Graph Techniques for Next Generation Cybersecurity

by Benjamin Bowman

B.S. in Computer Engineering, December 2013, University of Maryland, College Park

A Dissertation submitted to

The Faculty of
The School of Engineering and Applied Science
of The George Washington University
in partial satisfaction of the requirements
for the degree of Doctor of Philosophy

November 3, 2021

Dissertation directed by

H. Howie Huang
Professor of Electrical and Computer Engineering

Graph Techniques for Next Generation Cybersecurity

Benjamin Bowman

Dissertation Research Committee:

H. Howie Huang, Professor of Electrical and Computer Engineering, Dissertation Director

Tarek El-Ghazawi, Professor of Electrical and Computer Engineering, Committee Member

Guru Venkataramani, Professor of Electrical and Computer Engineering, Committee Member

Ahmed Louri, Professor of Electrical and Computer Engineering, Committee Member

Suresh Subramaniam, Professor of Electrical and Computer Engineering, Committee Member

Aylin Caliskan, Assistant Professor of Computer Science, Committee Member

Abstract

Graph Techniques for Next Generation Cybersecurity

Since the dawn of the Information Age, we have increasingly relied on the technology that we develop to support nearly every aspect of modern living. At this point in our history, we are dependent on complex computer systems to manage and maintain the way our society and our world operates. Computers manage our transportation infrastructure, energy production, agriculture, political elections, military operations, financial markets, and many other critical processes that we all rely on. It is imperative that we keep these systems safe from attack.

Existing cybersecurity techniques depend heavily on identifying patterns or signatures of known-malicious activity, which are relatively easy to circumvent. Exacerbating this problem is the heavy reliance on the manual correlation of security events to determine the full scope of an attack, a task that is time consuming and error prone. Unfortunately, our adversaries are successfully overcoming these defenses far too frequently, as indicated by the many high-profile cybersecurity attacks of the last decade. Our cyber defenders need new tools and techniques for utilizing the wealth of data generated in modern computing environments to more effectively detect and mitigate cyber threats before they can cause harm.

In this dissertation, we will discuss some of the most critical cybersecurity challenges, and introduce novel techniques for detecting malicious activity in a variety of security contexts. Specifically, we will look at three graph-based techniques for generating new insights from existing data sources available in the cybersecurity domain. First, we will discuss the problem of automated software vulnerability detection, and introduce a technique which moves beyond signature-based detection to identify new vulnerabilities in source code based on a flexible and robust graph similarity metric. Second, we will discuss the problem of identifying malicious authentication activity within a computer network, and

introduce a technique which utilizes unsupervised graph machine learning to detect this critical stage of a cyber attack. Third, we will discuss a system which combines aspects of the previous two works into a network-wide monitoring and detection capability based on graph data structures and algorithms capable of detecting full-scale APT attack campaigns. Each work presented provides motivation for, and justification of, a graph-first approach to specific cybersecurity challenges. Through these works we hope to show how graph data structures and graph algorithms are a heavily under-utilized resource that cybersecurity defenders stand to benefit from.

Table of Contents

List of Figures

List of Tables

Chapter 1: Introduction

Cybersecurity is an exceedingly challenging problem for many reasons. First, the attack surface is vast, and increasing daily, resulting in an ever changing threat landscape. For example, as we continue to write and produce software, we provide a steady stream of software vulnerabilities which can be exploited by adversaries. In 2020 alone there were over 17,000 new vulnerabilities reported to the National Vulnerability Database [4]. Additionally, we continue to expand the attack surface as we add devices to our networks, such as the infestation of "Internet-of-Things" (IoT) devices, which provide adversaries a multitude of typically low-security entities which can be used to perform cyber attacks, such as the highly publicized Mirai botnet [60].

To make matters worse, our cyber defenders are inundated with a deluge of cyber-relevant data, and tasked with the largely manual effort of differentiating benign events from malicious. Many security appliances generate low-level security event information which can easily overwhelm cyber analysts causing what has been dubbed as "alert fatigue" [28]. This can contribute to critical security events getting lost in the sea of false positives, such as in the high-profile Target hack of 2013 [100].

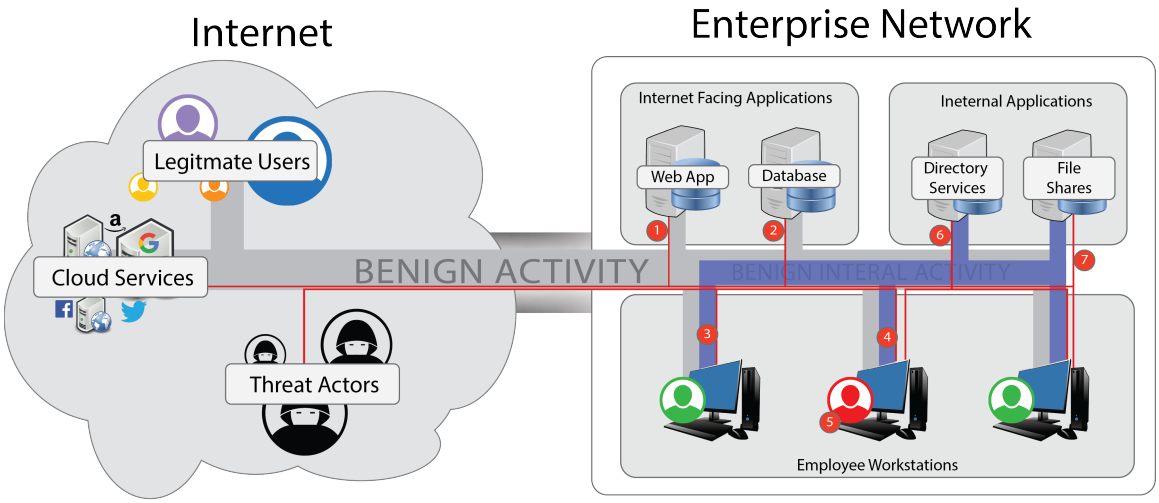
Our cybersecurity professionals are in need of new tools and techniques which can better utilize the vast and diverse quantities of cyber-relevant data to provide actionable and accurate insights to aid in the defense of our most critical information networks.

1.1 Background

In this section we will introduce some key concepts of cybersecurity, as well as graph data structures, and how we combine these two concepts to perform various tasks for improved cybersecurity.

1.1.1 Cybersecurity

Cybersecurity threats span essentially all domains of computing and technology. In this work, we will be focusing on the types of threats and vulnerabilities concerning business enterprises, and the large and complex computer networks that they manage to support their operations. Examples of these types of enterprises could be banking institutions, government institutions, universities, hospitals, etc. Essentially all modern enterprise environments will manage or access some computing resources, and will be subject to many of the threats we discuss in this work.



1. Buffer Overflow - a software vulnerability that can allow remote code execution to attackers
2. SQL Injection - a software vulnerability that exposes unsecure access to SQL databases
3. Trojanized Email Attachment - a weaponized file (e.g. .pdf, .doc) sent as an attachment to unsuspecting victims
4. Malware Execution via Watering Hole Attack - hosting malware on sites frequented by enterpise users
5. Insider Threat - the persistent threat from internal employees who may wish to do harm to the company
6. Lateral Movement via Kerberoast - Exploiting internal directory service vulnerabilities to move to new machines
7. Data Exfiltration - moving sensitive internal data to accessble locations outside of the enterprise network

Figure 1.1: An example of the many possible ways threats can manifest in enterprise computer networks

Figure 1.1 provides a high level example of a typical enterprise computer network, and the various ways in which threats can manifest. On the left of the figure we can see entities that exist outside of the enterprise network, including things like legitimate

users who may be accessing resources available from the enterprise, as well as the various cloud services that may be used by the enterprise or accessed by enterprise users (e.g., Google, Amazon, Twitter, etc). In addition to all of the benign entities on the Internet, the Internet also provides access to various threat actors. These can range from low-level "script kiddies", or individuals who are able to run standard hacking tools, up to highly advanced and sophisticated "Advanced Persistent Threats", or APTs, which are organized hacking groups that are typically associated with governments and nations.

On the right side of the figure we can see an example of an enterprise computer network. The enterprise has some Internet-facing applications, in the form of a web application, and a database. Because these elements are exposed to the Internet, they are highly vulnerable to attack. The other entities inside the network would typically have very limited, if any, visibility from outside of the internal enterprise network. Despite this fact, we will discuss ways in which an adversary could still compromise every system inside the internal enterprise network.

At the bottom of the figure is a list of example threats and vulnerabilities that could be exploited by threat actors in order to compromise the enterprise network. The Internet facing applications will always be susceptible to attack via software vulnerabilities, such as (1) buffer overflows or (2) SQL injection, which allow attackers to gather information and even run arbitrary code on the vulnerable systems. Additionally, the workforce will always be vulnerable to the highly effective attacks targeting users. For example (3) phishing campaigns, where users are emailed trojanized files which contain malware, or (4) watering hole attacks where threat actors will host malware on external resources that enterprise users unknowingly download and execute. There is also always the possibility of (5) insider threats, where legitimate employees of an organization, either for malice or personal gain, decide to perform cyber attacks from within the enterprise network. Once an adversary gains a foothold inside the network from any of the techniques mentioned previously, there are many new attack vectors now available. Typically internal assets that would not be exposed

to the Internet are significantly less hardened and secure. An adversary could exploit the directory services of the enterprise, which manages users and credentials within the network, to (6) move laterally to new systems via attacks such as the formidable Kerberoast attack. Finally, one possible result of the cyber attack could be something like (7) data exfiltration, where sensitive files and information are moved outside of the enterprise network, and either published publicly, or sold to the highest bidder.

