconstruction (PGER) examines and attempts to solve two challenges while maintaining legal admissibility: the storage of event data, and event abstraction.

Current research has determined that an ontological approach is an effective way to handle the problem of heterogeneous data [9]. This method creates standardized data structures for events, allowing for relationships between heterogeneous data sources. This creates data that is highly connected, representing a graph-like structure. However, this research stores event data based on a relational database standard for use on the internet, prioritizing the ability to split or shard the data onto multiple servers. This structure makes it more difficult to quickly query data based on the ontological relationships.

Forensic examiners need assistance to handle the overwhelming amount of data. Many different types of research have developed ways of abstracting portions of forensics data into higher-level events, giving examiners a smaller dataset to investigate. For example, AutoProv attempts to provide examiners the provenance of a file on the forensics image [19]. Any new forensics tool should be able to perform some sort of abstraction.

1.3 Hypothesis

PGER is a novel attempt to improve the storage of event data by using a native graph database for storage. By using this storage method, PGER should able to query event data faster by leveraging the established relationships. It also simplifies query construction for users, allowing for intuitive traversal through the graph to find specific patterns. PGER also attempts to provide two methods for abstracting data. The first is to use expert knowledge from forensic investigators to create rules that identify patterns in the graph. The second method uses rules provided by machine generated patterns to find higher-level events. Allowing for multiple types of event abstraction, PGER aims to be a flexible forensics tool. Performance

1.4 Assumptions

There are several assumptions PGER uses to operate. First, PGER is currently configured to process Windows forensics images only. Next, it is assumed that all timestamps are accurate and not maliciously modified. Finally, any abstraction generated by either method only serves to provide guidance to examiners. Abstracted events may represent false positives or false negatives and need to be verified by examiners to ensure legal admissibility.

1.5 Result

Designed in a layered format using the visualization tool docker, PGER is flexible and allows for multiple methods of data input, processing, and abstraction. PGER is able to provide an ontological database for events in a native database format, providing query simplification and speed improvements for common queries. It also provides runtime improvements to machine-generated ruleset abstraction while maintaining the accuracy of Temporal Event Abstraction and Reconstruction (TEAR). Expert rules can be applied to the a forensics image, providing useful information for the examiner.

1.6 Conclusion

Forensics examiners face three main challenges: data volume, data heterogeneity, and legal admissibility. The four NIST phases help provide guidance on how to produce legally admissible results. Research has shown that ontological storage of event data provides rich results for examiners by creating relationships between events, but that data is stored in a format that does not quickly query relationships. PGER's novel approach utilizes a native graph database that takes advantage of the ontological relationships for faster performance. PGER also allows for two types of abstraction to help examiners handle the immense data volume of modern forensics images. This research tests the effectiveness and processing speed of PGER compared to relational database solutions.

II. Related Work

Reconstructing events from digital artifacts can be broadly categorized into three steps: data extraction, data representation, and event reconstruction. Research shows that there are many ways to accomplish each step. Research into event reconstruction, for example, has attempted to use finite state machines, machine learning, and inference rules. The most appropriate methods for PGER must be chosen for the tool to be successful. The method of event storage is also important and can drastically impact performance. Native graph database operation is critical to understanding why neo4j was selected as the storage medium.

The first three sections examine current research in the steps listed above. The last section discusses graph databases and some of their inherent advantages in certain circumstances.

2.1 Data Extraction

After forensic evidence is gathered in the collection phase, event data must be extracted to perform analysis. Although data extraction can be performed manually, the time requirements are too great and automated tools are generally used instead. Automated tools can be broken down into four categories: Single Data Source, Supertimelines, Investigative Tools, and Pre-Event Collection.

Single Data Source.

Many topics of digital forensics research focus on how to extract data from one area of interest on a target machine. These techniques can be very specific, focusing on a small set of data for a certain operating system. For example Alghafli, et al. [1] extract details like network connectivity and most recently used files of the Windows 7 registry through their developed tool. Other research focuses on certain applications like web history for Firefox [35] or download history [43]. Specific data collection tools can be valuable to examiners, but they suffer isolation from other data sources [41] and cannot easily form correlations between disparate extracted datasets. This forces examiners to use multiple tools and make correlations manually. Despite the lack of interoperability, however, these single data source tools do often highlight cutting edge research and are adopted into other data collection techniques that provide a standardized data representation.

Supertimeline.

Supertimelines utilize multiple tools, called parsers, and combine the results into a standardized format [21]. log2timeline is the best recognized tool for the creation of supertimelines. Expanding on ideas from zeitline [5], it stores heterogeneous data ordered by time, allowing examiners to group events temporally. It was originally written in perl [21], but has been ported to Python and is now called PLASO (Plaso Langar Aő Safna Öllu) [12]. PLASO still receives regular updates from the community and contains hundreds of different parsers. Parser highlights include file metadata, web history, link files, prefetch, sqlite, recycle bin, system event logs and Windows registry entries. Many of these parsers are modified versions of single source tools such as regripper for registry files and the sleuth kit for file table entries.

The wide variety of information provided can provide context for examiners but also creates another problem: overwhelming data [12]. Supertimelines contain a lot of data, making it harder for examiners to find essential information [12]. This extra data necessitates a lot of manpower from examiners to find relevant data. Additionally, supertimelines are only matched temporally to each other. This means a search is required to find all entries pertaining to a specific file. Furthermore, the standard

17 field format for log2timeline includes useful information unique to a specific event type in the "message" field. For instance, registry keys may have the filename of a most recently used document in the message field. As another example, the target of a link file is also contained in the message field. Because of varying data, information in the message field requires additional parsing or examination to extract, adding difficulty for examiners if supertimelines are the only tool utilized. To be truly effective, supertimelines need to extract data that depicts relationships between events.

Investigative Tools.

Forensic tools like EnCase, Autopsy, The Sleuth Kit (TSK), and The Forensic Toolkit (FTK) provide a graphical interface to examine evidence on a forensic image [5]. These programs focus on recovering data from the filesystem, particularly hidden or deleted data [5]. Investigative tools index all files and allow users to search by keywords, but additional analysis is limited to web history or recent files [5]. Using investigative tools to determine a sequence of events is time intensive, forcing examiners to piece together a sequence of events and likely use other tools to help corroborate these events. Further, most of these tools store data in proprietary or non-standard ways, preventing data from being combined with other sources. Supertimelines provide a much richer picture of the temporal events on a computer, but the keyword searches of investigative tools rapidly identify evidence for examiners.

Pre-Event Collection.

Some data collection tools utilize a constant logging and monitoring of system events, not just a snapshot of a disk image. Collecting data in this fashion can be costly on resources, but it can provide a richer sequence of events for examiners, allowing them to 'step-through' the sequence of events [18]. These systems also require collection tools to be in place before an incident occurs [30], making the use of these tools very dependent on their implementation by system administrators. Outside of a corporate environment, this type of system would be rare, limiting its usefulness [18].

2.2 Data Representation

A common problem with the data collection techniques is the inability to handle heterogeneous data. Single data source and investigative tools typically have a unique output format that is not easily combined with other sources. The best known super-timeline, PLASO, provides a common format for all events, but hides rich data into the catch-all field of ‘message’, requiring additional parsing to understand. A common data representation for events is critical to establishing relationships between heterogeneous data and creating a richer set of events for the examiner.

Visualization.

Displaying events on a visual time line is one method that makes data processing easier for examiners. This technique sorts events by a timestamp field and displays events on an interactive timeline. By providing a temproal context between events, this method has the ability to speed up investigations that only use investigative tools similar to FTK [33]. With the timeline, it is easy to see the order of events or how one event affects another. However, just like a supertimeline, visualizations can quickly overwhelm examiners, and steps must be taken to limit the number of events displayed. Zeitline, for example, allows users to create complex events, grouping many events into one [5]. Still, while visual timelines can help speed up investigations in certain cases, they do not help with automated event reconstruction, limiting their impact on this research.

Ontological.

Several recent forensic researchers have leveraged an ontological approach to representing events. This approach uses well-defined definitions (semantics) for objects, properties of objects, and relationships between objects. The goal of this representation is to organize data in a consistent way that computers can understand [41]. Further, it allows for experts to design rules that allow for inferred facts based on the presented knowledge [41]. For example, Google uses an ontology for its knowledge graph, allowing for richer data to be displayed in search results [16]. Typing the name of a music album will present the name and other biographical information of the artist, as well as display similar artists. This works because many different websites are providing information using the same definitions. Ontologies can be stored in many different ways, but these models will be discussed in another section. The forensic ontologies important to this research are discussed below.

ECF.

Event Correlation for Forensics (ECF) [13] contains a simple ontology consisting of a source, an action, and a target as an atomic event. The source is the object (program, system process, etc.) that causes an action, the action is a description of the activity, and the target is object that is affected by the source [13]. This style allows events to depict cause and effect relationships between objects in an image. An example would be an executable program that changes a .txt file. The event-based entry would describe the source as the executable, the action as ‘Changed File’, and the target as the .txt file; this creates a visible relationship between the executable and the .txt file that is not apparent in a standard supertimeline format.

SOSLA.

In the Single Object String and Loops Abstraction (SOSLA) [32], each event consists of a start and end time along with an activity [32]. The activity is made up of an operator (source), an action, and a set of objects (targets). This varies from the ECF format in that there can be multiple objects affected in one event. This makes it possible for SOSLA to combine many low-level events into one. If an antivirus scans all files in a system, it could create a large number of similar events called a loop. With SOSLA, these activities can be consolidated to one event, using the antivirus as the operator and all the affected files as targets. In addition, SOSLA can merge multiple diverse events that affect one object into one event. For example, if there are create, access, modify, and change time events for the same file, and they all happen at the same time, the events could be combined into one event with an action of 'File Created' [32]. These abstraction techniques can significantly reduce the number of events a supertimeline represents.

FORE.

Forensic of Rich Events (FORE) [39] was one of the first digital forensics tools to use the Web Ontology Language (OWL) to define its semantics. The basic ontology consists of two main categories: entities and events. Entities are similar to objects in the ECF/SOSLA examples; they are tangible objects in the forensic image [39]. The entity can associated properties like username and domain for a user ID found in the system. Events describe changes in state, for instance, downloading a file [39]. Events map changes to entities in order to track changes. More current research has pointed out some failures of this model [11], insisting that the semantics are not detailed enough to accurately describe all forensic events. However, it does serve as a starting point for the next ontology.

DFAX.

Digital Forensics Analysis Expression (DFAX) [9] is an attempt to include law enforcement requirements in a forensic ontology. Its mission is to create an ontology so law enforcement agencies can easily share digital forensics information. As such, the ontology details more than just the events of a forensic image. It contains semantics for case numbers, attorneys, investigators, subjects, victims, and other case information [9]. As a subset of this ontology is another ontological language to describe digital observables called Cyber Observable Expressions (CybOX). This language is an open-source model with development led by the Department of Homeland Security and MITRE [9]. CybOX uses a top level designation of 'Observable' to describe a series of events. Events consist of metadata and actions. Actions are the base-level event and consist of extracted properties and other metadata. The strength of this model is its flexibility to describe many different artifacts. For instance, CybOX has different models for emails and registry entries and can create additional semantics for more data types [12]. Since publication, it has merged with Structured Threat Information Expression (STIX) [14].

ORD2I.

The Ontology for the Representation of Digital Incidents and Investigations (ORD2I) draws inspiration from CybOX and the Provenance (PROV-O) ontologies [12]. PROV-O is a W3C recommendation similar to the law enforcement semantics included in the DFAX ontology [28]. ORD2I consists of 3 main layers: specialized knowledge, common knowledge, and traceability knowledge. Traceability knowledge incorporates the PROV-O standard, providing semantics for the examiner, tools, and case number [12]. The common knowledge layer includes information that every event possesses, such as a physical and/or virtual location, event name, and an object. The specialized knowledge layer incorporates the CybOX ability to distinguish artifacts from one another. This gives more fidelity to the object created in the common knowledge layer [12]. ORD2I is very close to the DFAX ontology, but it is part of a larger tool to help reconstruct events.

2.3 Event Reconstruction Techniques

Due to the ever-increasing amounts of data forensic examiners have to analyze, a focus of digital forensics research is to try and create an automated way to consolidate data and reconstruct events. In addition, many forensic investigations are for legal proceedings with strict rules regarding evidence, so it is desirable that event reconstruction be the result of a formal theory [17]. Previous research has tried many different methods to establish such a theory: finite state machines [17], machine learning [26], and inference rules[39].

Finite State Machines.

Finite state machines provide a mathematical foundation for forensic events, providing rigor to findings [17]. Finite state machines are constructed by working backwards from the final state and using expert knowledge to make transitions and states, eventually ending with the events that need to happen before the final state can be accomplished [24]. Unfortunately, the number of possible variations from each final state produce very large finite state machines and are hard to create by experts [10]. At times, the finite state machines may be shrunk by previous events or other evidential information [24]; however, this is currently a manual task and limits the usefulness of this process [10].

Machine Learning.

Expert-created event patterns can be complex and time-consuming to create. Some researchers have tried using machine learning to automatically find patterns in data. In 2006, researchers proposed a neural network that found the execution of various sequences in Internet Explorer [26]. This neural network was able to reconstruct events with an accuracy of 90%. However, neural network techniques do not show how low-level actions are associated with other events to infer a high-level activity [10]. Since examiners were not able to explain why the neural network created a certain series of events, this technique is hard to use in evidential situations [12].

TEAR.

Temporal Event Abstraction and Reconstruction (TEAR) is another attempt at using machine learning [31]. Unlike neural networks, this method of pattern matching allows humans to certify the identified patterns and trace the high-level events to the individual low-level events. Its algorithms create a hierarchy of events using pattern matching in order to represent a high-level event (See Figure 21) [31]. At the lowest level are terms which represent an atomic event such as a registry key modification. Each term has an action, a type, and a regular expression to determine what events get a specific term label. A term representing a created file in a user's document directory would have an action of 'Created', a 'file' type, and a regular expression of '^.*/Documents/.*$'. The next level on the hierarchy is strings. Strings are composed of other strings and/or terms. Next, production rules consist of both terms and strings, and provide a successful pattern for a high-level event. This helps represent multiple paths to the same high-level event. If a production rule with no parent finds a match in the data, then the high-level event occurred.