The goal of the cybersecurity defenders is to minimize and mitigate risk as much as possible. In this work, we will be focusing primarily on two problem domains: software security with the goal of identifying vulnerabilities in program source code such that they can be remediated prior to exploitation, and network security with the goal of identifying adversaries that have gained a foothold inside of our computer networks such that we can eradicate them before they can cause damage.

1.1.2 Graph

Cybersecurity data is multi-modal due to the many diverse, yet related, sources from which it is generated. For that reason, our techniques in this work always rely on a robust graph-based representation of the underlying data. A *property graph* $G = \{V, E, P\}$ is an abstract data structure characterized by a set of nodes or vertices V, a set of edges or relationships E between the elements in V, and a set of properties P which characterize individually each node or edge in G. Figure 1.2 shows a simple property graph with 5 nodes and 4 edges, encoding some information on how a user interacts with a computer system. We can see various node type properties such as node 2 which is identified as a *system* node, node 1 which is identified as an *employee* node, and node 3 as a *program* node. We can also see various edge properties corresponding to different relationships between the nodes, such as the *authetnicates_to* edge between nodes 1 and 2, and the *runs_program* edge between 2 and 3. This simple graph also encodes an example of some potentially malicious activity, as we see node 3, which represents the *word.exe* program initiating network communication to

two external network nodes: *microsoft.com*, and *badguy.com*. The former is likely to be a benign license or update check, however the latter could indicate that *word.exe* has been compromised by, for example, a malicious email attachment containing a trojanized word document.

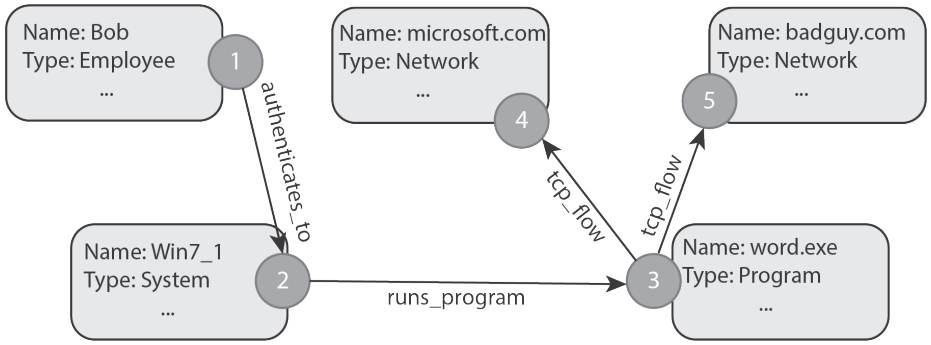


Figure 1.2: An example of a property graph containing various nodes and edges associated with cyber-relevant data sources.

Graph data is traditionally a computationally challenging data structure to maintain and analyze [63, 12, 72, 62, 73], . Traditional algorithms and data structures often rely on the property of locality, where data that is stored close together in memory is likely to be related to the data that is around it. Unfortunately, for graph data, this is not the case, as there is no inherent ordering to the data. Consider Figure 1.2 again, this is only one of infinitely many 2-D representations of the data contained in the graph. All of the nodes could be drawn in new locations, yet it would be the same graph provided that the edges maintain their respective connections. For this reason it is necessary that we pay special attention to the way we in which we build our graph data structures, write our algorithms, and manage our systems [48, 43, 71, 49]. Poor choices in any of these fundamental design considerations could lead to severe inefficiencies in storage, timing, or accuracy.

1.2 Applying Graph Techniques to Cybersecurity

Applying graph techniques to cybersecurity is a challenging yet fruitful task [47, 16, 40, 86, 39, 17, 15]. It requires expertise in both graph theoretic areas pertaining to graph structures,

graph systems, and graph algorithms, in addition to a strong working knowledge of how cyber attacks are carried out, what are the data sources of interest, and how to best detect activity related to the attacks.

Next, we will introduce three areas in which we have made contributions by applying graph techniques to solve challenging problems in cybersecurity: source code vulnerability detection, lateral movement detection, and APT attack campaign detection.

1.2.1 Detecting Vulnerabilities in Source Code

Software is written by humans and thus will always have the potential to be flawed. Software vulnerabilities are leveraged significantly by advanced offensive hacking organizations, or APTs, in order to gain an advantage over their cyber adversaries. These advantages often come in the form of "zero-days", or vulnerabilities in software that are unknown and undisclosed to the public. To illustrate how effective zero-days can be, consider in 2017 when the hacker group called the Shadow Brokers released the hacking tools of the Equation Group, who are widely suspected to be associated with the National Security Agency, which contained several zero-day exploits. One of these potent exploits was the now infamous Eternal Blue exploit which, even after having a patch released, caused devastation world-wide in the form of the WannaCry ransomware attack [21]. In order to stay ahead of the hackers, it is imperative that cybersecurity defenders proactively investigate and hunt for software vulnerabilities so that they can be patched prior to exploitation by malicious actors.

Vulnerabilities in software are often identified by simply matching software versions with that of known-vulnerable versions [107]. Source code analysis tools may perform deeper analysis and scan source code looking for vulnerable patterns of code based on known ground truth vulnerable samples. This is the task of *code clone detection*. Unfortunately, these techniques leave the majority of vulnerable code clones undetected due to the fact that they are only able to detect vulnerable code clones within a very slim margin of modification from the original sample.

In our work [15], we introduce VGRAPH, which is a graph-based system and representation of vulnerabilities in source code designed for efficient and flexible matching. A VGRAPHis a graph-based representation of the key characteristics that make a particular piece of code vulnerable, or patched, to a particular vulnerability. We develop an efficient graph-based matching algorithm which matches VGRAPHs against unknown source code at various levels of abstraction, allowing for improved detection of vulnerable code clones that have been significantly modified from their original source code, while still retaining good resistance to false positive alerts.

1.2.2 Detecting Lateral Movement in Computer Networks

Advanced adversaries use a number of techniques, ranging from zero-day vulnerabilities to targeted phishing campaigns, in order to establish a foothold inside of a computer network. However, after this foothold is established, an adversary is typically required to engage in many more actions prior to achieving their goals. One of these actions is the act of *lateral movement*, whereby an adversary is required to move laterally through a network in order to access systems or acquire permissions necessary to accomplish their mission. This is very challenging to detect, as often an adversary will utilize credentials gathered from the environment, as well as legitimate authentication channels and tools in order to perform the lateral movement.

Network security, similar to software security discussed previously, relies heavily on detection of signatures of known-malicious activity [102]. This means, if the adversary is using tools that are native to an environment (for example Windows Management Instrumentation, or WMI, a tool used for legitimate remote access in Windows domains), traditional network intrusion detection systems are likely to miss this activity entirely, or it will be buried among thousands of related benign events.

In our work [17], we introduce a technique based on unsupervised graph machine learning, or graph AI, for the purpose of detecting lateral movement entirely without

signatures, and instead using the knowledge learned from the authentication activity of the monitored network. We build an authentication graph data structure from industry standard network logs, and develop an algorithm which learns to predict authentication events between entities in our authentication graph. Our anomaly detection algorithm is capable of identifying anomalous authentication events with improved accuracy over traditional rule-based detection algorithms, as well as traditional machine learning algorithms that do not utilize the authentication graph data structure.

1.2.3 APT Attack Campaign Detection

Full scale APT attacks will include zero-day exploits, lateral movement, as well as a myriad of other activities that are performed by the attackers involving many systems, users, and how these entities interact. In order to successfully identify and eradicate APT-level attack campaigns, cybersecurity analysts must identify related malicious activity that can span both time and space, and differentiate this activity from the vast quantities of similar benign activity.

As mentioned previously, signatures are the main way malicious activity is detected, such as by detecting malicious byte streams over network connections, or a malicious file hash on a system. In order to move beyond signature based detentions, behavioral analytics are used to identify malicious behaviors in order to detect wrongdoing. However, in practice, these are often far too granular which results in 'alert fatigue', or they are too strict and require a human analyst to define an attack pattern for new threats, which is a brittle and time-consuming approach.

In our work [14], we introduce an APT attack detection technique based on identifying anomalous connected components in our Cyber Activity Graph data structure. By ingesting granular host-level event data into a single graph data structure, we are able to accurately correlate related events together in terms of the users and systems involved in the activities, as well as the types of activities that are performed. We developed a novel anomaly detection

algorithm based on identifying temporally and structurally anomalous behaviors in our Cyber Activity Graph, and show how these clusters of anomalous activity can be used to identify entire attack campaigns in two real-world datasets.

1.3 Outline

The remainder of this dissertation is organized as follows. In Chapter 2, we will discuss in detail our graph-based technique for detecting vulnerable functions in source code. In Chapter 3, we will discuss our graph model and unsupervised graph AI algorithm used for learning patterns of authentication interaction in enterprise computer networks. In Chapter 4 we will discuss our graph model and anomaly detection algorithms for identifying APT attack campaigns. In Chapter 5 we will conclude and discuss some future research directions.

Chapter 2: Robust Vulnerable Code Clone Detection

Computer software is written by humans, and thus is always at risk of containing flaws and vulnerabilities which can be exploited by our adversaries. In the last five years alone, there have been over 60,000 new additions into the National Vulnerability database [4], including the vulnerability exploited in the high profile Equifax hack of 2017, which exposed personal data of over 145 million Americans [90]. In a time when computer software plays such a critical role in our world, it is imperative that we find a way to identify vulnerabilities in source code before they are exploited by cyber adversaries.

In this chapter, we will introduce a technique for identifying vulnerable code clones with a higher degree of modification than previously possible, allowing for defenders and software maintainers to identify and patch code before it can be exploited for malicious intent.

2.1 Introduction to VGraph

Exacerbating the problem of vulnerable code is the growth of popular open-source software packages distributed freely on the Internet. At the time of writing this, the most popular source code repository GitHub [2] has over 20 million public source code projects, and 36 million registered users. The purpose of open-source code is to allow for the open distribution and reuse of computer software. Unfortunately, this leads to an increase in vulnerable code clones, which occur when unknowingly vulnerable code is copy-pasted from one location to another. When the vulnerability is discovered and patched, there is no guarantee that all occurrences of that vulnerability in all other locations within and across various projects and versions are patched as well. This means the source code with the vulnerable code clones will likely go unpatched, leaving them at risk for malicious exploitation.

Existing techniques for vulnerable code clone detection fall into two main categories: *code similarity* [66] [57] [51] [67] [75] [25], and *functional similarity* [68] [111] [99] [113] [114] [5] [110]. In code similarity approaches, target source code is compared against a set of known vulnerable code samples, and determined to be vulnerable if a threshold of similarity is met. Code similarity approaches are typically classified based on four types of detection coverage [96]: type-1 (identical), type-2 (syntactically equivalent), type-3 (syntactically similar), and type-4 (semantically similar). Existing code similarity techniques perform well when detecting identical (type-1) or syntactically equivalent (type-2) code clones, but suffer when the code has increased modification, such as the addition and deletion of lines of code (type-3 and type-4).

On the other hand, functional similarity approaches seek to generate abstract functional patterns of code which model vulnerable behavior. If the functional patterns are simple, the techniques suffer from low accuracy as they generate many false positives. Conversely, if the functional patterns are complex, they have the capability to identify vulnerable code clones with significant modifications, up to and including type-4 code clones. However, due to the complexity of building such a pattern, these techniques are typically specialized to only a small class of vulnerabilities, or to a particular source code project, rendering them ineffective as general-purpose vulnerable code clone detection techniques.

A recent study [45] of 35 software projects found that 50-60% of all vulnerable code clones were the result of type-3 syntactically similar code clones. However, existing techniques suffer to detect type-3 code clones, as they are either too strict, covering only identical or near-identical code clones, or too narrow, spanning only a few vulnerability classes or source code projects.

In this work we introduce VGRAPH, a code-similarity style technique which is capable of identifying highly modified vulnerable code clones, while remaining generic to all vulnerability types. VGRAPH abstracts vulnerabilities in source code to the graph domain, allowing for the ability to identify key relationships between textual elements that are not

directly discernible from the text alone. Additionally, we utilize not only the vulnerable code, but also the patched code, to identify specific relationships in the graph that are tied directly to the vulnerable code segment, the patched code segment, and the contextual code of a particular vulnerability. By separating the vulnerability representation into these three components, we are able to develop our matching algorithm to tolerate modifications at each level independently, providing more robust detection of modified vulnerable code clones.

We build a database of VGRAPHs by mining vulnerable and patched source code for 8 popular open source projects from GitHub [2], resulting in VGRAPHs for 711 vulnerabilities (CVEs) spanning 51 vulnerability types (CWEs). We download an additional 5,566 vulnerable and patched code clones from different versions of the source code as our test dataset. Our evaluation shows that VGRAPH is able to accurately identify vulnerabilities in the test dataset with an F1 score of 97%. When detecting highly modified vulnerable code clones, VGRAPH is able to achieve an F1 score of 85% compared to 74% and 50% by state-of-art vulnerable code clone detection systems ReDeBug [46] and VUDDY [57] respectively.

To evaluate real-world applicability, we utilize the VGRAPH system to identify previously unknown vulnerable code clones. We apply our method on several versions of popular software packages FFmpeg and OpenSSL, and identify 10 vulnerable code clones that were silently patched and are not listed in the NVD.

In summary, we make the following contributions:

- A novel graph-based source code vulnerability representation containing key relationships between the vulnerable code, the patched code, and the vulnerability context, which is generic to all vulnerability types.

- A matching algorithm capable of identifying highly modified vulnerable code clones while still retaining the ability to differentiate between vulnerable and patched code samples.

- A framework for generating VGRAPHs in a data-driven and automated way allowing

for the methods to scale to newly published vulnerabilities with ease.

The remainder of this chapter is organized as follows. Section 2.2 will discuss the background and related work. Section 2.3 will provide a high level overview of the VGRAPH system. Section 2.4 will discuss our approach to modeling source code and our novel VGRAPH representation. Section 2.5 will discuss our technique for detecting vulnerable code clones. Section 2.6 will explain how we acquire our vulnerable code samples in an automated way. Section 2.7 will discuss our evaluation and experimental results. Section 2.8 will discuss some limitations and future work and Section 2.9 will conclude the work.

2.2 Background and Related Work

A vulnerability in source code can be defined as any weakness of the code which can be exploited to perform unauthorized actions. For example, Figure 2.1 shows a synthetic function foo both before (Figure 2.1a) and after (Figure 2.1b) a vulnerability was discovered and patched. Both versions of the function read some input into a variable x on line 2. Then, both compare that input value against some variable MIN, and if x is larger then they will proceed inside the conditional statement. Both versions then perform some transformation of x into the variable y. Next, in the vulnerable version of foo, the value of y is simply passed to the $output$ function. Differently, in the patched version of foo, the value of y is first compared against some variable MAX, and is only passed to the $output$ function provided that y is less than MAX. Based on both the vulnerable version and the patch version of the function, we can infer that the function $output$ is only defined on values less than MAX, and is not safe to use with values above that limit. Thus, the vulnerability in this case was the omitted upper-bounds check on the value passed to the function $output$.

When vulnerabilities are discovered in software, they go through a process where they are assigned a Common Vulnerability Enumeration (CVE) identifier. This uniquely identifies the instance of a particular vulnerability, and is tied to specific versions of a software product. Additionally, CVEs are associated with Common Weakness Enumeration

```
1  void foo() {
2    int x = intput();
3    if (x > MIN) {
4      int y = x * 10;
5      output(y);
6    }
7  }
```

(a) Vulnerable code as the call to $output(y)$ has an unchecked upper bound on the variable y.

```
1  void foo() {
2    int x = intput();
3    if (x > MIN) {
4      int y = x * 10;
5      if (y < MAX)
6        output(y);
7    }
8  }
```

(b) Patched code as the variable y is checked against the upper bound MAX prior to the call $output(y)$.

Figure 2.1: An example of a source code function in both the vulnerable state and the patched state.

(CWE) identifiers, which represent different classes of vulnerabilities, such as improper input validation (CWE-200), out-of-bounds read (CWE-125), and use-after-free (CWE-416).

2.2.1 Clone Type Taxonomy

To compare the coverage of code clone detection techniques, we use the standard clone type taxonomy as introduced in [96]:

Type-1: Identical code except changes to whitespace and comment lines.

Type-2: Syntactically identical code with modifications to identifiers, literals, types, whitespace, and comments.

Type-3: Syntactically similar code with addition and/or deletion of lines, as well as modification to identifiers, literals, types, whitespace, and comments.

Type-4: Syntactically different code with the same functionality (i.e., semantically similar)

Figure 2.2 shows an example of each type of code clone for the vulnerable function introduced in Figure 2.1a. The type-1 code clone has a single comment line added on line 2. The type-2 clone has the bounds check variable *MIN* renamed to *minimum* on line 3. The type-3 code clone defines an additional variable z and initializes it to the value of x on line 4. The Type-4 code clone has replaced the $y = x * 10$ multiplication statement instead with a series of 10 addition operations on lines 5-7. Note that each example represents a pure

```
1  void foo() {
2    // comment line
3    int x = intput();
4    if (x > MIN) {
5      int y = x * 10;
6      output(y);
7    }
8  }
```

(a) Type-1

```
1  void foo() {
2    int x = intput();
3    if (x > minimum) {
4      int y = x * 10;
5      output(y);
6    }
7  }
```

(b) Type-2

```
1  void foo() {
2    int x = intput();
3    if (x > MIN) {
4      int z = x;
5      int y = x * 10;
6      output(y);
7    }
8  }
```

(c) Type-3

```
1  void foo() {
2    int x = intput();
3    if (x > MIN) {
4      int y=0;
5      for(int i=0;i<10;i++){
6         y=y+x
7      }
8      output(y);
9    }
10 }
```

(d) Type-4

Figure 2.2: Example of type-1 through type-4 code clones of the source code introduced in Figure 2.1.

clone, meaning it only has the modification most associated with each type. However, based on the definitions, each clone type can also include the modifications associated with the types below it. For example, in the type-3 code clone, we could also rename the variable *MIN* to *minimum*, and it would still be considered a type-3 clone.

Type 1-3 clones can be thought of as clones which are textually similar, while type-4 clones are functionally similar. The related works can be broadly categorized based on these two similarity measures.