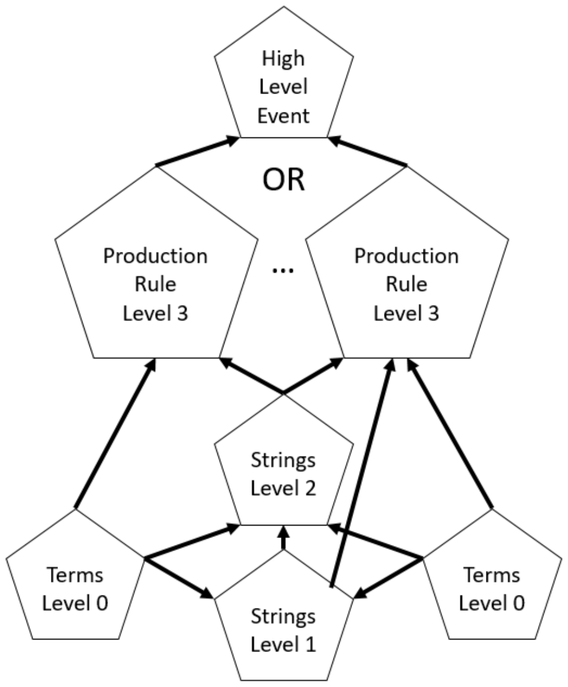

Figure 2. Temporal Event Abstraction & Reconstruction Hierarchy.

Inference Rules.

Inference rules allow forensic tools to reconstruct events by applying rules to existing information from the device image. This method performs similarly to Expert Systems; some tools even use an Expert System directly [39]. The rules are used to find patterns in existing data and, if a rule matches a pattern, a reconstructed event is created. For instance, an inference rule could look for events that contributed to the insertion of a USB device. If the rule triggers on the appropriate registry entries, it could create a new reconstructed event stating that the USB device was inserted. In order for this technique to work, examiners must have well-structured ontological data because the applied rules rely on the semantics of the data [10]. In research, the FORE tool uses the expert system in the Jena Apache Framework [39]. The Semantic Analysis of Digital Forensic Cases (SADFC) uses the ORD2I ontology and queries on the dataset as its rule set. If a query matches a set of data, they are combined to form a reconstructed event [10]. A tool called Parallax Forensics (ParFor) uses its own tool to query data and functions similarly to SADFC [41].

Data Correlation in Non-Forensic Fields.

Establishing correlations in data is an important problem outside of digital forensics as well. Expert systems that use inference rules are used extensively to solve this problem in the medical field, and some of the established techniques have direct applicability to digital forensics. A survey paper written in 2007 for real-time temporal abstraction states that ontological approaches can lead to better results but are computationally difficult to perform in real-time [40]. However, post-event digital forensics does not always have such a time constraint and provides further credence to the ontological approaches above. The following paragraphs list other techniques that utilizes inference rules and/or ontological data schema.

Some expert systems try to combine machine learning with expert knowledge to create a more adaptable expert system. Sequence Clustering-Based Automated Rule Generation (SCARG) does this by establishing initial rules using expert knowledge, and as data is analyzed, machine learning alters the rules [29]. This technique provides a significant advantage over traditional expert systems, recording about a 20% classification improvement in the paper's test case. Since the changes in rules are visible, SCARG could be applied to digital forensics while still maintaining legal acceptability.

Research on genetic sequencing provides further proof that normalizing data is essential to utilizing inference-based systems [4] because genetic sequence alignment uses heterogeneous data. This research first normalizes the data into common classes and then applies expert rules to the refined data.

Another validation of the ontological approach is GIDL (GenBank Intelligent Data

Loading) [34]. This program normalizes data from several different genetic databases into a standard ontology and applies expert system rules to develop correlations. This system uses both OWL (Ontological Web Language) and CLIPS (C Language Integrated Production System) to generate new facts. Additionally, it is built from the ground up to be distributed if large datasets require additional processing power.

Finally, there are two other main types of expert systems commonly cited in research, probabilistic and fuzzy. Probabilistic expert systems use Bayesian belief networks and a concept of entropy to establish correlations [23]. This does not appear to be an effective mechanism for digital forensics due to the effort required to build the extensive belief network required. Fuzzy expert systems rely on predetermined membership functions that determine the likelihood of event B happening if event A occurs [27]. The huge variety of digital forensics data limits the potential of this technique, as the number of membership functions required is very large.

2.4 Graph Databases

A graph database allows the user to interact with data as nodes and edges. This is much different from a traditional relational database that represents data as tables. Using a graph representation can result in performance benefits for connected data, but it is highly dependent on how the graph database model is constructed [37]. Each graph database model is on two spectrums: the data format and the processing method (Figure 3) [37].

Although all graph databases represent a graph, each graph model has a data format on a spectrum between non-native and native [37]. Non-native storage converts nodes and edges to relational database tables or another format (e.g. document-based). This can be useful if the database is large, allowing for a straightforward way to shard the data [37]. Sharding allows the data to be distributed from multiple

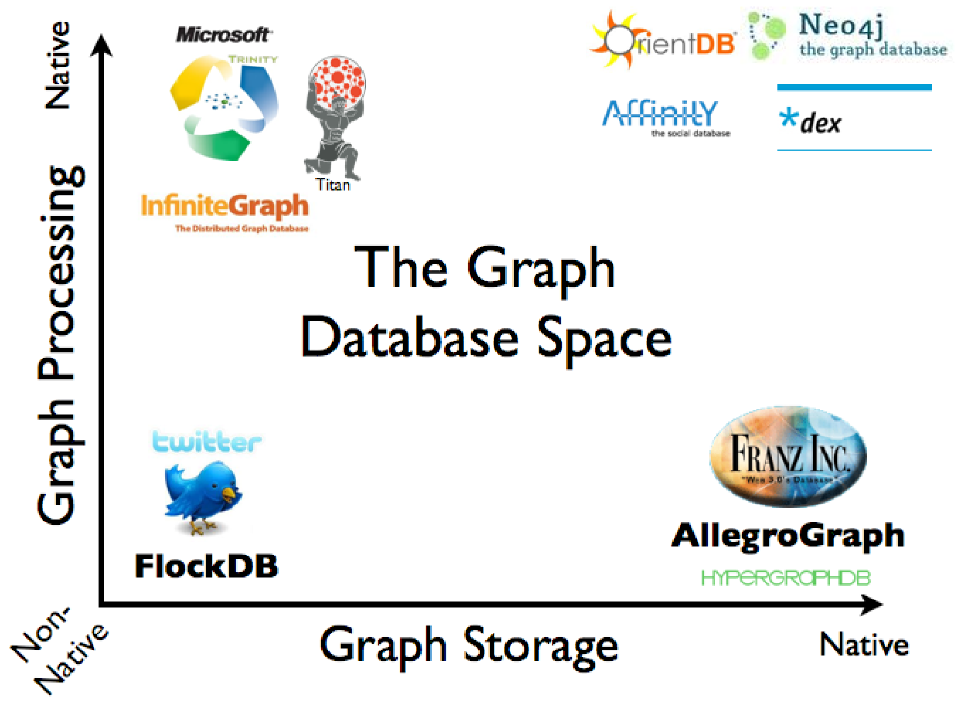

Figure 3. Graph Database Spectra [37].

servers. For native storage, the graph is the storage mechanism. This can provide performance benefits for certain queries, as the database does not have to construct the graph before processing [37].

Graph databases also have processing method on the spectrum between non-native and native. This is how the graph handles all Create, Read, Update, Delete (CRUD) operations [37]. Non-native processing does not use a graph to conduct an operation. Instead, it deals directly with how the data is stored. This style can utilize performance benefits like indexes from relational or document databases [37]. Native processing utilizes a graph to perform CRUD operations, providing performance benefits unique to graph databases [38].

One of the biggest performance benefits for native graph processing is the ability to perform an index free traversal [38]. In highly connected data, it is often useful to examine the relationship between data. To determine if two pieces of data share a relationship, a traditional database (non-native processing) would need to perform a join on multiple tables. This would require at least two index-based searches with a runtime of $O(\log_2 n)$ [38]. To determine more complex relationships, like finding if a particular walk on a graph exists, the queries become even more complex and time-consuming. Native graph processing allows a search on related data (data incident to another piece of data) in constant time, otherwise known as an index free traversal [3]. Searching a graph for a particular walk can be much faster by processing in a native format.

There are advantages and disadvantages to utilizing certain choices on the spectrum. Below are the most common graph database models and their attributes.

Triples.

Triples is a non-native storage model where an atomic entry consists of a subject, predicate, and object [42]. The most popular version of this storage method is Resource Description Framwork (RDF) [2]. RDF is a World Wide Web Consortium (W3C) standard that outlines the structure of the subject, predicate and object [22]. The Ontological Web Language (OWL) is another W3C standard that sits on top of RDF and describes how semantics are defined [8]. Most ontological digital forensics research uses RDF and OWL to store event data. FORE, DFAX, and ORD2I are all stored using these standards. Since this is a non-native storage model, the previous examples use relational database tables to store all triples. This means that these databases do not have the ability for index-free traversal, making queries regarding graph structure time consuming [2]. However, due to its use of established standards, one of RDF's strengths is the ability to easily share information with other systems. Another strength is that its non-native storage allows for sharding of the dataset between multiple servers.

Labeled Property Graph.

Labeled Property Graphs are typically native storage and processing models. Data in the graph is created by inserting nodes and edges. One major difference between this model and triples is the ability to store data in nodes, allowing for more compact graphs in certain instances [2]. This model also allows for index-free traversal of the graph, allowing queries to take advantage of relationships between nodes for rapid queries [2]. In fact, the main use of labeled property graphs is for rapid transactions [2]. neo4j is one of the leading databases of this type.

2.5 Summary

The first section of this chapter detailed the main challenges in creating an event reconstruction tool for digital forensics. PGER must solve data volume, data heterogeneity, and legal requirements challenges of digital forensics. The biggest challenge of data extraction and data representation is to provide a consistent model for event reconstruction. Ontologies are the main way this is performed in current research. In regards to event reconstruction, inference rules were the most common solution, utilizing expert-created rules. Finally, the main types of graph databases were detailed, stating that index-free traversal is a key benefit of Labeled Property Graphs.

III. Property Graph Event Reconstruction (PGER)

Property Graph Event Reconstruction (PGER) was created for this research to perform abstraction of user actions from digital media. It utilizes a native labeled property graph to store event data and uses an ontology-based storage method while earning performance gains through the index-free traversal of native graphs. Unlike other ontological approaches in Chapter 2, users can directly query event data by ontological relationships without the cost of constructing a graph or completing numerous join queries. By discussing events in terms of types and relationships, queries become easier to understand and create. It also allows users to leverage quick path searching and utilize index-free traversals to quickly find subgraph patterns in event data. PGER is a combination of several tools, some were created specifically for PGER and others were re-purposed for PGER. The origin of the tools used in PGER is in Table 1.

PGER accomplishes these tasks in four processing layers (Figure 4). The first layer takes a device image and extracts events. The second layer converts the extracted events into ontological subgraphs stored in neo4j. The normalization layer ensures identical objects are represented by the same object. Finally, the abstraction layer

Table 1. PGER Tool Origins.

PGER Step	Tool Name	Existing Tool	Created
Data Extraction	PLASO	X	
	TEAR Event Extractor	X	
Graph Conversion	Logstash	X	
	Logstash Parsers		X
	Python Script		X
Normalization	Normalizer		X
Abstraction	Expert Rules		X
	Application of TEAR Ruleset		X
	TEAR Ruleset	X	

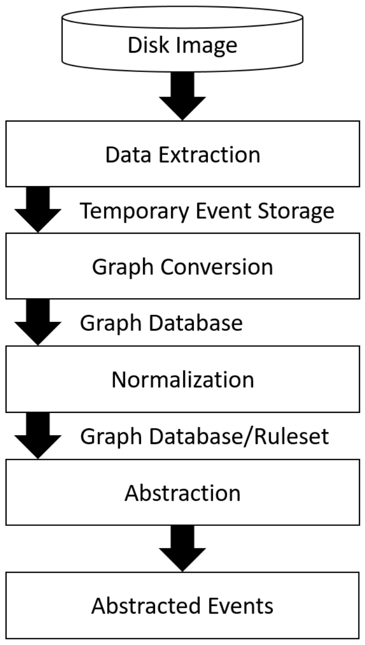

Figure 4. PGER Processing Layers.

uses either expert rules or a machine-generated ruleset to extract higher-level events. The next section discusses the PGER environment configuration, followed by details on each component.

3.1 Environment

Each forensics image is represented by a separate neo4j database and is created or enhanced by PGER in several steps. Each of these steps is designed as atomic entity, allowing for independent operation. This design isolates versions of a database during testing or multiple cases so they can be processed in a pipeline-like fashion. Docker is the key technology that allows for this separation and is described below. The file structure is also detailed to help explain PGER execution.

Virtualized Containers.

Docker, a level 2 virtual machine, is core component of all the processing layers. A user can run self-contained programs called a containers on any machine that has the docker software installed. This allows PGER to run on any machine with a limited set of requirements and dependencies. It also allows users to overcome neo4j's limitation of operating one graph database per instance by easily spawning new instances. Containers are obtained by downloading from the docker repository. The two most common containers in PGER, neo4j and python, have existing images in the repository. If changes to a container are desired, a dockerfile can be used to add additional features. For example, plugins for neo4j are installed in a dockerfile for PGER to operate.

Several configuration options are available for containers, but the most important options are networks and volumes. Multiple containers can communicate with each other by setting up isolated networks and can communicate outside of these networks by exposing ports to the host machine. Any data created when containers are running is not permanent unless it is stored in a mounted data volume. Data volumes can be created by docker for use only by containers or can be shared between container and host by mounting a host directory to a specific container. Docker container parameters such as networks and data volumes can be configured using a YAML (docker-compose.yml) file if the docker-compose program is used. Using this configuation file allows for multiple containers to be run using one command.

File Structure.

PGER has the following file structure:

```
PGER
├── installPrereq.sh
└── docker
    ├── initialSetup.sh
    ├── dockerComposeEnv
    ├── commonCode
    │   └── startup
    ├── commonDockerfiles
    │   └── <container type>
    │       └── <container variant>
    └── <processing layer>
        ├── <container name>
        │   └── <container files>
        ├── .env
        ├── startup.py
        ├── config.yml
        └── docker-compose.yml
```

installPrereq.sh This file is used when PGER is first run on a debian-based linux machine. It installs docker, docker-compose, and python pip, as well as increases the virtual memory address space for the elastic stack (elasticDB, Kibana, Logstash).

docker The folder that contains all the processing layers of PGER, with each layer using atomic docker containers.

initialSetup.sh A script that updates the host's version of python pip, installs necessary python packages, builds docker containers and starts elasticDB and Kibana.

dockerComposeEnv A text file that contains environmental variables for docker-compose.yml files in each processing layer. Contains the elastic stack version used and the location of the data directory.

commonCode A folder that contains the python code that is used in all processing layers.

commonDockerfiles A folder that contains dockerfiles that are used in all processing layers. neo4j and Python are examples of container types. neo4j_plugins and neo4j_no_plugins are examples of variants.