2.2.2 Textual Similarity Techniques

The textual similarity techniques generally involve dividing a program into individual units (e.g., files, functions, tokens, etc), and performing a similarity measurement against a set of known vulnerable code samples. In general, these approaches perform well at detecting type-1 and type-2 code clones, but fail to detect type-3 and type-4 code clones. For example, VUDDY [57] is a technique which relies on hashing vulnerable functions to allow for a

quick table lookup to determine if a target function is vulnerable. VUDDY is able to detect type-1 and type-2 vulnerable code clones, but it will fail to detect type-3 and type-4 clones.

As textual similarity techniques grow more abstract in order to detect type-3 vulnerable code clones, they often have a severe decrease in accuracy. This is due to the fact that a patched function is often itself a type-3 code clone of the vulnerable function. Therefore, any techniques that do not take into account information from the patch will suffer from the inability to differentiate between vulnerable code clones and patched code clones.

SourcerCC [97], CPMiner [66], and CCFinder [53] are all code clone detection techniques which tokenize source code and identify clones based on some measure of token overlap. In each case, however, their applicability to vulnerable code clone detection is inhibited since they do not take into account information from the patch. Similarly, DECKARD [51] builds abstract syntax trees (AST) from the code and identifies similar subtree structures as clones, but again suffers from the inability to differentiate between vulnerable and patched code.

On the other hand, a recent work ReDeBug [46] is a technique which does use the information in both the vulnerable code and the patched code. ReDeBug performs sequence based matching utilizing the *diff* files associated with a particular vulnerability. A *diff* file contains the lines that were explicitly modified during the transition of the code from vulnerable to patched, as well as some context code within close textual proximity. This allows ReDeBug to detect some type-3 clones, however if the code modification is near the location of the lines modified during the patch process, this technique will fail to detect the vulnerable clone.

Because there are so many different textual similarity techniques, each with their own strengths and weaknesses, VulPecker [67] developed a technique which identifies a vulnerability-to-similarity-algorithm mapping. This way each algorithm can be applied to the vulnerabilities to which they are best suited. However, this approach is still limited by the underlying accuracy of the similarity algorithms, and only achieves a recall score of

60%, meaning many vulnerable clones were left undetected.

2.2.3 Functional Similarity Techniques

The functional similarity techniques are markedly different from the textual similarity techniques discussed previously as they attempt to model functional patterns indicative of vulnerable code, rather than using the textual contents of the code directly. This means these techniques are better suited to identifying type-3 and type-4 vulnerable code clones. However, in general, these techniques are either exceedingly noisy with many false positives and false negatives, or very narrow in scope as they apply to only specific vulnerability types, or are tied to particular source code projects.

The simplest functional similarity approaches are based on manually defined patterns of functionality which have been deemed vulnerable or unsafe based on software security experts. These patterns exist in many open-source vulnerability discovery tools such as FlawFinder [110] and the Rough Auditing Tool For Security (RATS) [5]. Due to the simplicity of the functionality defined in the patterns, these techniques often have many false positives and false negatives. In addition, they require an expert to manually define the functional patterns of vulnerable code.

Other techniques resort to identifying anomalous functionality as a proxy for vulnerable functionality. Yamaguchi et al. [115] [112] identifies anomalous input validation routines according to other functions in the same code repository. Chang et al. [20] identifies similar missing conditions by mining program dependence graphs (PDG) and identifying outliers. Not only are these techniques tied to a specific class of vulnerability, but their results are based on individual code repositories, and information gleaned from one project is likely not applicable to another.

In other related works, Yamaguchi et al. [114], [113] [111] extrapolates known vulnerabilities by mining information from Abstract Syntax Trees (AST) and Code Property Graphs (CPG). Based on a seed vulnerability, they use graph traversals to extrapolate to

new vulnerable code clones. While these techniques are capable of identifying type-4 code clones, they are highly coupled to the type of vulnerabilities being extrapolated, as well as the code repositories on which the analysis is run.

VulDeePecker [68] introduced a deep learning framework for learning the features necessary to identify vulnerable source code functionality. However this requires a robust dataset so that the machine learning algorithm can accurately learn the vulnerable functional pattern. Because of this requirement, their method was only evaluated on two vulnerability classes, and would require significant manual effort to extend to additional types.

2.2.4 VGRAPH Comparison

Comparatively, VGRAPH is a textual similarity vulnerable code clone detection system, with emphasis placed on accurately detecting type-3 code clones, while remaining generic to all vulnerability types. Our technique is most similar to ReDeBug, as we focus on the lines of source code that are modified during the patching process. However, unlike ReDeBug, we utilize a graph representation of the code rather than direct sequences of text, which allows for more robust detection of type-3 clones as the underlying text can change while the resulting graph structure remains the same. Our technique is generic to all vulnerability types and can be applied across many programs, contrasting the functional similarity approaches which often only target a few vulnerability classes.

2.3 VGRAPH System

Prior to discussing any individual component, we will first provide an overview of the VGRAPH vulnerable code clone detection system. A high level system architecture diagram is shown in Figure 2.3. The VGRAPH system consists of two distinct phases: a *Generation phase*, and a *Detection Phase*. During the *Generation phase*, source code repositories are mined in an automated way to identify references to known vulnerabilities, and the relevant source code is downloaded and cataloged. We download both the vulnerable source code,

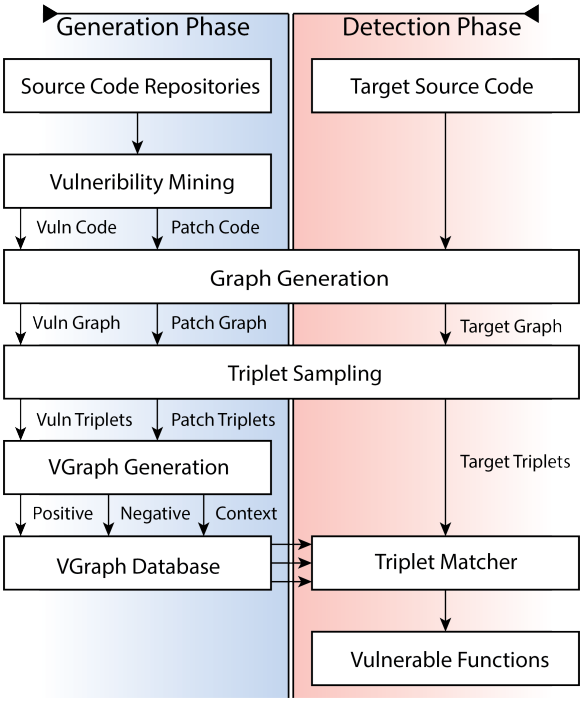


Figure 2.3: VGRAPH Vulnerability Detection System.

as well as the patched source code, as both are needed to accurately differentiate between vulnerable code clones and their patched counterparts. These pairs of vulnerable and patched code samples are then sent to the *Graph Generator*, which converts the raw source code into the highly expressive Code Property Graph (CPG) representation [111]. We utilize the open source utility *Joern* [3] to accomplish this. In order to avoid expensive graph matching, the CPGs of the vulnerable and patched code are then sampled via the *Triplet Sampler*, which converts the graphs into a set of code property triplets. These triplets are then sent to the VGRAPH *Generator*, which is responsible for generating the positive, negative, and context triplets of each vulnerability. This process is discussed at length in Section 2.4. These triplets and their corresponding vulnerability CVE identifier are then stored in the VGRAPH *Database*.

During the *Detection Phase*, target source code that is to be evaluated is first converted into CPGs via the *Graph Generator* and then sampled via the *Triplet Sampler* using the same pipeline as during the *Generation Phase*. The target triplets are then sent to the *Triplet*

Matcher, which performs the matching algorithm described in Section 2.5. This component will detect if any of the target functions are vulnerable code clones of any of the functions with VGRAPHs in the VGRAPH *Database*.

2.4 Vulnerability Representation

Determining a representation for vulnerable source code generally requires two main design choices. One is the level of granularity (e.g., program level, file level, function level, line level, token level), and the other is the source code representation (e.g., text, metric, tree, graph). We discuss our choices below.

Granularity Level. We choose to operate at the line level of granularity, meaning a vulnerability will be defined as a series of lines of source code. When vulnerable code is patched, the act of patching involves the addition, subtraction, or modification of specific lines of code. Thus, we believe operating at the line level of granularity is the most logical way to represent a vulnerability if we hope to be able to differentiate between vulnerable and patched functions, while also being able to detect highly modified code clones. Additionally, popular source code repositories such as GitHub [2] make it possible to acquire the source code for both vulnerable and patched code samples for a wide variety of programs. This allows us to acquire the specific source code lines which are added, removed, or modified for a wide variety of vulnerabilities in a reliable and automated way.

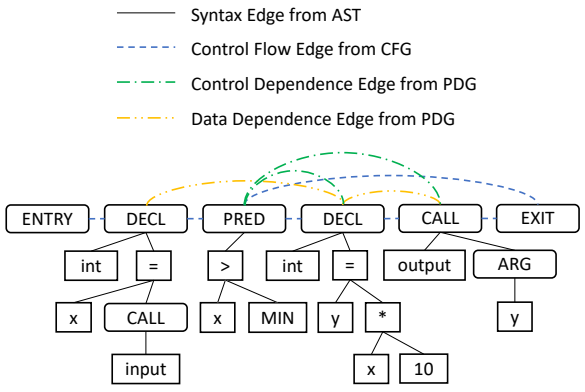


Figure 2.4: Example Code Property Graph for the source code function in Figure 2.1a.

Table 2.1: A sample of the positive, negative, and context triplets generated for the vulnerability in Figure 2.1.

Positive Triplets (PT)	Negative Triplets (NT)	Context Triplets (CT)
$(x > MIN, CONTROLS, output(y))$	$(y < MAX, CONTROLS, output(y))$	$(x = input(), DEF, x)$
$(y = x * 10, FLOWS\_TO, output(y))$	$(y = x * 10, FLOWS\_TO, y < MAX)$	$(y = x * 10, IS\_AST\_PARENT, x * 10)$
$(y = x * 10, FLOWS\_TO, Expression)$	$(y = x * 10, FLOWS\_TO, Condition)$	$(x = intput(), REACHES, y = x * 10)$

Source Code Representation. We choose to represent source code as a graph. Specifically, we choose to utilize the Code Property Graph (CPG) [111] which is a multigraph containing the representative nodes and edges from the Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Program Dependence Graph (PDG). Because vulnerabilities are highly complex, capable of manifesting in many different ways, it is important that we have a representation of source code that is capable of modeling a wide variety of complex relationships. The CPG is one of the most complex and expressive representations of source code available, as it has the ability to simultaneously model syntactic structure, control and data flow, and control and data dependence. An example CPG for the source code in Figure 2.1a is shown in Figure 2.4. We can see how the AST provides the general syntactic structure of the source code, tokenizing each statement and categorizing them as declaration statements (DECL), call statements (CALL), predicate statements (PRED), etc. The CFG then provides an ordering to the AST elements, identifying all the possible logic traversal paths, such as the path from the predicate statement to the variable declaration, or the function exit. Finally, the PDG provides information on control and data dependence between elements, such as the data dependence between the call to the *output* function and the previous variable declaration of y. This source code representation will allow us to extract relationships between source code elements that would not be directly discernible based on the textual contents alone. For a full discussion of all the various node and edge types sepresented in the CPG, we refer the reader to the original work [111].

2.4.1 VGRAPH Structure

Conceptually, the VGRAPH structure is the combination of elements extracted from the graph representation of both the vulnerable function as well as the patched function for a particular vulnerability. Each graph element is attributed with identifiers indicating if it was found in exclusively the vulnerable function, exclusively the patched function, or both. This way, the VGRAPH structure captures key relationships indicative of the vulnerability, the patch, and the necessary function context.

The first step in generating a VGRAPH for a particular vulnerability is to generate the Code Property Graph of both the vulnerable function as well as the patched function. A CPG is a directed, edge-labeled, attributed multigraph of the form $G = (V, E, \lambda, \mu)$ where V is a set of nodes, E is a set of directed edges, λ is an edge labeling function, and μ is a node property labeling function. We utilize the open source tool *Joern* [3] to accomplish this task.

At this point, we need to identify the overlapping graph elements so that we can extract the key relationships related to explicitly the vulnerable and patched code. As subgraph isomorphism is a computationally expensive NP-complete problem, we utilize a sampling technique where we convert the graphs into triplets of the form $(Source, Relationship, Destination)$, where *Source* is a source node property, *Destination* is a destination node property, and *Relationship* is the type of edge between these two nodes as found in the CPG. For each node in the graph, we extract the code property triplets as described in Algorithm 1. We

Algorithm 1 Triplet Sampler

```
1:  procedure TRIPLET_SAMPLER(G)
2:      triplets = []
3:      for n1 ∈ G.nodes do
4:          for n2 ∈ G.neighbors(n1) do
5:              for e ∈ G.edges(n1, n2) do
6:                  triplets.append(n1.code, e, n2.code)
7:                  triplets.append(n1.type, e, n2.code)
8:                  triplets.append(n1.code, e, n2.type)
9:                  triplets.append(n1.type, e, n2.type)
10:     return triplets
```

typically generate four separate triplets as seen in lines 6 through 9 of the algorithm for each edge in the CPG. Line 6 generates a triplet containing the textual source code contents. We could stop here, but this representation would not lend itself to type-2 and beyond code clones, so we add additional triplets with varying levels of abstraction. In Line 7 and 8, we abstract the source node and destination node respectively to their node types, rather than the textual contents. Finally, in line 9, we abstract both nodes to their type representation.

By generating the code property triplets in this manner, we are able to not only capture the relationship between the textual source code contents, but also the more abstract relationships between types of source code statements. This way, even if a piece of source code has textual modification, there will still be triplets containing relevant information in our VGRAPH structure.

We can now generate a VGRAPH based on the set of vulnerable code property triplets V extracted from the vulnerable function, and the patched code property triplets P extracted from the patched function. We define the three components of a VGRAPH as follows:

Positive Triplets (PT): This is the set of triplets from the vulnerable graph which are not found in the patched graph. Intuitively, this can be thought of as the specific relationships in the graph which contributed to it being vulnerable. Note that this is not strictly textual modifications, as textual modification will result in additional changes to the graph structure, which is explicitly captured by this approach. Formally, PT can be defined as:

$$PT = V \setminus P$$

Negative Triplets (NT): This is the set of triplets from the patched graph which are not found in the vulnerable graph. Intuitively, this can be thought of as the specific relationships of the graph which contribute to it being patched to a particular vulnerability. Formally, NT

can be defined as:

$$NT = P \setminus V$$

Context Triplets (CT): This is the set of triplets that are shared by both the vulnerable and the patched graph. Intuitively, these are the contextual relationships in the function that were not modified during the transition of the function from vulnerable to patched. As vulnerabilities are highly context dependent, this component is very important to represent the required context for the vulnerability to be present. Formally, *CT* can be defined as:

$$CT = V \cap P$$

These three structures combined represent a VGRAPH for a particular vulnerability. Table 2.1 provides a sample of the VGRAPH triplets generated for the example vulnerable and patched code in Figure 2.1. From the table we can see that the PT and NT accurately capture key information as to what relationships between source code elements are related to the function being identified as vulnerable vs. patched. In the first row we can see that the PT and NT capture the different control dependence relationships on the $output(y)$ call, with the source code $x > MIN$ controlling the $output(y)$ call in the vulnerable function, and $y < MAX$ controlling the $output(y)$ call in the patched function. Similarly, in the second row we can see that the PT and NT capture the different control flow relationships which occur after the initialization of the y variable, with control flowing directly to the $output(y)$ call in the vulnerable function, and to the bounds check condition $y < MAX$ in the patched function. In the third row, we see a similar relationship to the second row for the PT and NT, however this time more abstract. The PT here represents control flow from the initialization of the variable y to any expression statement, and the NT to any condition statement. This is an accurate, yet much more abstract representation of a key relationship that contributes to the vulnerability determination.

Differently, in all rows of the CT column we can see various general contextual information of the function. The first row provides some information on how the variable x is defined based on the call to the *input* function. The second row provides some syntax-related context between the declaration for the variable y and the $x * 10$ expression. The third row provides additional data dependence context between the declaration for y and the initialization of the variable x.

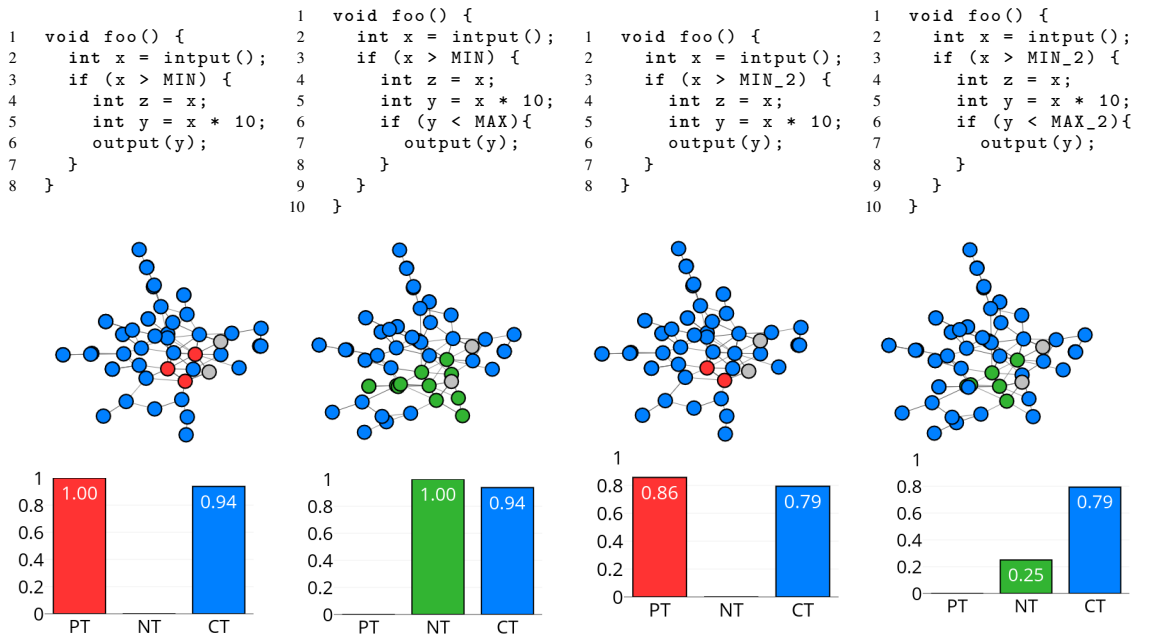
2.5 Vulnerability Detection

The goal of our detection algorithm is to provide an efficient way to utilize our VGRAPH representation to accurately identify vulnerable code clones ranging from exact clones to highly modified clones. This means we need an approximate matching algorithm which will not overwhelm our results with false positives. As our core VGRAPH representation is based on sets of graph triplets, we are able to use highly efficient set overlap operations to perform the bulk of the matching. Thus we develop a *Triplet Match* algorithm which we discuss below.