Processing Layer Each processing layer contains this file structure. Each docker container used in a processing layer has its own separate folder that contains the files necessary to run. For example, a neo4j folder will have the neo4j configuration file and a python folder will have the scripts that execute the processing layer.

.env A symbolic link to dockerComposeEnv.

startup.py To ease the execution of each processing layer, a python startup script is used to configure and start the appropriate docker containers. This script allows users to interface with a simple command line program instead of needing to understand the appropriate docker commands. This script also sets neo4j settings that cannot be configured directly by a docker-compose configuration file. This python script accomplishes the following tasks:

- Accepts layer specific arguments
- Finds location of neo4j database on host machine
- Configures neo4j to use the selected database
- Starts neo4j database and waits until it is running
- Starts python script responsible for processing layer

For configuration options that do not commonly change, such as the URL of a neo4j instance, a configuration file is used. This is a YAML file that is loaded at the beginning of the startup script.

In addition to PGER, a data directory located anywhere on the host machine is necessary to store database and forensics image information. The file structure of this directory is as follows:

```
<Data Directory>
├── elastic
├── neo4j
├── eventData
├── images
└── rulesets
```

elastic/neo4j Folders that store database information for the elastic stack and neo4j.

eventData A folder that stores TEAR's generated event files for a forensics image.

images A folder that contains device images for processing.

rulesets A folder that stores the TEAR rulesets.

3.2 Data Extraction

The first processing layer takes a device image and converts it into an intermediate format. This format can then be converted to a graph database in another processing layer. There are two ways PGER can create this intermediate format: PLASO and TEAR event extraction (Figure 5).

PLASO converts a device image into a supertimeline and outputs the resulting data to an elastic database. PLASO uses two commands to complete this process.

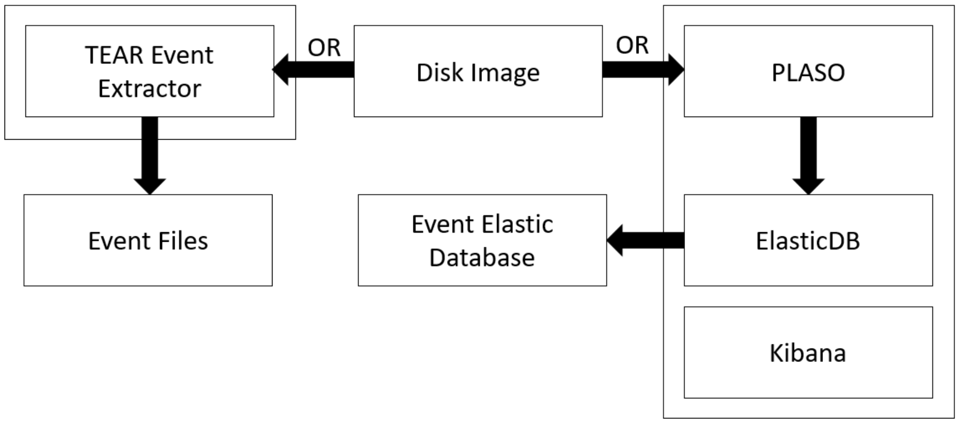

Figure 5. Data Extraction Layer.

The first command, log2timeline, extracts events from a device image and creates a datastore unique to PLASO. All partitions and volume shadow copies are analyzed. The second command, psort, converts the PLASO data store into an elastic database.

The other method of data extraction is utilizing the TEAR event extraction. It is a C++ program that takes the device image and creates a series of event files. File table, registry, and Windows events are among the data sources that are stored as CSV files. Web history is stored in a series of sqlite files.

3.3 Graph Conversion

Graph conversion transforms events stored in an intermediate format (either an elastic database or TEAR event extraction files), into a subgraph that is stored into a neo4j database. There are two methods for conversion in PGER (Figure 6): logstash and a python script. Logstash is a more flexible method of conversion, allowing for either intermediate format as an input and easy expansion of the analyzed events through its pipeline. However, Logstash suffers from a speed bottleneck when sending subgraphs to neo4j, as it must send each event as a single HTTP transaction, overwhelming connection limits for neo4j. The second conversion method provides bulk uploads to neo4j via a python script, significantly increasing conversion speed, but it is only currently configured to convert the TEAR event extraction. Regardless of the selected conversion method, different event types require separate processing.

Different event types, such as registry keys or prefetch information, contain unique details that provide additional insight on the forensics image. Examples include the values of a registry key or the time an executable was run. The processing steps required for an event are controlled by the identification of the event type. In PLASO, the event type is contained in the parser field, and in the TEAR event extractor, it is contained in the filename. The identified event type's processing steps are included

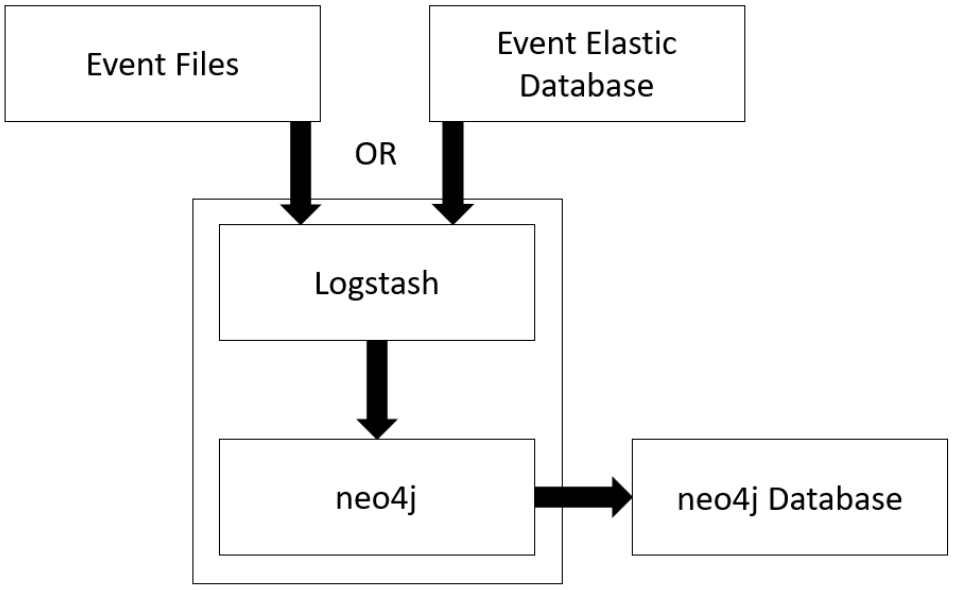

Figure 6. Graph Conversion Layer via Logstash.

in a parser. The parsers applied by the graph conversion programs are controlled by profiles in its configuration file (config.yml is the default). Each profile has a name, a type (blacklist or whitelist), and a list of parsers to include or exclude. If an event type is not included on the selected list of parsers, it is not processed. This allows users to control the events that appear in the neo4j database so they can focus on a small set of event types or remove event types that are not currently relevent to an investigator. A psudocode example of event processing is in Figure 7. The various types of subgraphs created during this process are in the next section.

Subgraphs.

The main purpose of the graph conversion processing layer is to convert heterogeneous events into a semantic, graph-based format. Every event creates a subgraph that is combined with the existing subgraphs in the neo4j database. The base of the semantic format is the core subgraph (Figure 8). This is based on the standard format found in SOSLA [32]. Every event contains this subgraph and consists of

```
#get profile information from processing layer configuration file
profileList = importProfile(<config.yml file>)

#create a function for the profiles included in the profile list
profileFunction = profileFactory(profileList)

#import the events from the intermediate format
#eventList is a list of dictionaries that are of the following
    format:
#{"Type": <event type as a str>, "Data": <varies between
    intermediate formats>}
eventList = getEvents(<intermediate format>)

#process each individual event
for eventDict in eventList:
        #eventData stores event information in a format for use
            in neo4j
        eventData = None

        #find the unique data of an event based on the
            profileList
        eventData += profileFunction(eventDict)

        sendToNeo4j(eventData)

#sample profileFunction
def profileFunction(eventDict):
        profileEventData = None
        if eventDict["Type"] is "mactime":
                profileEventData += mactimeParser(event["Data"])
        elif eventDict["Type"] is "prefetch":
                profileEventData += prefetchParser(event["Data"])
        <Additional elif statements as necessary>
        #if no matches in the profile, return nothing
        else:
                return None
        #get information common in every event
        profileEventData += getCoreSubgraph(event["Data"])
        return profileEventData
```

Figure 7. Event Processing Pseudocode

an action, an object, a parser, and a time. The time is a unix timestamp and is unique in the database. This represents the time that an action occured. The action contains a description of an action that affects a digital object. The object is the digital object that is affected by the action and contains an identifying name of an object such as a URL, registry key, or file path. The identifying name is unique in the

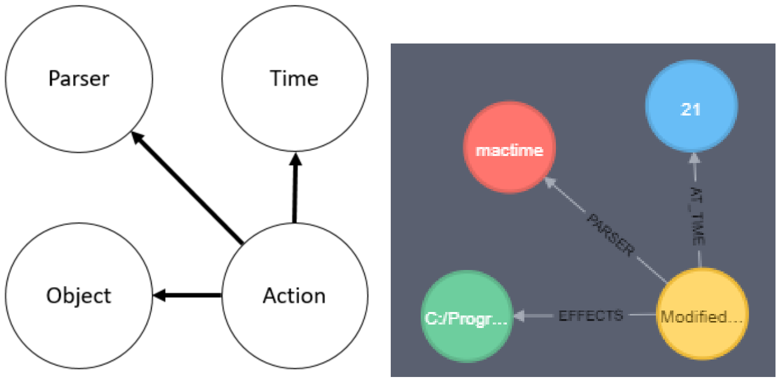

Figure 8. Core Subgraph.

database. Since objects are unique, they are also indexed by neo4j, providing speed improvements during queries. If there are different objects with the same name, such as registry keys in user hives, the username is appended to the beginning of the identifying name. The sections below show the most common types of subgraphs in a forensics image.

File Table.

The File Table parser produces at least one subgraph for each item in a system's file table. This subgraph contains an additional node to the core subgraph that represents the extension of the file object. The extension node is unique in the database. To help reduce congestion in the database, file table times (created, changed, modified, accessed) are combined into the same action if they occurred at the same time. The action description lists all the times that have changed during that timestamp ('Modified, Created Time Altered') and a new field is added to the action where each changed time type is an item in a list ("['Modified', 'Created']").

Web History.

Each web browser has its own parser due to differences in history recording. In general, a parser records three different events: history, downloads, and keyword

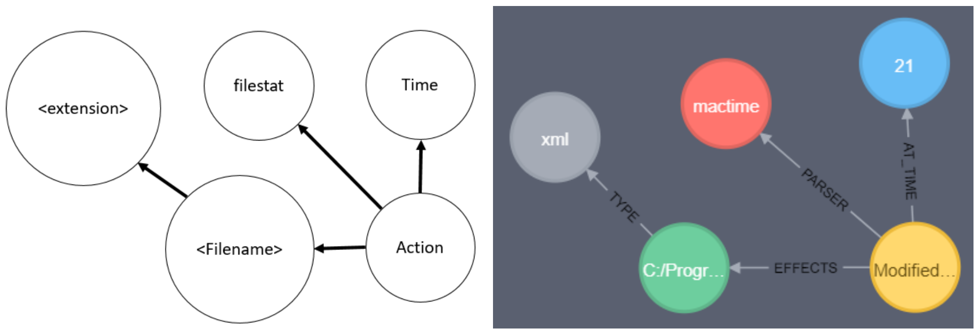

Figure 9. File Table Subgraph.

searches. Figure 10 details subgraph examples. History events adds a visit ID to the core subgraph. Visit IDs link to other visit IDs to indicate a sequence of events in a browser. Downloaded events show the location of the downloaded object and the URL source. Keyword search subgraphs add the core subgraph but include a field in the action node that indicates the words used in a search.

Registry Keys.

All parsed registry keys that are unique for each user (userclass.dat and ntuser.dat files) contain the subgraph in Figure 11. All other registry keys omit the user node in their subgraph. Both the registry type and owner nodes are unique in the database. The action node contains a field that holds the value of the registry key. Registry keys that provide more information are below.

Recent Apps (SOFTWARE/Microsoft/Windows/CurrentVersion/Search/RecentApps/) and User Assist (SOFTWARE/Microsoft/Windows/CurrentVersion/Explorer/UserAssist) registry keys help provide evidence of program execution. Both keys are updated when a program is run to populate recently used programs lists in Windows. These entries use the registry subgraph but add an additional object node with an edge to the action, indicating the program specific program that was executed.

Most Recently Used (MRU) registry entries (SOFTWARE/Microsoft/Windows/

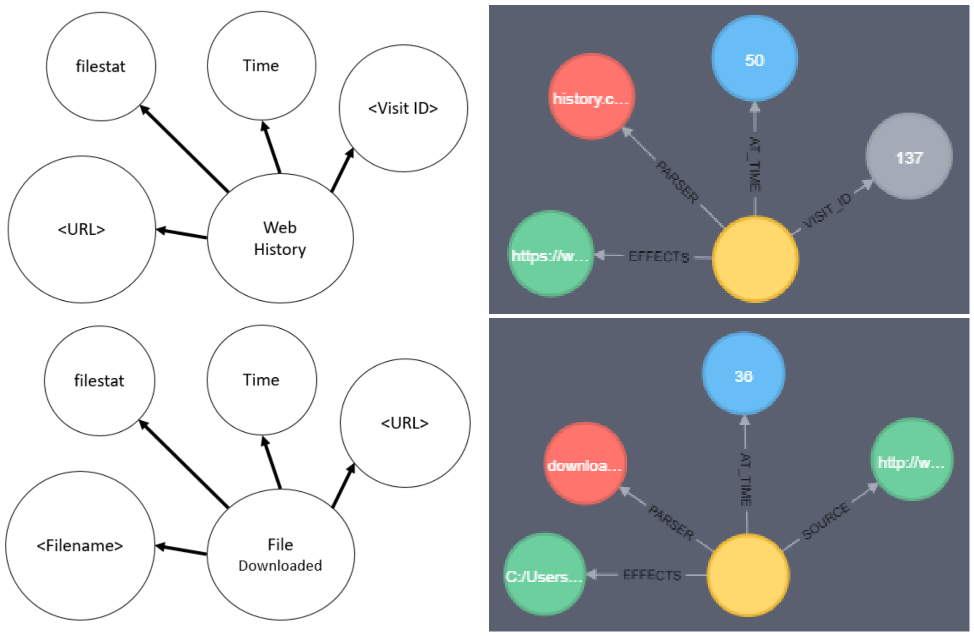

Figure 10. Web History and Download Subgraphs.

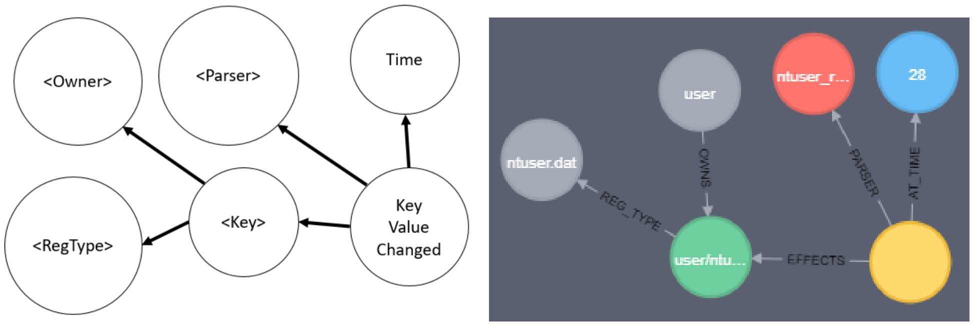

Figure 11. Standard Registry Subgraph.

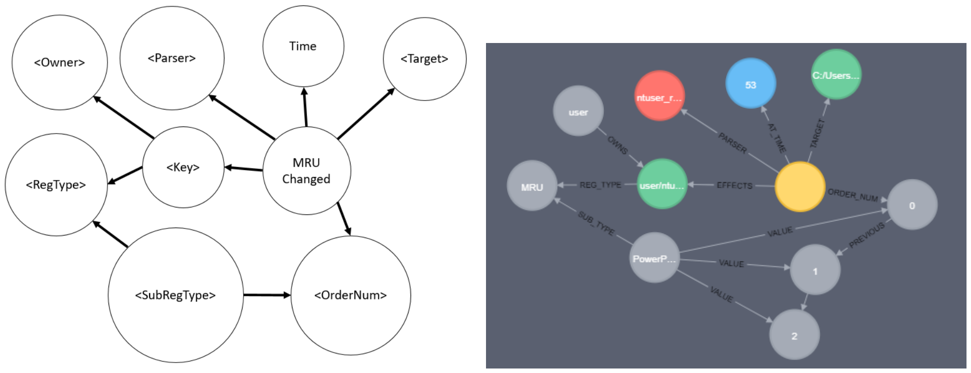

Figure 12. MRU Registry Subgraph.

CurrentVersion/Explorer/RecentDocs/) retain a list of recently used files for each extension (.doc, .jpg, etc). The recently used files for each extension are listed in ascending numerical order where 0 represents the most recently used file. The MRU subgraph (Figure 12) captures this data by creating a sub-registry type node that represents the most recently used files of a particular extension. In Figure 12, this is .ppt. The numerical order of an extension's MRU files are created as separate nodes to allow easy traversal of the files in order of use. There is also an additional object attached to the action node that was extracted from the value of the correct registry key and indicates the used file. The rest of the subgraph follows the standard registry subgraph.

Additional registry entries that can be processed are appcompatcache and shellbag keys. Like recent apps and users assist entries, appcompatcache helps provide evidence of program execution, updating registry values when a program is executed. Shellbag entries maintain UI information for folders viewed in the Windows File Explorer. These values can remain in the registry if the origial directory is deleted, possibly providing additional information on the filesystem.

Other Subgraphs.

Windows event log parsers produce the core subgraph for selected events. The action contains the event message and the object uses an identifying name of $\langle$ Event Type $\rangle$/$\langle$ Event Code $\rangle$. Windows events that are parsed include Customer Experience Improvement logon/logoff; event log service start/stop; computer uptime report; computer name changes; network interface connections/disconnections; wake/sleep states; installation result; and installer-initiated restarts.

The link parser analyzes all the link files in Windows and establishes a relationship between the link file and the target of the link file. This allows investigators to match events that affect a link with its corresponding target file. The link subgraph follows the same structure as the Downloaded File Subgraph in Figure 10, with the link file and the target file as the object.

Prefetch files store information to help load programs that have been previously executed. Prefetch files also include up to the last eight execution times, making them a valuable forensics tool. The parser creates an event for each of these execution times in the same subgraph style as the Downloaded File Subgraph using the prefetch file and the executable file as the two objects.

The mountpoints parser examines the addition or removal of removable media. The subgraph contains the device ID and the root of the file structure on the operating system. Again, this subgraph takes a form similar to the Downloaded File Subgraph.

neo4j.

Once neo4j receives data from the parser, it performs a transformation of the time node. Using the timetree plugin from GraphAware, these time events are converted into a time tree (Figure 13) [20]. The timestamp is divided into year, month, day, hour, minute, and second levels, and all actions are attached to the second nodes. This transformation allows queries that search consecutive events by following the next relationship. neo4j allows for queries that find variable length paths, so all events that occur in a X second window (starting at time Z) can be found with the following query:

```
MATCH (:Second {time: Z})-[:NEXT*0..X]->(window:Second)
MATCH (window)<-[:AT_TIME]-(act:action)
RETURN act
```

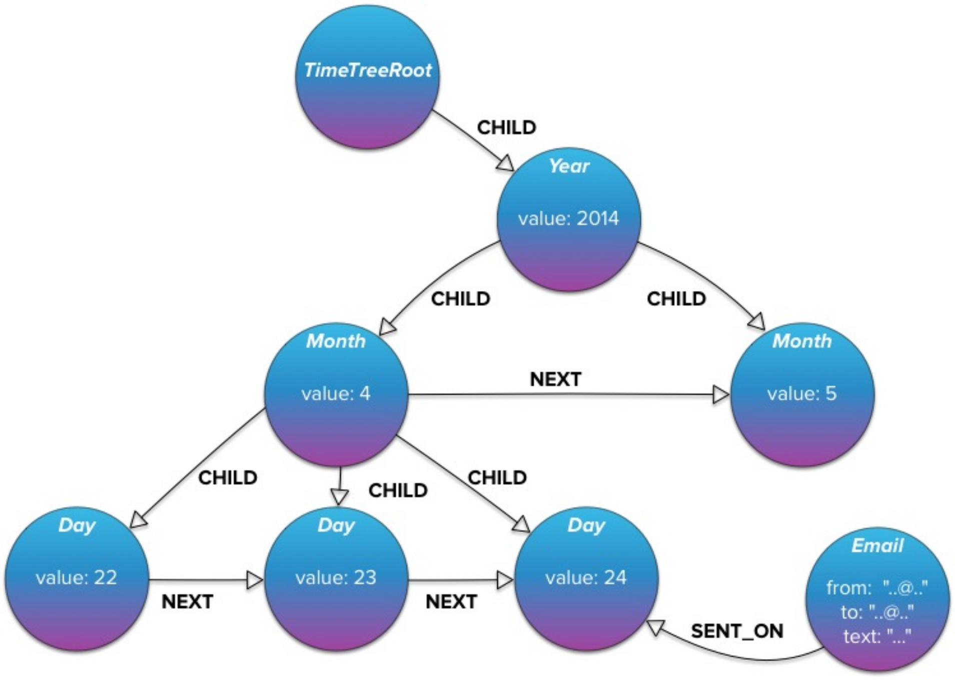

Figure 13. Time Tree Format [20].

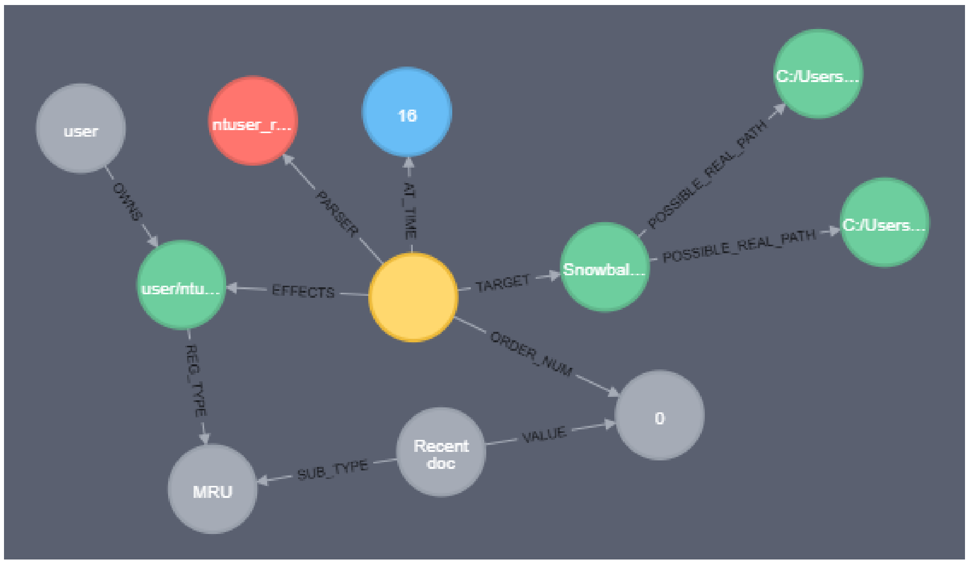

Figure 14. Normalizing MRU Entries.

3.4 Normalization

The next processing layer normalizes the neo4j database by combining objects that are the same but have a different identifying name. For example, MRU Registry Values only list the filename of the used file (Snowball_Fighting(1).doc) in an object node where the file table entry for the same file will be listed in an object node using the whole path (C:/Users/user/Downloads/Snowball_Fighting(1).doc). In order to properly see all the actions that happen on an object, these nodes need to be combined. The normalization processing layer combines nodes if there is only one match. If there are multiple matches, the node is linked to the possible matches with the relationship called 'POSSIBLE_REAL_PATH' (Figure 14).

3.5 Abstraction

Abstracting low-level event data into higher-level events is the final processing layer. Abstraction in PGER can be accomplished in two different ways: expert rules and applying TEAR rulesets.

Expert Rules.

Expert forensics knowledge can extract high-level events from a sequence of low-level events. For example, if there is evidence of a program execution, and a file is accessed with an extension associated with the executable within a certain time window, it is likely that the two actions are related. These known sequences can be quantified into rules to find specific subgraphs. Examples of these patterns can be found in the subsequent paragraphs.

History of a File.

A simple example of extracting information from the graph database is determining a file's history. Since every file is unique in the database, the specific node can be expanded to list all the actions that affect the file. This can be done with the following query:

```
1 MATCH (obj:object)
2 WHERE obj.filename = "<filename>"
3 MATCH (sec:Second)<-[:AT_TIME]-(act:action)--(obj)
4 OPTIONAL MATCH (obj)<-[:LINK_TARGET]-(lnkObj)--(lnkAct:action)
5 RETURN sec.time, collect(act.action), collect(lnkAct.action)
```

The first two lines find the desired object. The third line finds all the actions and times that affect the desired object. Line four gives the history of link files that are associated with the desired object.

Power Events.

Using a combination of objects from the Windows event logs, power events can be determined (shutdown, startup, sleep). If enabled, the Windows Customer Experience Improvement (CEI) service will start and stop just before power events. The Windows Event Log service does the same. The Event Log also records the Windows Version and uptime when Windows powers on. Each of these events appear at every power event and serves as the baseline for power events for a given time window. Additional events may appear in the shutdown/startup process: Microsoft Windows Power Troubleshooter - System Returned from a Low Power State; Network Interface Connected/Disconnected; Microsoft Windows RestartManager - Starting/Ending Session. These optional events help raise the likelihood of a power event occurring. Altogether, these events create a subgraph that shows power events:

```
1  MATCH (:parser {parserName: "eventLog"})<-[:PARSER]-(act:action)
       -[:EFFECTS]->(event:object)
2  WHERE event.filename IN ["EventLog/6006", "EventLog/6005"]
3  MATCH (act)-[:AT_TIME]->(sec:Second)
4  MATCH p = ()-[:NEXT*5]->(sec)-[:NEXT*5]->()
5  WITH p
6  UNWIND nodes(p) AS secNodes
7  MATCH (secNodes)<-[:AT_TIME]-(reqAct:action)--(reqObj:object)
8  WHERE reqObj.filename IN ["Microsoft-Windows-Winlogon/7001", "
       Microsoft-Windows-Winlogon/7002", "EventLog/6006", "EventLog
       /6009", "EventLog/6013", "EventLog/6005"]
9  OPTIONAL MATCH (secNodes)<-[:AT_TIME]-(optAct:action)--(optObj:
       object)
10 WHERE optObj.filename IN ["Microsoft-Windows-RestartManager/10000
       ", "Microsoft-Windows-RestartManager/10001", "e1iexpress/32",
       "e1iexpress/27", "Microsoft-Windows-Power-Troubleshooter/1"]
11 RETURN nodes(p), collect(reqAct), collect(optAct)
```

Lines 1-4 find a time window that contains one of the required events (event log service start or stop). The time window is then searched for related events in lines 5-11. If the required events are in the time window (reqAct), then it is recorded as a abstracted event. Optional matches bolster the likelihood of the extracted event and are also included in the abstracted event. An example of a subgraph that this query generates is in Figure 15.

Installed Programs.

Installed programs can also be determined by event logs. MSI Installer provides several event entires:

- 1040: Beginning Install Transaction

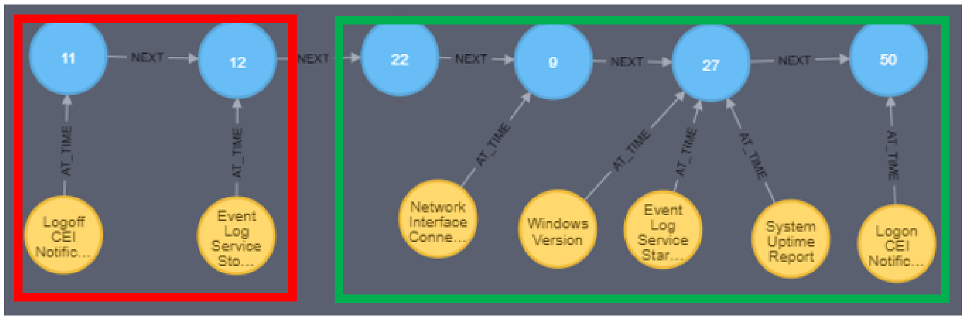

Figure 15. Shutdown Event (Red), Startup Event (Green).