2.5.1 Triplet Match

The intuition behind our triplet matching algorithm is relatively straightforward. We expect vulnerable code clones of a particular vulnerability represented by a VGRAPH VG to have the following characteristics: (1) share many context triplets with VG, (2) share many positive triplets with VG, and (3) share few negative triplets with VG. To improve the ability to identify vulnerable code clones in the type-2 to type-4 range, we match triplets at each level independently (positive, negative, and context), and allow for some level of mismatch at each stage.

Algorithm 2 provides the pseudocode for our triplet matching algorithm. This algorithm takes as input a VGRAPH as well as graph triplets of an unknown target function, and produces a binary result indicating if the particular target function is detected as a vulnerable



(a) Vulnerable clone with single line added.

(b) Patched clone with single line added.

(c) Vulnerable clone with a variable renamed and line added.

(d) Patched clone with a variable renamed and line added.

Figure 2.5: Using a VGRAPH to detect type-3 vulnerable code clones and differentiate from their patched counterparts based on clones of the source code introduced in Figure 2.1.

code clone of the vulnerability represented by the VGRAPH. The *overlap* function in the algorithm is a simple set overlap routine which returns the ratio of the query triplets found in the target triplets. There are two thresholds in the algorithm, $thresh_C$ and $thresh_P$. These thresholds dictate the amount of mismatch permitted, and hence the approximation ability of the matching algorithm. These thresholds are determined experimentally based on our VGRAPH dataset in Section 2.7. Notice that there is no threshold for the negative triplets, as we found that simply comparing the negative triplet score against the positive triplet score provided the best performance.

We can see in Algorithm 2 that our matching routine proceeds in a very hierarchical nature. This is by design, as real world vulnerabilities are highly context dependent. Thus, we first match against the context triplets (line 2), and only continue pursuing a match if the CT score exceeds $thresh_C$ (line 3). Next we match against the positive triplets (line 4)

Algorithm 2 VGRAPH Vulnerability Detection

```
1: procedure ISVULNERABLE(VGraph, target)
2:     score_C = overlap(VGraph.CT, target)
3:     if score_C > thresh_C then
4:         score_P = overlap(VGraph.PT, target)
5:         if score_P > thresh_P then
6:             score_N = overlap(VGraph.NT, target)
7:             if score_N < score_P then
8:                 return True
9:     return False
```

and only continue if the PT score exceeds $thresh_P$ (line 5). The last step is to perform the negative triplet matching (line 6). If the NT score is less than the PT score (line 7) then a true result will be returned (line 8), indicating that the target function is a vulnerable clone of the VGRAPH. In all other cases the function will return false, indicating that the target function is not a vulnerable clone of the VGRAPH.

To illustrate the ability of this algorithm to detect vulnerable code clones, we return to the example source code provided in Figure 2.1, and the associated VGRAPH as discussed in section 2.4. Based on this code, we generated 4 code clones, two of the vulnerable function, and two of the patched function. Figure 2.5 shows the cloned source code, the CPG of the source code with nodes highlighted according to which elements of the VGRAPH they matched with, and the overlap score for each VGRAPH triplet component. Red corresponds to positive triplet matches, green to negative triplet matches, blue to context triplet matches, and gray to no match.

Figure 2.5a and Figure 2.5b show type-3 code clones of the original vulnerable and patched code respectively. In both cases, a new variable declaration is made on line 4. We can see from the highlighted graph structure and the overlap scores, that the VGRAPH for the original vulnerability matches significantly differently against these two very similar functions. Despite the code modifications, all positive triplets and none of the negative triplets matched in the vulnerable function, and all of the negative triplets and none of the positive triplets matched in the patched function. This means, not only would our detection

algorithm be able to accurately detect these type-3 vulnerable and patched code clones, but also, there is significant room for additional modification to the function while still maintaining the ability to detect this vulnerability.

Figure 2.5c and Figure 2.5d show another more complex type-3 code clone pair, this time with an additional type-2 style modification to the variables used in the critical bounds checks. The variable *MIN* has been replaced with *MIN_2* in both the vulnerable and patched functions, and *MAX* with *MAX_2* in the patched function. Despite this increase in modification, we can see that the VGRAPH was again able to identify many of the critical elements of the vulnerability in the vulnerable clone, and none of them in the patched clone. There was significantly less negative triplet matching in this patched function, however the NT score was still higher than the PT score, and the PT score was nearly 0%, indicating that our detection algorithm would have properly labeled this function as not-vulnerable.

It is also important to notice that in all four of these examples the CT scores remained very high, indicating that the required context was present for these vulnerabilities to occur. This example shows that, with appropriate thresholding on the positive, negative, and context triplet overlap scores, the VGRAPH structure and triplet matching algorithm is able to accurately identify the vulnerable code clones, and, importantly, differentiate from their highly similar patched counterparts.

2.6 Vulnerability Mining

Another important consideration is how to acquire samples of vulnerable and patched source code. Many related works manually generate a dataset of vulnerable code samples, and, because of this, often only cover a small number of programs and/or vulnerability types. In addition, there is likely some bias introduced on behalf of the researcher as to what samples are added to the vulnerable code dataset.

We believe it is important to have an automated way to generate vulnerable source code samples for a wide range of programs and vulnerability types. Only when this is the case

will a code-similarity-based technique be able to keep up with the continuous flow of new and diverse vulnerabilities. Therefore, we utilize an approach similar to [57] and mine content from the popular open-source code repository GitHub [2]. Note that any version control repository could be used, provided there is the ability to download specific versions of files, and there exist meaningful comments associated with the code modifications.

Version control software is widely utilized by developers of software projects both large and small as a way to manage and track changes to source code. They provide fine-grained and detailed information regarding what changed in the code, when, and why. We leverage this information to identify specific changes to source code that are related to security vulnerabilities. We developed a GitHub mining utility which downloads source code from before and after a change was made associated with a particular vulnerability identified by the CVE number. The samples from before the security-relevant code modification are labeled as vulnerable, and after, as patched.

It should be noted here that we make a fundamental assumption that modifications to source code files which reference CVEs are related to the process of patching the vulnerability. This is consistent with a popular related work VUDDY [57]. In addition, during our manual evaluation of several hundred commits related to the patching of vulnerabilities we found this assumption to hold.

2.6.1 GitHub Mining

We adopt a method very similar to [57] and mine samples of source code from the popular open-source software repository GitHub [2]. We identify code changes related to security vulnerabilities and download the code from the functions in their vulnerable state, as well as their patched state. Each step in the process is outlined below, accompanied by the *git* commands utilized to perform the actions.

Commit Log Parsing . In order to identify source code relevant to a specific vulnerability, we utilize the log messages associated with the commits to a GitHub repository. A commit

log is the message associated with some modification to the source code. It is intended to contain information relevant to the purpose for the code change. We identify the commits containing any reference to the string "CVE-20" as being commits related to the referenced vulnerability. This is accomplished with the following git command:

```
git log -grep="CVE-20"
```

File and Function Parsing. Each commit identified in the previous step uses a unique hash value as a commit identifier. For each commit, we use a second git command to show the details of that commit, which will include the files, functions, and locations of source code additions, deletions, and modifications. If the modifications happen inside a function of a C/C++ source code file, the hash value for both the original file and the modified file are identified so that we can use them to download the files in the next step. In addition, we parse out the modified function names so that we know what functions inside the source code files are of interest. The command below provides such details:

```
git show <commit_ID>
```

Source Code Download. At this point we have the hash ID for a source code file which contains our functions of interest for both the original vulnerable version, as well as the patched version. We can now use a final git command to download the source code files for both the vulnerable and patched version. We are then able to parse out the specific functions of interest which were explicitly modified during the transition from vulnerable to patched. The command below will show the contents of the source code file at a specific version of the code.

```
git show <file_ID>
```

The result is a set of source code function pairs, vulnerable and patched, each associated with a particular CVE which we can use to build our VGRAPH Database.

2.7 Evaluation

We utilize the typical metrics for accuracy comparison. True positives (TP) represent the number of functions which are identified as vulnerable, and are truly vulnerable. False positives (FP) represent the number of function which are identified as vulnerable, and are not vulnerable. True negatives (TN) are the number of results classified as not vulnerable and are truly not vulnerable. False negatives (FN) are those functions that are labeled as not vulnerable, but are truly vulnerable. Precision (P) is the ratio of the true positives to all classified positive functions: $P = TP/(TP+FP)$. Recall (R) is the ratio of true positives to all labeled positive functions: $R = TP/(TP+FN)$. F1 is an overall performance metric including both precision and recall: $F1 = 2/((1/P)+(1/R))$.

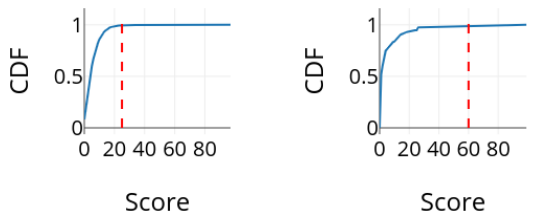
2.7.1 VGRAPH Database

We generated a VGRAPH Database based on 8 popular software packages that maintain source code on GitHub. Table 2.2 provides details of our VGRAPH database. In total we generated 1031 VGRAPHs for 711 unique vulnerabilities identified by their CVE number. For each CVE we determined the CWE, or vulnerability class, and found that our VGRAPH database covered 51 unique CWEs.