- 11707: Installation Completed
- 1033: Install Info
- 1042: Intall Transaction Complete
- 1005: Installer Initiated Restart
- 1038: Windows Installer Requires a System Restart

These entries contain valuable information on the installation of a program. As the first event, the Beginning Install Transaction stores the location of the installation file. The next entries indicate if the installation was successful or if a restart is required. Combining these events together can be used to abstract an installation attempt on the forensics image. A sample query is below:

```
MATCH (:parser {parserName: "eventLog"})<-[:PARSER]-(act:action)
   -[:EFFECTS]->(event:object)
WHERE event.filename IN ["MsiInstaller/1040"]
MATCH (act)-[:AT_TIME]->(sec:Second)
MATCH p = (sec)-[:NEXT*10]->()
WITH p, act, event
UNWIND nodes(p) AS secNodes
MATCH (secNodes)<-[:AT_TIME]-(act2:action)--(obj2:object)
```

```
WHERE obj2.filename IN ["MsiInstaller/11707", "MsiInstaller/1033"
    , "MsiInstaller/1042", "MsiInstaller/1005", "MsiInstaller/1038
    "]
WITH act, event, collect({event: obj2.filename, act:act2}) AS
    actList
RETURN p, act, event, actList
```

Since this query also uses event log information, the query is similar to the power event abstraction in the previous section. An example of the install subgraph is in Figure 16. To extract more useful data, a file search can be conducted in the time window between the start and end of the installation to find the files installed, including the new executable file.

Office Suite Execution.

There are several artifacts that can indicate the execution of an program in the Office Suite. In this section, Word will serve as the example. The first identifying artifacts are prefetch files, covered in a previous section. There are also Word MRU registry entries. This set of registry keys can store several files that have been opened by Word, but it only stores time data for the last time a file was used. By expanding the history of all files except the most recent, more artifacts may be uncovered to

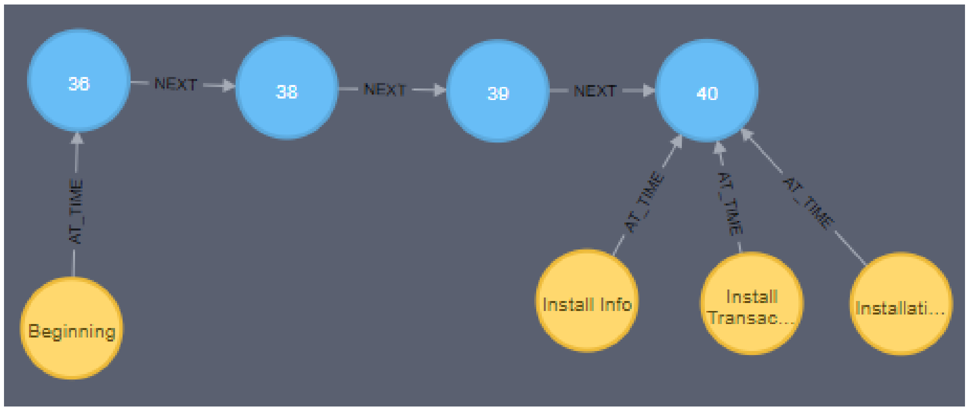

Figure 16. Installation Subgraph.

determine when a file was opened in Word.

Artifacts from the file history can include recent apps and recent item registry keys. Each user on the computer has a unique set of keys located in the ntuser.dat file. Executables have a specific application ID and are linked with this application ID using the keys following the pattern: /SOFTWARE/Microsoft/Windows/ CurrentVersion/Search/ RecentApps/⟨ APPLICATION ID ⟩/AppId. The value of this key is the path to the executable represented by the application ID. The keys for files have a naming convention as follows: /SOFTWARE/Microsoft/Windows/ CurrentVersion/ Search/RecentApps/⟨ APPLICATION ID ⟩/RecentItems/ ⟨ FILE ID ⟩/⟨ DATA TYPE ⟩. The data type of 'path' links a file ID with a filename. There are many other data types that include telemetry data from Microsoft like longitude and latitude. By linking the application IDs with the data type of path, it can be determined that a specific application used a certain file.

Another artifact is link files. Both Windows and the Office Suite make links to files and store them in the file system when used, helping to provide execution information. These files are located in the following locations:

- /Users/⟨ USERNAME ⟩/AppData/Roaming/Microsoft/Office/Recent/ ⟨ FILENAME ⟩.LNK
- /Users/⟨ USERNAME ⟩/AppData/Roaming/Microsoft/Windows/Recent/ ⟨ FILENAME ⟩.LNK
- /Users/⟨ USERNAME ⟩/Local/Microsoft/Windows/History/⟨ FILENAME ⟩.LNK

The final type of artifact is Word Reading Locations. These registry keys help remember where a user was located in a file so the program can start the file in the same place. The location of these keys is /SOFTWARE/Microsoft/Office/15.0/Word/Reading Locations/Document ⟨ DOCUMENT NUMBER ⟩. The keys are unique per user and are located in the ntuser.dat file. The keys also store file and time information for further evidence of program execution.

These artifacts can be gathered quickly by using the following query on the database:

```
MATCH (ft:fileType)<-[:TYPE]-(obj:object)
WHERE ft.name IN ["doc", "docx"]
OPTIONAL MATCH (obj)<-[:LINK_TARGET]-(obj2:object)
WITH DISTINCT obj, obj2
WITH [x IN collect(obj) + collect(obj2) | id(x)] AS docObjIds
MATCH (docObjs:object)--(docAct:action)-[:AT_TIME]->(docSec:
    Second)
WHERE id(docObjs) IN docObjIds
RETURN docSec.time AS time, docObjs.filename AS filename, docAct.
    action AS action
UNION
MATCH (obj:object)
WHERE toLower(obj.filename) CONTAINS "winword.exe"
OPTIONAL MATCH (obj)<-[:LINK_TARGET]-(obj2:object)
WITH DISTINCT obj, obj2
WITH [x IN collect(obj) + collect(obj2) | id(x)] AS exeObjIds
MATCH (exeObjs:object)--(exeAct:action)-[:AT_TIME]->(exeSec:
    Second)
WHERE id(exeObjs) IN exeObjIds
RETURN exeSec.time AS time, exeObjs.filename AS filename, exeAct.
    action AS action
UNION
MATCH (:parser {parserName: "mactime"})<-[:PARSER]-(act:action)
    --(hist:object)
WHERE hist.filename CONTAINS "AppData/Roaming/Microsoft/Office/
    Recent" OR hist.filename CONTAINS "AppData/Local/Microsoft/
    Windows/History" OR hist.filename CONTAINS "AppData/Roaming/
```

```
      Microsoft/Windows/Recent"
21  OPTIONAL MATCH (hist)-[:TYPE]->(ft:fileType)
22  WITH DISTINCT hist, act, ft
23  WHERE ft IS null OR ft.name <> "lnk"
24  MATCH (act)-[:AT_TIME]->(sec:Second)
25  RETURN sec.time AS time, hist.filename AS filename, act.action AS
       action
26  UNION
27  MATCH (:regType {type: "ntuser.dat"})<-[:REG_TYPE]-(key:object)
      --(keyAct:action)-[:AT_TIME]->(keySec:Second)
28  WHERE key.filename CONTAINS "Word/Reading Locations"
29  RETURN keySec.time AS time, key.filename AS filename, keyAct.
      action AS action
```

This query is a set of sub queries joined by the 'UNION' statements. The first sub-statement (Lines 1-8) find relevant files ('.doc' and '.docx' files) and any corresponding link files. Lines 10-17 return Word's executable file and associated link files. The next statement finds relevant Office or Windows files, and the final statement finds the Reading Location Registry Files.

Web History.

Combining web history entries can show complex activities. As mentioned previously, there are three main types of data recorded in the graph: history, downloaded files, and keyword values. By chaining consecutive web visits using the visit IDs and time, a sequence of visits can be obtained. Furthermore, downloaded files can be tracked to the destination on the host file system where file history can be obtained (Figure 17). This figure shows the web history entry (1) and the downloaded location of the file (2). The subgraph also shows that Word opened using a prefetch file (3).

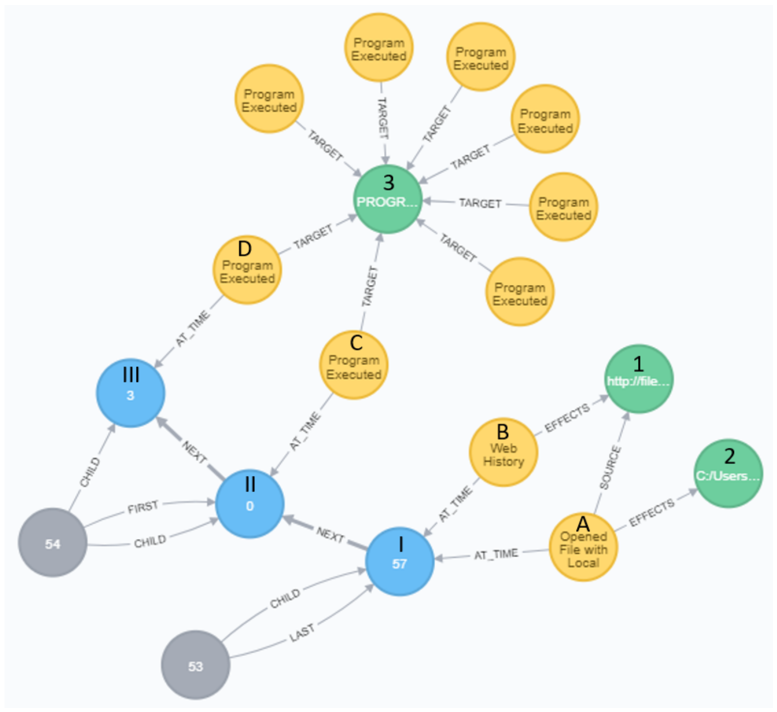

Figure 17. Download/File System Integration.

1. Match a pattern of level $n-1$ abstractions
2. Find the earliest and latest timestamps in matched abstractions
3. Create level n abstraction node
    - 3.1. Create relationships with all matched abstractions of level $n-1$
    - 3.2. Create relationships with matched values in step 2

Figure 18. Steps to Create an n Level Abstraction.

Combining Expert Rules.

The previous abstractions can also combine together to create new layers of abstraction. Figure 17 is a basic concept of this idea. Both web history and file table information are combined to enrich an examiner's understanding of events that occurred. A graph database is a great way to structure abstraction using a tree-like structure (Figure 19). The steps to create this structure is in Figure 18. Consider actions as a level 0 abstraction. Level 1 abstractions have edges that link them to the events loaded into the subgraph. For example, a shutdown event would have edges to the Customer Experience Improvement Service Shutdown and the Window Event Log Service Shutdown. To represent the time window for an abstraction, the actions would also be used. The two actions with the earliest and latest time would be used for the 'Start' and 'End' times for an abstraction. Using Figure 17 as an example again, a level 1 abstraction would have edges to all action nodes (A-D) and a start time of (I) and a end time of (III). Multiple level 1 abstractions could then be abstracted into a higher level event.

TEAR Ruleset.

The second method of abstraction is to use a machine-generated ruleset created by Temporal Event Abstraction and Reconstruction (TEAR). As mentioned in Chapter 2, TEAR is a machine-generated method of modeling high events using pattern matching. The steps to prepare the graph database for the application of the TEAR ruleset are as follows: build the ruleset tree, apply terms to the graph, create the time windows, and find the high-level events.

Build Ruleset Tree.

As shown in Figure 21, strings, production rules, and terms can be built in a tree-like structure. The very first step is to ensure that all production rules, terms, and strings are unique in the graph database by setting uniqueness constraints. These constraints allow for indexing and faster lookup. Adding rules to the tree occurs in two steps: first the production rules are built and second the strings and terms are added.

The production rules, like the terms and strings, are stored in a SQLite database

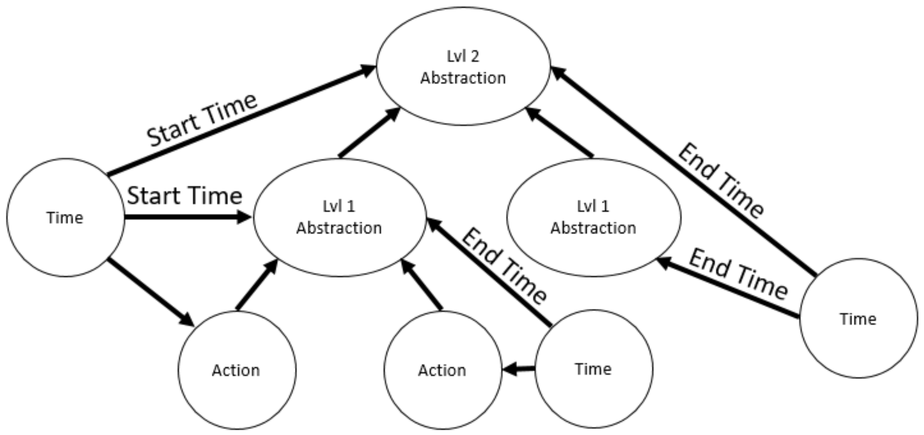

Figure 19. Example Abstraction Tree.

1. Build ruleset tree
   - 1.1. Import production rules
     - 1.1.1. Remove single variants
     - 1.1.2. Remove chain rules
     - 1.1.3. Create root rule
   - 1.2. Import strings and terms
     - 1.2.1. Add ‘parent’ relationships
     - 1.2.2. Remove orphan nodes
     - 1.2.3. Label levels of the rule tree
     - 1.2.4. Create term lists
2. Apply terms
3. Create time windows
4. Find high-level events

Figure 20. Steps to Apply TEAR Ruleset.

by TEAR. This ruleset must be converted into a graph format. The conversion starts by understanding the construction of production rules. These rules can be composed of other production rules, terms, and strings. Additionally, if a production rule is a child of another production rule, there are two relationship options. Component is

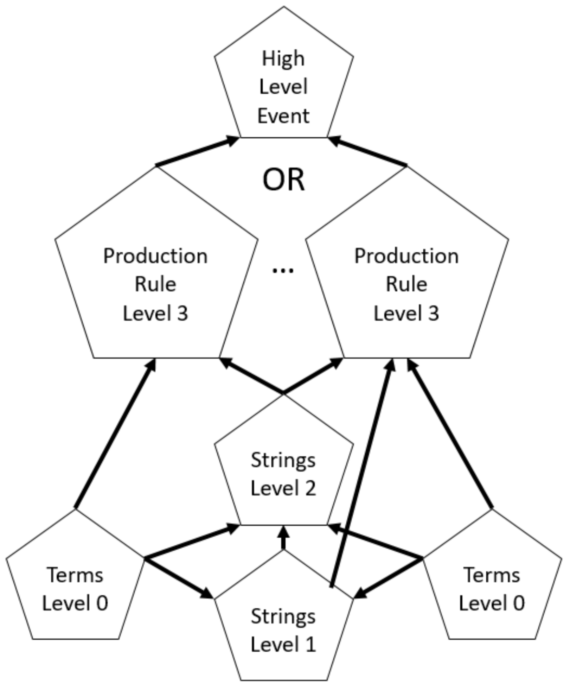

Figure 21. Temporal Event Abstraction & Reconstruction Hierarchy.

the first type of relationship. Component relationships are an ‘AND’ relationship; all children must exist in order for the parent to exist. Further, all children must occur in a specific temporal order. Each child has a number that determines the sequence of events. Variant is the second type of relationship. This relationship creates an ‘OR’ relationship with its children. If one child matches, then the parent exists. Conversion into the graph database makes these relationships explicit (Figure 22).

Production rules and a child are retrieved from each row of the SQLite query. These items are added a list of entries in Python for mass upload to the graph database. These entries are then sent to the neo4j database using the commands ‘UNWIND’ and ‘MERGE’. The unwind command breaks down a list to each individual element, allowing entries to be processed one at a time. The merge commands only add an object if it does not already exist in the database. This command, along with the uniqueness constraint, allows separate references to the same object.

Once the production rules are added to the rule tree, there are several ways to clean it up before adding terms and strings. First, single variants can be removed. Some production rules have one child and a variant relationship with another production rule. These rules can be combined without affecting the end result of the rule tree; if the child exists, the parent will exist. Another cleanup procedure is to remove 'Chain' production rules. These are production rules with one child and a component relationship. Like the previous rule, if the child exists, the parent will exist as well. The last procedure is to create a root rule and establish an relationship labeled 'ROOT' between the root rule any production rules without a parent. Any production rule with this relationship to the root rule is considered a top-rule. If a top-rule exists, a high-level event has occurred. Once the cleanup procedures are complete, the rule tree's terms and strings can be added

Terms and strings are added to the root tree in a similar manner to the production rules. Also like production rules, there are several cleanup procedures that occur to the rule tree after the terms and strings are added. The first procedure is to add a 'PARENT' relationship to each set of nodes that has a parent/child relationship, as in Figure 22. Prior to this procedure, nodes only have a component or variant relationship. Adding this relationship makes rule tree traversal easier in neo4j since all parent/child relationships have a common label. Another procedure eliminates 'orphan' nodes. These nodes do not have a path to the root node and cannot affect the execution of a high-level event. The next procedure breaks the rule tree into levels. Level 0 is all terms, level 1 is all strings composed of only terms, and levels 2 to n are determined by the distance a string or production rule is from a level 1 string. Figure 21 provides a small example of level determination. Establish levels helps future procedures in ensuring that all lower-level items are processed before higher items in the tree. The next procedures create all possible event combinations for high-level events.

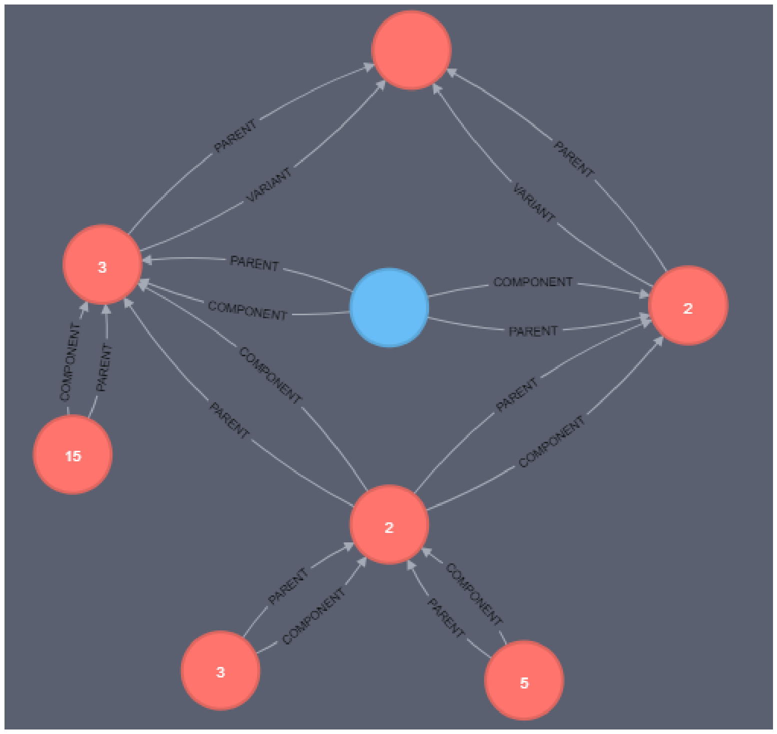

Figure 22. Portion of Production Rule Tree, Production Rules (Red), Term (Blue).

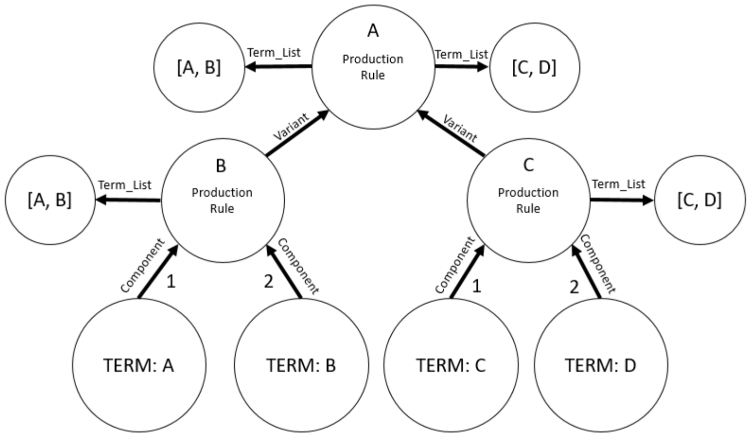

Figure 23. Example of Term List Construction.

The order of terms in a production rule is important for determining if a high-level event occurred. The order of terms refers to the arrangement of terms at the bottom of the rule tree. Since all terms have component relationships, there is an order of terms for all subsequent strings and production rules. Special node(s) called term lists are attached to each string and production rule (See Figure 23) to reflect this order. Production rules B and C have a single term list, reflecting the order of terms below it. However, production rule A has variant relationship with the production rules below, changing how term lists are created. In this relationship, the term list(s) of the child nodes are copied and added to production rule A, since any term sequence is valid. If production rule A encounters a component relationship above it, the number of term lists for the parent node is determined by the multiplying the number of term lists of each component. For example, say there is another production rule, D, that has three term list nodes. If production rule A and production rule D were components of the production rule E, there would be six term lists attached to E. This is because any term list attached to A can be combined with any term list in D. Term lists are created by level, ensuring that all child nodes have at least one term list node when the parents are processed. When this process is complete, all possible term lists for top-rules are determined.

The last procedure before terms can be applied is to add a property to all term list nodes that represents top-rules. This property represents the unique terms in each term list. This is used when finding high-level events.

Apply Terms.

Now that the rule tree is complete, terms can be applied to the event graph. As discussed in Chapter 2, all terms have a regular expression pattern, an object type, and an action. If an object matches both the object type and the regular expression

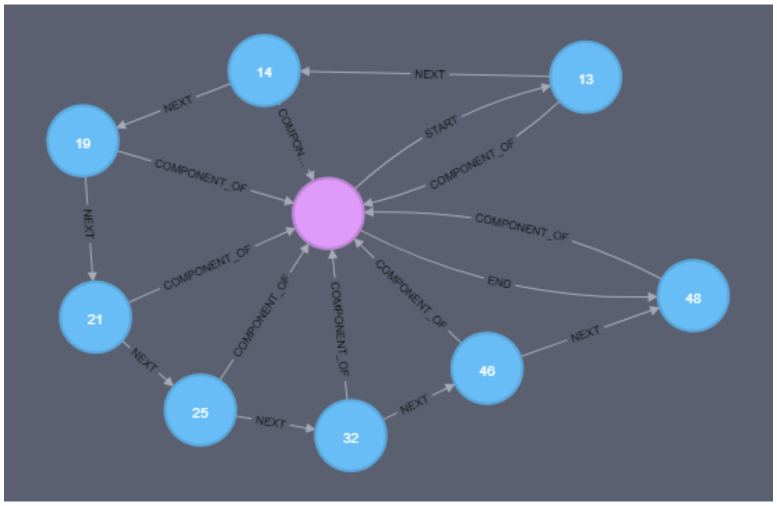

Figure 24. Example of Time Window, Seconds (Blue), Time Window (Purple).

of the term, and the term action matches the action that corresponds to the object, there is a term match. A term match creates an edge between the matched action and the term. Objects might match more than one regular expression, so the tiebreaker is the longest match. Terms are applied by type. The graph finds all terms and object/action pairs of the same type and finds matches until all object/action pairs are exhausted.

Create Time Windows.

The next step is to create time windows. The default time windows are 90 seconds for TEAR rulesets and are created so no time window is a subset of another. A time window has a start and end relationship with the first and last second nodes respectively in the time window (Figure 24). All second nodes in the time window have a relationship titled 'COMPONENT_OF' with each applicable time window. An example of a time window is presented in Figure 24.

After time window creation is complete, additional procedures are accomplished to shorten high-level event processing. The first procedure is to find the unique terms in each second of the graph. This is done through the neo4j query below:

```
1 MATCH (sec:Second)
2 SET sec.TEAR_terms = []
3 WITH sec
4 MATCH (sec)<-[:AT_TIME]-(act:action)
5 MATCH (act)-[:MATCHED_TERM]->(term:TEAR_Term)
6 WITH DISTINCT sec, term.ID AS termID
7 WITH sec, collect(termID) AS termList
8 SET sec.TEAR_terms = termList
```

Lines 1-3 find all the second nodes and sets the TEAR_terms property to an empty list. Lines 4-6 find all distinct seconds and terms. Lines 7-8 create a list of the distinct terms and set the TEAR_terms property. The next procedure finds unique terms for each time window using a similar query.

Find High-Level Event.

The final step in this process is to match terms contained in time windows to the top-rule variants. The top-rule variants are acquired from neo4j and are stored in Python dictionary (rule dictionary) for comparison. The next step is to import all time windows into another dictionary (time window dictionary) for comparison. Each time window has an entry in the dictionary and contains terms grouped by Unix timestamps.

Once the time windows and top-rule variants have been stored in Python, the next two procedures filter time windows based on the probability that they will contain a specific top-rule. Shrinking the amount of time windows to test for a particular top-rule is important for fast execution. This is accomplished by reviewing all variants for a particular top-rule to find terms that appear in 100% of variants. These terms are then compared to the unique terms in a time window; if a window contains all these terms, the time window is retained for further analysis. The retained time windows then compare its unique terms to the unique terms in each top-rule variant. If a time window contains a specific threshold (default is 90%) of the terms in a variant, the variant is stored in the time window dictionary for further processing. A time window can have many variants that meet the threshold. These procedures can significantly cut the number of time windows and variants that need further processing. The next steps process the remaining time windows and variants to find high-level events.

The first method of finding high-level events is to ignore term order. All remaining time windows and variants compile the number of each term. The number of each term is compared, and if the time window is only missing a specific threshold (default is four) of terms from a specific variant, the time window matches the variant.

The second method of finding high-level events is to find substrings in time windows. Each matching variant has a term list indicating the order of terms, and the time window has an ordered dictionary of timestamps (key) and terms (values). The terms contained in the first timestamp in the time window are compared to the first term of the variant term list. If there is a match, the term from both the variant and the time window is deleted. This process repeats until there are no more terms or there is no match. In either case, the next timestamp in the time window is reviewed until the whole time window has been examined. If the variant term list is empty or contains a certain percentage of the original term list, the time window is determined to match the high-level event.

Using the substring method, steps must be taken to ensure duplicates are eliminated. If two time windows contain the same sub-window with all the terms in a variant, the same event appears as a duplicate event. Therefore, the program records the beginning and end timestamps of a matched variant. If two events share the same timestamps, they are duplicates and only one match is kept.

3.6 Conclusion

In conclusion, PGER accomplishes its goal of abstracting user action events from digital artifacts in four steps. The first step extracts data from a device image. The second and third layer convert the extracted data into a usable ontological graph using neo4j. The last step allows examiners to extract user actions by expert rules or through machine-generated rulesets.

Performance is improved over traditional ontological databases by using a native graph processing and storage format. This allows users to leverage the advantages of a graph database without the cost of just-in-time assembly of a graph or the completion of many join statements. Utilizing a graph database also allows for more natural queries of ontological data, and enables users to find subgraphs by searching for path patterns.

IV. Results

PGER utilizes a graph database to help improve performance of queries common in ontological datasets. By allowing fast path traversals to explore relationships, a graph database avoids the costly join statements of a traditional database. PGER is also able abstract low-level events into higher-level events that are easier for examiners to understand. It does so through two methods, expert rules and machine-generated data sets. This chapter evaluates the effectiveness of PGER in five different categories.

The results of this project are divided into six sections. The first section elaborates on the testing environment for all tests in the chapter. Sections 2 and 3 examine different methods inside the graph conversion and data extraction processing layers. Section 4 highlights some speed advantages that neo4j has over SQLite when conducting certain queries. Section 5 examines PGER as a platform to apply expert rules to forensics images. Examples are provided to show potential applications. The final section provides a performance comparison between TEAR and PGER when applying machine-generated rulesets.

4.1 Testing Environment

All tests were conducted on a machine with the following specifications:

- CPU: i3-6100U (2 Cores, 4 Threads)
- RAM: 12 GB
- HDD: 250 GB Samsung 840 EVO SSD
- OS: Ubuntu 16.04 LTS
- Docker Version: 17.09

- Python Version: 3.6
- Elastic Stack Version: 5.5.2
- neo4j Version: 3.2.2
- PLASO Version: 1.5.1

The test image was a 65 GB Window 10 image with sample activities that included:

- Web Browsing/Downloading Files from Microsoft Edge, Mozilla Firefox, and Google Chrome
- Microsoft Office: Creation/Manipulation of Word, Excel and PowerPoint Files
- Viewing Downloaded PDFs
- Sleep, Startup, and Shutdown Sequences
- View image files
- Manipulate files in Windows Explorer

There were two additional Windows 7 images in .dd format (10 GB and 20 GB) that were used to test data extraction performance.