Table 2.2: VGRAPH Database Details

Repository	**CVEs**	**Functions**
Linux Kernel	197	269
OpenSSL	82	105
tcpdump	89	166
libtiff	31	40
FFMpeg	66	80
LibAV	58	72
QEMU	94	166
Xen	94	133
Total	711	1031

In order to determine the thresholds for our detection algorithm, we evaluated the



(a) Context Triplets (b) Positive Triplets

Figure 2.6: The overlap thresholds marked via the dashed red line for the context triples and the positive triples.

matching characteristics of each VGRAPH against all functions in our VGRAPH database, with the assumption that each VGRAPH should not detect vulnerable code clones in the vast majority of other functions. We want to set our thresholds to the minimum value to allow for the best ability to identify modified code clones, but not so low as to introduce a significant number of false positives. To that end, we compute the Cumulative Distribution Function (CDF) of the context triplet scores, and the positive triplet scores, for each VGRAPH against all other functions in the database. Figure 2.6 shows both of the computed CDFs. We set our context triplet threshold, $thresh_C$, at the minimum value where at least 99% of the true negatives fall below. This was a score of 25% as marked in the figure with the red dashed line. Similarly, for the positive triplets, at the given context triplet threshold of 25%, we again compute the CDF, and set our positive triplet threshold, $thresh_P$, at the minimum value where at least 99% of the true negatives fall below. This occurred at a score of 60% as marked in the figure by the red dashed line. This difference in threshold values makes sense, as the set of context triplets typically contains many more entries than the set of positive triplets for a particular vulnerability. This means we can afford a less strict threshold for the context matching compared to the positive matching. The referenced thresholds of $thresh_C = .25$ and $thresh_P = .60$ were used in the remaining experiments. In Section 2.7.7, we present a sensitivity analysis of these two parameters.

2.7.2 Test Dataset

In order to generate our test dataset, we require code clones of the functions in our VGRAPH database. According to a recent study [7], the average lifespan of a vulnerability in source code is 6.9 years. Thus, we assume that samples of the same function at different versions prior to the patch will still be vulnerable, provided they are well within the 6.9 year period. Similarly, we assume that versions of a function subsequent to the patching of a particular vulnerability will remain patched to that vulnerability. Therefore, we again utilize GitHub, and download samples of the functions in our VGRAPH database at six different time intervals relative to the patch date: immediately before, 1 month before, 1 year before, and immediately after, 6 months after, 1 year after. We temporally spaced our code samples in this manner in order to extract code clones with both small modifications (type-2 and below) to large modifications (type-3 and above), with the assumption being that temporally distant versions of the code are more likely to have a higher level of modification. We then label the functions as a vulnerable code clones if they were samples from before the original vulnerability-patching commit, or patched code clones if from after.

The details of the test dataset are listed in Table 2.3. Our test dataset includes 2,840 vulnerable code clones and 2,726 patched code clones, for a total of 5,566 code clones. In some cases the vulnerabilities used to build our VGRAPHs did not have commits to the same file at the various time intervals, which is the reason why these two numbers are not exactly equal.

2.7.3 Scoring Criteria

The results generated against our test dataset can be scored in two ways. In the simpler case, our dataset can be labeled with a binary output of either vulnerable or patched, in which case scoring the results is trivial. In the more complex case we can label each sample in our dataset on a per-CVE basis. However, this introduces a challenge, as each sample in our dataset is labeled only with the CVE with which it was generated. We found that, in many

Table 2.3: Test Dataset Details

Repository	Vulnerable Clones	Patched Clones
Linux Kernel	762	743
OpenSSL	310	288
tcpdump	370	349
libtiff	120	98
FFMpeg	220	233
LibAV	199	190
QEMU	483	463
Xen	376	362
Total	2840	2726

cases, a function associated with a particular CVE is also associated with other CVEs. It is also possible that some functions may in fact be unknown vulnerable code clones, and thus not be labeled in our test dataset.

To remedy this, we score CVE-aware techniques as follows. For every positive result generated, it is considered a true positive under the following conditions. **Condition 1**: The function is a labeled vulnerable code clone and its CVE matches that of the source vulnerability. This is the base case where the original vulnerable function was used to identify a vulnerable code clone generated from the same originating CVE. **Condition 2**: The code repository, file name, and function name match that of the source vulnerability, and the source vulnerability is *newer* than the target function. In this case, a CVE from a particular function identified the vulnerability in an older version of the same function which happened to be associated with a different CVE. **Condition 3**: Manual inspection was performed and it was determined that the result is in fact a vulnerable code clone of the source vulnerability. This accounts for true vulnerable code clones which can span functions and even repositories which we did not have a label for in our test dataset. We refer to matches associated with conditions 2 and 3 as *cross-CVE* clones as one CVE is used to identify a vulnerability in a function associated with a different CVE.

For the negative results, a false negative is scored for every known vulnerable code clone that failed to generate a result, and a true negative for every known patched code clone that

did not generate a result.

2.7.4 Detection Comparison

We compare with four state-of-art vulnerability detection techniques: FlawFinder [110], RATS [5], VUDDY [57], and ReDeBug [46]. FlawFinder and RATS are both open-source tools used in industry for identifying bugs in source code, and are both based on manually generated vulnerable functional code patterns. VUDDY is a vulnerable code clone detection technique which is based on the hashing of source code functions and a subsequent lookup of known vulnerable hashes. ReDeBug is a detection technique based on identifying sequences of known vulnerable and patched code harvested from the *diff* files associated with the patching process.

To compare with FlawFinder and RATS, we were able to download the respective tools, and run them directly on our test dataset. As these techniques do not contain any internal notion of CVEs, we can utilize the simple scoring criteria based on the binary label of the code samples.

To compare with VUDDY, we utilized their open web service [6] to identify all vulnerable code clones in our test dataset. In order to compare techniques fairly, we only calculate scores for CVEs that were shared between VUDDY and VGRAPH. This way we are comparing detection techniques rather than database generation techniques. To compare with ReDeBug, we were able to download the source code, generate the required *diff* files using the same functions used to generate the VGRAPH database, and apply the detection algorithm to the test dataset. As these three techniques are CVE-aware, we scored results of each technique independently based on the conditions stated in Section 2.7.3.

Our results from these experiments are listed in Table 2.4. We can immediately see that both of the functional pattern based techniques, RATS and FlawFinder, have fairly poor results, with F1 scores below 40%. This is due to the fact that these systems are based on manually generated, predefined patterns of functionality which fail to accurately reflect most

Table 2.4: Vulnerable Clone Detection Comparison

System	TP	FP	FN	P	R	F1
RATS	33	**23**	2807	59	1	2
FlawFinder	712	663	2128	52	25	34
VUDDY	2021	117	371	95	84	89
VUDDY*	2021	27	371	**99**	84	91
ReDeBug	3401	94	329	97	92	94
VGRAPH	**3824**	63	**147**	98	**96**	**97**

real world vulnerabilities.

Compared to RATS and FlawFinder, we can see that VUDDY, ReDeBug, and VGRAPH are all able to identify significantly more vulnerable code clones. Surprisingly, VUDDY did not immediately reveal itself as the most precise of the techniques. Upon investigation, we found that a small number of CVEs were alerting on a large number of functions, causing many false positives. For example, CVE-2017-11108, a vulnerability found in a single function of a single file in the *tcpdump* program, generated 342 results across 93 different functions. VUDDY uses a similar GitHub mining technique to generate their vulnerability hashes, and it is likely they mistakenly associated this particular CVE with an incorrect commit (likely a merge commit) covering many files unrelated to the actual vulnerability. If we disregard only two CVEs (CVE-2017-11108 and CVE-2017-5202), VUDDYs precision is over 99%, which is the level of precision we expected out of this hash-based technique. This test corresponds to the VUDDY* row in Table 2.4. In both scenarios, however, we can see that VUDDY performs the worst of the code similarity style techniques, with a recall score of only 84%. Also notice that VUDDY returns significantly less true positives than ReDeBug and VGRAPH, so the true recall is likely much less if we were to consider the cross-CVE results generated by the other techniques in the recall calculation.

ReDeBug, although performing with a slight reduction in precision compared to VUDDY, is able to identify over 1000 more vulnerable code clones than VUDDY, achieving 92% recall. ReDeBug is able to detect more vulnerable code clones because it only considers a localized context window around the location where the vulnerability was patched. This

allows ReDeBug to detect vulnerable code clones with a significant amount of modifications provided that they occur in areas outside of the context window.

The best performing vulnerable code clone detector was VGRAPH, with an F1 score of 97%. Notably, VGRAPH is able to achieve this high score while also returning the most true positives out of all of the techniques, returning over 400 more vulnerable clones than ReDeBug, and over 2500 more vulnerable clones than VUDDY. Because of this, VGRAPH achieves the highest recall, detecting 96% of the ground truth vulnerable code clones. This improved recall is due to VGRAPHs increased ability to accurately detect type-3 and type-4 vulnerable code clones. Because VGRAPH captures key relationships associated with the contextual code, vulnerable code, and patched code, and subsequently matches on each independently, VGRAPH is able to tolerate more modification than the comparison works.