4.2 Data Extraction Performance

The PLASO and TEAR event abstraction methods contained in the data extraction processing layer are compared using consistency, the number of events captured, and processing time. Consistency is the most important attribute; an examiner needs to know the data extractor can find events if they are on the forensics image. The number of events captured reflects how closely extracted events represent the actions that occurred on a device image. This provides a more complete picture to the examiner, making event reconstruction more accurate. Finally, processing time is measured. This is not the most important metric, but it can direct an examiners choice if time is a factor. It is important to note that neither method involves the neo4j database; PLASO outputs results to an elasticDB and the TEAR event extractor outputs several text and SQLite files. However, these tests are important as the output of the event extractor affects the efficacy of the graph conversion and abstraction processing layers.

All tests were conducted on three Windows images in .dd format, sizes of 10 GB, 20 GB, and 65 GB. Both PLASO and TEAR event extraction were run as the only docker containers on the test machine. Timing results were an average of three runs.

In regards to the number of parsers used, PLASO had the clear advantage by using 57 on the win7_slow preset. It records events types such as scheduled events, recycle bin activity, and metadata inside files. In addition, PLASO parsed registry keys common to forensics investigators such as MRU lists, userassist, and appcompatcache. In contrast, the TEAR event extractor recorded 14 different types of events. Registry entries from each hive were stored in a single file so useful keys could be parsed in the next processing layer.

In terms of consistency, the TEAR event extractor excelled. In every image, the program was able to find entries on all 14 parsers. PLASO, however, did not find any registry entries on two of the three images. This is a critical failure because registry entries provide essential information for examiners. The cause for this is currently unknown, but Windows 10 might contribute to this problem since one of failed images used this Windows version.

The processing times for each method are listed in Table 2. PLASO took a

Table 2. Data Extraction Processing Time.

Image Size (GB)	PLASO (h:mm:ss)	TEAR Event Extractor (h:mm:ss)
10	0:36:12	0:04:23
20	0:45:44	0:05:03
65	2:15:16	0:08:23

significantly longer time to generate event, but it also captured many more events compared to the TEAR event extractor. The advantage in this evaluation depends on how important processing time is to user. If the need is urgent, TEAR provides events quickly, even if it does not provide as much context as PLASO.

Due to its consistency, the TEAR event extractor was used in the remaining tests. The PLASO version that was tested missed critical events, making it unusable for the other evaluations in this chapter. If future versions become more consistent, PLASO is worth the time penalty for the extra fidelity on the device image unless time is a critical factor.

4.3 Graph Conversion Performance

PGER also provided two methods to import parsed artifacts into the graph database: logstash and a python script. To compare these methods, flexibility, processing time, and accuracy were important factors. A flexible method of graph conversion can handle multiple formats of data extraction and easily add the ability to convert a new type of event into a subgraph. Processing time is important because PGER must be able to compete with other tools such as TEAR; any performance benefit of PGER is minimized if converting a dataset is time-consuming. Finally, accuracy is important as well because both methods should produce the same graph if they used the same extracted events and parsed the same events.

All tests involved the 65 GB image described in the first section. Timing results were an average of three runs. Both methods converted the following events from the TEAR event extractor dataset: firefox history, chrome history, all registry entries, NTFS file table entries, prefetch files, and Windows event log entries.

Logstash provided a more flexible interface for graph conversion. It contains plugins to ingest 48 different formats, and 3d party plugins are available to process even more formats. In PGER, logstash ingested three different input formats: elasticDB, csv, and SQLite. If event extractors developed different output formats, logstash would likely be able to ingest the new format with little additional programming. In contrast, the python script only ingests the TEAR event extractor.

The processing times for each method are in Table 3. There are two main reasons why the python script was substantially faster than logstash. First, logstash needed an operational elastic stack (elasticDB and kibana) to operate, requiring more resources than a python script; CPU utilization on the test machine was consistently above 90%. The second, more important factor was the lack of bulk operations with the neo4j database. Each event is sent to neo4j as a separate REST query, quickly overwhelming connection limits. A potential work-around would be to take the results from logstash and place them in an intermediate elasticDB index. Logstash interfaces directly with elasticDB, so it caches processed entries for bulk imports. The entries in this intermediate database could be processed by a python script to bulk import the data to neo4j.

Both methods provided accurate results; each method produced identical graph characteristics (Table 4).

If processing speed can be improved, logstash provideed a more flexible method of

Table 3. Graph Conversion Processing Time.

Logstash (h:mm:ss)	Python Script (h:mm:ss)
1:13:44	0:15:26

Table 4. neo4j Database Parameters After Graph Conversion.

neo4j Database Size	656.18 MB
Node Count	1,069,671
Relationship Count	2,587,503

graph conversion, but the immense speed difference and resource requirements makes the python script the ideal method at this time.

4.4 neo4j Performance

To analyze the performance of neo4j compared to the SQLite database where the TEAR dataset is stored, three types of queries were evaluated by time. Timing results were an average of three runs.

The first query type was a join query to find all files in the MRU registry keys. The SQLite database has a separate table for MRU entries that includes a timestamp. The file represented by the MRU entry is located in the items table, containing all objects in the forensics image. A join is required to connect both tables. The syntax used to find the MRU files for both databases is below:

```
#SQLite Query
SELECT mruList.TimeStamp, items.sort
FROM mrulist
JOIN items ON mruList.TraceID = items.TraceID

#neo4j Query
MATCH (:regType {type: "MRU"})<-[:REG_TYPE]-(key:object)--(act:
    action)--(mruObj:object)
MATCH (act)-[:AT_TIME]->(sec:Second)
RETURN sec.time, mruObj.filename
```

The second query type filtered a single table to find all files with the extension of 'doc'. This query filters the rows of the items table in the SQLite database by the extension column. The syntax for the second query is below:

```
#SQLite Query
SELECT items.sort
FROM items
WHERE items.ext IS "doc"

#neo4j Query
MATCH (:fileType {name: "doc"})<-[:TYPE]-(obj:object)
RETURN obj.filename
```

The last query retrieved all rows from the item name (called sort) column in the database. The SQLite query returns the sort column in the items database. The syntax for this query is below:

```
#SQLite Query
SELECT items.sort
FROM items

#neo4j Query
MATCH (obj:object)
RETURN obj.filename
```

The performance of all query types are in Table 5. In queries that require a join statement from the SQLite database, the advantages of a graph database become

Table 5. SQLite and neo4j Query Comparison.

Query Type	Database Type	Time (ms)
Join	neo4j	45
	SQLite	5548
Filter	neo4j	31
	SQLite	128
Whole Column	neo4j	987
	SQLite	162

apparent. neo4j was able to leverage the relationships in the database, only requiring one index lookup to find the correct regType node. Since there are only eight nodes of this type in the database, this lookup is extremely fast. It then utilized the relationships in the database to traverse the registry key and action nodes to find all files that belonged to the MRU registry keys. For the filter query, neo4j was still faster, but the gap is narrowed. neo4j still took advantage of conducting one index lookup for the fileType, of which there are only 582 nodes. However, not every query was faster in neo4j; the whole column query type serves as an example. SQLite was much faster at requesting data from one table where neo4j could not leverage relationships to its advantage.

4.5 Expert Rule Results

PGER can use expert rules to abstract forensics artifacts into more understandable events. This is accomplished by matching a specific pattern of low-level events to create a higher-level event. An example of this is combining several Windows event logs to determine a device shutdown. PGER's ability to abstract events is evaluated by the power events and file downloader rules.

Power Events.

A power event represents a shutdown or startup on the forensics machine. To find the power events on the device image, the rules in Chapter 3 were used. As an overview, a power event is primarily determined by the status of the CEI and Windows Log Services. If they are shutdown, it indicates a shutdown event; the opposite is also true. Startup also can contain optional Windows log entries to further bolster the evidence of a startup event.

After applying rules to the dataset, a total of 65 shutdown events and 67 startup

Table 6. Power Event Sequence.

Start Time	End Time	Event
1491971379	1491971414	Startup
1491972643	1491972656	Shutdown
1491972859	1491972888	Startup
1492112687	1492112712	Startup
1492113771	1492114425	Shutdown

events were detected. There was apparently an error in the rules as two shutdown events were missing. After examination, there were two sequences that contained consecutive startup events. One such sequence is in Table 6.

After reviewing the actions between the two consecutive startup events using the query below, a Windows Update seemed to occur.

```
MATCH (sec:Second {time: 1491972888})
MATCH p = (sec)-[:NEXT*..500]->(:Second {time: 1492112687})
UNWIND nodes(p) as secNode
MATCH (secNode)<-[:AT_TIME]-(act:action)-[:EFFECTS]->(obj:object)
RETURN secNode.time, collect(act.action), collect(obj.filename)
```

Of the 5706 objects that had altered timestamps, 4830 matched the pattern C:/ Windows/WinSxS/*; these files are known to be related to Windows updates. The shutdown sequence during Windows updates differs from other shutdowns and do not include additions to the event logs.

Updates might also explain the large gap in the start and end times for some shutdown events. For example, the last event in the table has a difference of 654 seconds between its start and end times. Upon examination, several files are changed 361 seconds after the start of the event. These files match the pattern C:/Windows/WinSxS/ amd64_windows-defender-am-sigs or C:/ProgramData/Microsoft/Windows Defender /Definition Updates/. As a result, it appears that Windows Defender Definitions are updated before a shutdown.

Downloaded Files.

The downloaded files expert rule is a great example of utilizing many different low-level event types to create a complex abstraction. The expert rules combine the following:

- Previous web history
- URL source for the downloaded file
- Location of the download file on the forensics image
- File history of the downloaded file
- Username responsible for modified registry keys

Firefox history shows nine files were download from the browser in the image and that the rules found all nine entries. One example entry is contained in Table **??**.

These tables highlight the insights gained through connected data. Through this abstraction, an examiner can see recent web history, the URL source, the file system destination for the downloaded file, and what actions took place on the downloaded file. PGER was able to perform this relationship-heavy query for all nine objects in 18 ms.

4.6 TEAR Ruleset Application Results

Applying expert rules, as evidenced by the results above, can be an effective way to abstract data. However, both of these rules in the previous section are short, only matching a few different events in a time window. Some user actions are incredibly hard to capture using expert knowledge alone due to the immense number of objects and events that affect the outcome. TEAR tackles this problem by using a machine

Table 7. **Downloaded File Expert Rule Result.**

Download Information	
Username	user
Download Time	1497574737
Filename	C:/Users/user/Downloads/Snowball_Fighting(2).doc
Shortened URL	http://files.geekdo.com/geekfile_download.php?

Web History	
Time	URL
1497574730	/filepage/28906/snowball-fighting-rules-word-doc
1497574732	/file/download/2hkk77tped/Snowball_Fighting.doc
All URLs start with https://www.boardgamegeek.com	

File History	
Time	Action
1497574765	Accessed, Created Time Altered
1497574767	Modified Time Altered
1497574768	Changed Time Altered
1499350036	Recent .doc Changed
1499350036	Recent Docs Changed

to generate patterns for complex events. For example, TEAR has found 86 different combinations of events that can occur when Mirosoft Word opens. Incorporating the TEAR method of finding high-level events is important for PGER to find complex events.

The final evaluation tested PGER's ability to replicate the results produced by TEAR on the same dataset. Both processing time and accuracy in replicating TEAR's results were the criteria for evaluating PGERs performance.

Testing for both PGER and TEAR used the same device image, event extractor, and pre-processed ruleset. The ruleset was is limited to Opening Word as the lone top-level rule. Terms that represented accessed prefetch files appeared in all term sequences. Since both tools used the same data extraction method, this step was ignored in both testing categories. Timing results were an average of three runs.

Processing Time.

Performance is based on the time required to determine high-level events from the output of the TEAR data extractor. TEAR accomplishes this in the following steps:

1. Load event data from a SQLite database to memory
2. Apply terms to events, abstract terms to strings
3. Process the generated term list in chronological order. Find high-level events by matching a variant with the term list of the current time window.

To accomplish the same feat, PGER uses the graph conversion, normalization, and abstraction processing layers. Graph conversion and normalization matches with TEAR's first step. PGER's abstaction layer accomplishes both steps 2 and 3. To provide a direct comparison, the PGER abstaction layer splits into two categories to better match TEAR's steps. The first group, corresponding to TEAR's second processing step, contains building the ruleset tree, term application, and time window creation. Finding high-level events corresponds with TEAR's third processing step. In both programs, steps 1 and 2 are only accomplished once per image as long as the same ruleset is applied. Table 8 contains the runtimes for PGER and Table 9 compares TEAR and PGER runtimes.

Table 8. PGER Processing Times.

Processing Layer	Time (mm:ss)
Graph Conversion	15:26
Normalization	1:02
Build Ruleset Tree/Apply Terms	2:44
Create Time Windows	1:48
Find High-Level Event	1:12
Total	22:12

Table 9. PGER and TEAR Runtime Comparison (h:mm:ss).

TEAR Step	TEAR Time	PGER Step	PGER Time
1	0:29:52	Graph Conversion	0:16:28
		Normalization	
2	3:02:06	Build Ruleset Tree/Apply Terms	0:04:32
		Create Time Windows	
3	0:09:34	Find High-Level Events	0:01:12

PGER applies the machine-generated ruleset to a forensics image in less time than TEAR. This advantage is shown the best in step 2 where the TEAR ruleset is constructed into a tree and used to find the objects that apply to each term. One significant factor in the processing time difference is that PGER does not abstract sets of terms into strings; PGER finds all high-level rules only as different compositions of terms. This results in longer comparisons between time windows and rulesets, but PGER filters time windows in step 3 that do not include terms that are in every variant. In this case, the difference is significant, eliminating nearly 80% of all time windows (From 10,178 to 2,444). If there were rules that did not filter out as many time windows, the PGER runtime to find high-level events could increase by a factor of five. However, PGER would still provide a performance advantage over TEAR.

Accuracy.

Running the TEAR program on this image resulted in the identification of 12 high-level events. In the case of this ruleset, all high-level events represented the user activity 'Word Opened' corresponding with start and end times established by the tool. There were two methods developed for PGER to find high-level events: unordered and substring. When applying the unordered methodology, PGER found 87 possible activities. This high amount of false positives was due to the shorter variants in the ruleset. The shortest rule required prefetch file exeuction and two modifications to registry files whose key matches the regular expression '^ .*/Windows/*'. If taken out of order, different combinations of a variation matched in many places causing false positives.

Substring matching cares about the order listed in the variant, but does not stop if there are extra terms that are not included in the variant. This significantly reduces matching time windows down to 26. The extra findings are not false positives, but are duplicates. If two time windows contain the same sub-window with all the terms in a variant, the same sub-window appears twice. Eliminating duplicates brings the total matches down to 12 with slightly different time windows compared to their TEAR counterparts due to different matching methods. Eliminating duplicates gives PGER duplicate results to the TEAR program.

4.7 Conclusion

The results of the PGER evaluation show that a graph database can provide performance gains over other storage methods using queries common in ontological datasets. PGER can also match the accuracy of TEAR and provide an increase in processing speed by filtering time windows and not abstracting terms into strings. In regards to specific methods inside of PGER, PLASO did not provide consistent results making the TEAR event extractor the default method for event extraction. For graph conversion, logstash lagged behind the python script due to a lack of bulk imports to the neo4j database. Finally, expert rules were applied to the database, providing useful information for the examiner. However, longer rules might be better created by machine pattern matching, like TEAR, due to the difficulty in codifying the execution of a large set of events and objects.

V. Conclusion

The demand for digital forensics has risen significantly. This need is, in part, driven by the time-consuming task of manual data correlation required for digital forensics investigations. Forensics research tries to provide data correlation tools for examiners that reduce the time needed to process each case. Researchers face three main challenges: data volume, data heterogeneity, and legal admissibility. The NIST model helps provide guidance on how to produce legally admissible results. Research has shown that ontological storage of event data provides rich data for examiners by creating relationships between events, but data is stored in a format that does not quickly query relationships. PGER utilizes a native graph database that takes advantage of the ontological relationships for faster performance. PGER also allows for two types of abstraction to help examiners handle the immense data volume of modern forensics images.

Previous research focused on four different categories: data extraction, data representation, event reconstruction, and graph databases. Data extraction involves finding events within a forensics image. Tools can focus on one type of data, but the most useful tools extract multiple types of data, allowing for a more complete picture of an image. Data representation organizes data in a common format to provide consistency for event reconstruction. PGER utilizes the SOSLA ontological format as a base subgraph for all events in the database and adds elements of ORD2I to expand subgraphs as needed. PGER also relies on inference rules and the TEAR method as event reconstruction methods. Finally, graph databases have two main structures: triples and labeled property graphs. Both structures have different advantages, but index-free traversal is a key benefit of labeled property graphs and important for PGER's performance.

PGER is designed into four processing layers: data extraction, graph conversion, normalization, and abstraction. The first step extracts data from a device image. The second and third layer convert the extracted data into a usable ontological graph using neo4j. The last step allows users to extract data by expert rules or through machine-generated rulesets. Each layer is separated from the others using docker containers. This virtualization helps minimize the dependencies on the machine running PGER and allows multiple instances of neo4j to run concurrently.

The performance of PGER is improved over traditional ontological databases by using a native graph processing and storage format. This allows users to leverage the advantages of a graph database without the cost of just-in-time assembly of a graph or the completion of many join statements. Utilizing a graph database also allows for more natural queries of ontological data, affording users the ability to find subgraphs by searching for path patterns. PGER can also match the accuracy of TEAR and provide an increase in processing speed by filtering time windows and not abstracting terms into strings. In regards to specific methods inside PGER, PLASO did not provide consistent results, making the TEAR event extractor the default method for event extraction. For graph conversion, logstash lagged behind the python script due to a lack of bulk imports to the neo4j database. Finally, expert rules were applied to the database, providing useful information for the examiner. However, longer rules might be better created by machine pattern matching, like TEAR, due to the difficulty in codifying the execution of a large set of events and objects.

5.1 Future Work

PGER has highlighted many areas areas in need of improvement. First, PLASO's effectiveness should be tested. During development, PLASO did not provide consistent results for data extraction. However, a new version has since been posted and may provide more consistent results. PLASO provides many different data types for evaluation, providing a richer dataset for examiners. The ability to use this tool could ease in the generation of expert rules by incorporating more events.

A second area of improvement involves PGER's graph conversion processing layer. The python script providing bulk imports is markedly faster than using logstash and the elastic stack. However, it can only process events from the TEAR event extraction process. If expanded to incorporate PLASO inputs, it could provide a performance benefit for PLASO imports when they become more reliable.

Another possible area for improvement is human curation of the TEAR rulesets. With machine generation, there may be terms or strings that are not significant in the production rules. This expands the rule trees and increases processing time. Human experts may be able to identify these extra items and eliminate them, thus improving performance.

User accessibility is also an extensive area of possible improvement. PGER in its current form is not user friendly for non-technical examiners. A user interface that used the neo4j database as the back-end would give non-technical users the ability to execute queries on the database. In addition, a script could be used to link and execute all processing layers together; currently each process layer has to be executed manually.

A final improvement would be the design and addition of a standardized interface for expert rules. PGER currently relies on queries and python scripts programmed by the user to create rules. This results in a non-standardized approach and requires knowledge of the database and programming. The ideal situation would enable a user to simply identify a set of objects or actions within a certain time frame that indicates a high-level event. The standardized interface would then interact with the database and provide the abstraction, requiring no special programming skills.

Bibliography

1. K. Alghafli, A. Jones, and T. Martin, "Forensic Analysis of the Windows 7 Registry," *Journal of Digital Forensics, Security and Law* (December), p. 17, 2010.

2. R. Angles, "A comparison of current graph database models," *Proceedings - 2012 IEEE 28th International Conference on Data Engineering Workshops, ICDEW 2012* , pp. 171–177, 2012.

3. R. Angles and C. Gutierrez, "Querying RDF Data from a Graph Database Perspective," *European Semantic Web Conference* , pp. 346–360, 2005.

4. M. R. Aniba, S. Siguenza, A. Friedrich, F. Plewniak, O. Poch, A. Marchler-Bauer, and J. D. Thompson, "Knowledge-based expert systems and a proof-of-concept case study for multiple sequence alignment construction and analysis," *Briefings in Bioinformatics* **10**(1), pp. 11–23, 2009.

5. F. Buchholz and C. Falk, "Design and Implementation of Zeitline: A Forensic Timeline Editor," *Digital Forensics Research Workshop* , pp. 1–7, 2005.

6. Bureau of Labor Statistics, "Occupational Outlook Handbook: Forensic Science Technicians," 2017.

7. Bureau of Labor Statistics, "Occupational Outlook Handbook: Information Security Analysts," 2017.

8. J. Carroll, I. Herman, and P. F. Patel-Schneider, "OWL 2 Web Ontology Language RDF-Based Semantics (Second Edition)," 2012.

9. E. Casey, G. Back, and S. Barnum, "Leveraging CybOX to standardize representation and exchange of digital forensic information," *Digital Investigation* **12**(S1), pp. S102–S110, 2015.

10. Y. Chabot, A. Bertaux, C. Nicolle, and M.-T. Kechadi, "A complete formalized knowledge representation model for advanced digital forensics timeline analysis," *Digital Investigation* **11**, pp. S95–S105, 2014.

11. Y. Chabot, A. Bertaux, C. Nicolle, and T. Kechadi, "Automatic timeline construction and analysis for computer forensics purposes," *Proceedings - 2014 IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014* , pp. 276–279, 2014.

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

13. K. Chen, A. Clark, O. De Vel, G. Mohay, and Q. Brisbane, "ECF Event Correlation for Forensics," *1st Australian Computer, Network & Information Forenics Conference 2003* (November), pp. 1–10, 2003.

14. Cyber Threat Intelligence Technical Committee, "CTI Documentation," 2017.

15. D. J. Daniels and S. V. Hart, "Forensic Examination of Digital Evidence : A Guide for Law Enforcement," *U.S. Department of Justice Office of Justice Programs National Institute of Justice Special* **44**(2), pp. 634–111, 2004.

16. X. Dong, E. Gabrilovich, G. Heitz, W. Horn, N. Lao, K. Murphy, T. Strohmann, S. Sun, and W. Zhang, "Knowledge vault: a web-scale approach to probabilistic knowledge fusion," *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '14* , pp. 601–610, 2014.

17. P. Gladyshev and A. Patel, "Finite state machine approach to digital event reconstruction," *Digital Investigation* **1**(2), pp. 130–149, 2004.

18. A. Goel, Wu-chang Feng, D. Maier, Wu-chi Feng, and J. Walpole, "Forensix: A Robust, High-Performance Reconstruction System," *25th IEEE International Conference on Distributed Computing Systems Workshops* (October), pp. 155–162, 2005.

19. R. A. Good, *AutoProv : An Automated File Provenance Collection Tool.* Masters thesis, AFIT, 2017.

20. GraphAware, "GraphAware Neo4j TimeTree," 2018.

21. K. Gujónsson, "Mastering the Super Timeline With log2timeline," 2010.

22. P. J. Hayes and P. F. Patel-Schneider, "RDF 1.1 Semantics," 2014.

23. E. H. Herskovits and G. F. Cooper, "Kutato: An Entropy-Driven System for Construction of Probabilistic Expert Systems from Databases," *Clinical Orthopaedics and Related Research* , p. In print, 2013.

24. J. James, P. Gladyshev, M. Abdullah, and Y. Zhu, "Analysis of evidence using formal event reconstruction," *Digital Forensics and Cyber Crime* **31**, pp. 85–98, 2010.

25. K. Kent, S. Chevalier, T. Grance, and H. Dang, "Guide to integrating forensic techniques into incident response," tech. rep., 2006.

26. M. N. Khan, E. Mnakhansussexacuk, and I. Wakeman, "Machine Learning for Post-Event Timeline Reconstruction," *PGnet* (January 2006), pp. 1–4, 2006.

27. J.-S. Kim, D.-g. Kim, and B.-N. Noh, "A fuzzy logic based expert system as a network forensics," in *Proceedings. 2004 IEEE International Conference on Fuzzy Systems*, **2**, pp. 879–884 vol.2, 2004.

28. T. Lebo, S. Sahoo, D. McGuinness, K. Belhajjame, J. Cheney, D. Corsar, and E. Al., "PROV-O: The PROV Ontology," 2013.

29. O. J. Lee and J. E. Jung, "Sequence Clustering-based Automated Rule Generation for Adaptive Complex Event Processing," *Future Generation Computer Systems* **66**, pp. 100–109, 2017.

30. K. K. Muniswamy-Reddy, D. a. Holland, U. Braun, and M. I. Seltzer, "Provenance-Aware Storage Systems," in *USENIX Association Annual Technical Conference*, **pages**(TR-18-05), pp. 43–56, 2006.

31. J. S. Okolica, *Temporal Event Abstraction and Reconstruction.* PhD dissertation, AFIT, 2017.

32. J. S. Okolica, G. L. Peterson, and R. F. Mills, "Temporal Event Abstraction and Reconstruction." 2017.

33. J. Olsson and M. Boldt, "Computer forensic timeline visualization tool," *Digital Investigation* **6**(SUPPL.), pp. S78–S87, 2009.

34. P. Pannarale, D. Catalano, G. De Caro, G. Grillo, P. Leo, G. Pappad, F. Rubino, G. Scioscia, and F. Licciulli, "GIDL: a rule based expert system for GenBank Intelligent Data Loading into the Molecular Biodiversity database," *BMC Bioinformatics* **13**(Suppl 4), p. S4, 2012.

35. M. T. Pereira, "Forensic analysis of the Firefox 3 Internet history and recovery of deleted SQLite records," *Digital Investigation* **5**(3-4), pp. 93–103, 2009.

36. Pew Research Center, "Pew Research Center: Mobile Fact Sheet," 2016.

37. I. Robinson, J. Webber, and E. Eifrem, *Graph Databases*, O'Reilly Media, Inc., 2nd ed., 2015.

38. M. A. Rodriguez and P. Neubauer, "The Graph Traversal Pattern," *Computing Research Repository* , pp. 1–18, 2010.

39. B. Schatz, G. Mohay, and A. Clark, "Rich Event Representation for Computer Forensics," *Asia Pacific Industrial Engineering and Management Systems APIEMS 2004* (April 2016), pp. 1–16, 2004.

40. M. Stacey and C. McGregor, "Temporal abstraction in intelligent clinical data analysis: A survey," *Artificial Intelligence in Medicine* **39**(1), pp. 1–24, 2007.

41. B. Turnbull and S. Randhawa, "Automated event and social network extraction from digital evidence sources with ontological mapping," *Digital Investigation* **13**, pp. 94–106, 2015.

42. C. Vicknair, M. Macias, Z. Zhao, and X. Nan, “A comparison of a graph database and a relational database: a data provenance perspective,” *Proceedings of the 48th Annual ACM Southeast Regional Conference* , 2010.

43. M. Yasin, A. R. Cheema, and F. Kausar, “Analysis of Internet Download Manager for collection of digital forensic artefacts,” *Digital Investigation* **7**(1-2), pp. 90–94, 2010.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704–0188

1. REPORT DATE *(DD–MM–YYYY)* 22–03–2018	**2. REPORT TYPE** Master's Thesis	**3. DATES COVERED** *(From — To)* Sept 2016 — Mar 2018

4. TITLE AND SUBTITLE Digital Forensics Event Graph Reconstruction	**5a. CONTRACT NUMBER**
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Schelkoph, Daniel, J, Capt, USAF	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering and Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765	**8. PERFORMING ORGANIZATION REPORT NUMBER** AFIT-ENG-MS-18-M-058
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) DoD Cyber Crime Center 1190 Winterson Rd Linthicum, MD 21090 POC: Eoghan Casey Email: eoghan.casey@dc3.mil	**10. SPONSOR/MONITOR'S ACRONYM(S)** DC3/DCCI
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION / AVAILABILITY STATEMENT

DISTRIBUTION STATEMENT A:
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

14. ABSTRACT

Ontological data representation and data normalization can provide a structured way to correlate digital artifacts. This can reduce the amount of data that a forensics examiner needs to process in order to understand the sequence of events that happened on the system. However, ontology processing suffers from large disk consumption and a high computational cost. This paper presents Property Graph Event Reconstruction (PGER), a novel data normalization and event correlation system that leverages a native graph database to improve the speed of queries common in ontological data. PGER reduces the processing time of event correlation grammars and maintains accuracy over a relational database storage format.

15. SUBJECT TERMS

graph databases, labeled property graphs, ontology, forensics, event abstraction

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**			Dr. Gilbert Peterson
U	U	U	U	87	**19b. TELEPHONE NUMBER** *(include area code)* (937) 255-3636; gilbert.peterson@afit.edu