2.7.5 Code Sensitivity Analysis

In order to understand the sensitivity of these techniques to modifications in both vulnerable and patched code clones, we compute accuracy metrics specifically for those samples in our dataset which we know have type-2 through type-4 modifications from the original vulnerable and patched functions. Thus, we only evaluate the results from our test set which were scored according to Condition 1 in the scoring criteria mentioned in Section 2.7.3, and do not consider the cross-CVE clones. This was in order to isolate the results to only those clones that we knew came from the exact same function, but at different versions in that function's lifetime. This is why there are significantly less results in this experiment.

Table 2.5: Modified Code Only Detection Comparison

System	TP	FP	FN	P	R	F1
VUDDY	254	**8**	496	**97**	34	50
ReDeBug	464	43	286	91	62	74
VGRAPH	**579**	40	**147**	94	**77**	**85**

Table 2.5 shows the results for this subset of the test data. We can now see where the

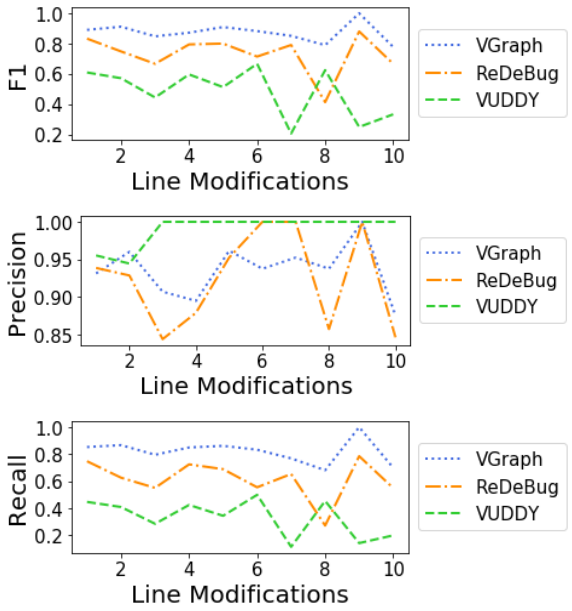


Figure 2.7: Accuracy comparison at different levels of modification in the code clones.

additional performance of VGRAPH is coming from. VGRAPH was able to identify twice as many modified vulnerable code clones as VUDDY, and over 100 more than ReDeBug. Note that VGRAPH is able to achieve this improved recall while still remaining very precise at 94%, only slightly less than VUDDY at 97%, and surpassing ReDeBug at 91%.

To further understand how modifications to the code affect detection accuracy, we compute accuracy metrics at explicit levels of modification to the original vulnerable and patched functions. We use the Linux utility *diff* to identify the number of line modifications between the test function and the original vulnerable or patched functions. Figure 2.7 plots the accuracy metrics for each of the techniques ranging from one line modification up to 10 line modifications.

We can see from these results again that VUDDY is the most precise of the techniques, however at the expense of recall. We can see that VGRAPH and ReDeBug are similar in terms of precision, with VGRAPH only showing a slight advantage across all levels of modification. However, VGRAPH is consistently the most performant with regards to the recall rate, having the highest score at each level of modification. This results in the F1

score for VGRAPH to outperform both techniques at all levels.

Next, we score the techniques based on code clone type. We use the following heuristics to label the clone type of each pair of functions. As all pairs of functions are the same function, but from different versions of the code, we consider each pair to be at least a type-4 code clone, as ultimately the function is the same in the greater application codebase, and thus likely providing similar functionality across versions. We differentiate between type-3 and type-4 clones by thresholding the number of line modifications. When the number of line modifications is greater than half of the overall function size, we consider this to be a type-4 code clone. To differentiate between type-2 and type-3 code clones, we consider any function pair where there exists a 1-to-1 mapping of lines that were modified to be a type-2 clone.

Figure 2.8 shows the results from this experiment. We can see that, as expected, type-4 clones are much harder to detect than type-2 clones. Across all clone types, VGRAPH again achieves the best F1 score due to an improved recall rate while maintaining a reasonably high precision.

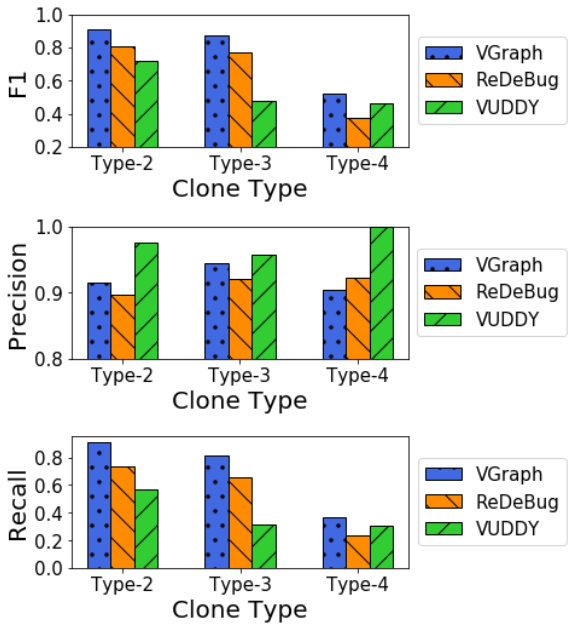


Figure 2.8: Accuracy comparison across code clone types.

```
if (ndo->ndo_vflag >= 1 &&
    ICMP_ERRTYPE(dp->icmp_type
    )){
  bp += 8;
  ND_PRINT((ndo, "\n\t"));
  ip = (const struct ip *)bp;
  snapend_save = ndo->
      ndo_snapend;
  ip_print(ndo, bp,
      EXTRACT_16BITS(&ip->
      ip_len));
  ndo->ndo_snapend =
      snapend_save;
}
```

(a) Vulnerable code with no check on attacker controlled variable *ip->ip_len*.

```
if (ndo->ndo_vflag >= 1 &&
    ICMP_ERRTYPE(dp->icmp_type
    )) {
  bp += 8;
  ND_PRINT((ndo, "\n\t"));
  ip = (const struct ip *)bp;
  snapend_save = ndo->
      ndo_snapend;
  ND_TCHECK_16BITS(&ip->ip_len)
      ;
  ip_print(ndo, bp,
      EXTRACT_16BITS(&ip->
      ip_len));
  ndo->ndo_snapend =
      snapend_save;
}
```

(b) Patched code with a single additional line calling *ND_TCHECK_16BITS*.

```
if (ndo->ndo_vflag >= 1 &&
    ICMP_ERRTYPE(dp->icmp_type
    )) {
  bp += 8;
  ND_PRINT((ndo, "\n\t"));
  ip = (const struct ip *)bp;
  ndo->ndo_snaplen = ndo->
      ndo_snapend - bp;
  snapend_save = ndo->
      ndo_snapend;
  ip_print(ndo, bp,
      EXTRACT_16BITS(&ip->
      ip_len));
  ndo->ndo_snapend =
      snapend_save;
}
```

(c) Vulnerable clone with a single line addition declaring *ndo->ndo_snaplen*.

Figure 2.9: Three versions of a code segment from the tcpdump function *icmp_print* associated with CVE-2017-13012. Only VGRAPH detected the vulnerable code clone.

2.7.6 Deep Analysis of Code Clones

In this subsection, we will analyze specific cases where VGRAPH and the comparison works diverge. Specifically, we found and evaluated three such cases where VGRAPH generated a true positive, false positive, and false negative, which were not shared by the comparison works. We will discuss each case in detail below.

First we will look at a case where VGRAPH accurately detected a vulnerable code clone that was missed by both VUDDY and ReDeBug. The vulnerability CVE-2017-13012 is a buffer over-read vulnerability in a packet processing routine of the common network utlity *tcpdump*. This vulnerability was given a score of 9.8 out of a possible 10 points, indicating that it is a very serious vulnerability. According to the NVD it affects all versions of *tcpdump* prior to version 4.9.2. This vulnerability allows an attacker to read raw memory content from the victim machine which could contain, among other things, usernames and plaintext passwords.

Figure 2.9 shows the same code segment from three different versions of the affected function. Figure 2.9a contains the source code as it existed when the vulnerability was discov-

ered. The key vulnerable code is related to the call to *ip_print* and *EXTRACT_16BITS(&ip->ip_len).* As the attacker controls the value of *ip->ip_len*, it is necessary to check the value of this variable prior to its usage. The patch to this vulnerability shown in Figure 2.9b involved a single line addition in which a call to *ND_TCHECK_16BITS(&ip->ip_len)* was added. Our test dataset contained a third version of this function, shown in Figure 2.9c, which existed 6 months prior to the discovery of the vulnerability. In the code clone we can see that the call to *ND_TCHECK_16BITS* function is omitted, indicating that this code is indeed vulnerable to CVE-2017-13012. Additionally, we see a single line addition *ndo->ndo_snaplen=ndo->ndo_snapend-bp* which does not affect the core vulnerable code, as it does not prevent the buffer over-read caused by the call to *EXTRACT_16BITS(&ip->ip_len)* and *ip_print*.

VGRAPH correctly identified the code clone in Figure 2.9c because it was able to detect key vulnerable conditions by matching positive triplets such as:

Source: *snapend_save=ndo->ndo_snapend*
Relationship: FLOWS_TO
Destination: *ip_print(ndo,bp,EXTRACT_16BITS(&ip->ip_len))* (2.1)

This triplet represents the condition of the vulnerable code where the declaration of the *snapend_save* variable flows directly to the call to the *ip_print* function.

Additionally, key negative triplets indicative of the patch failed to match, such as:

Source: *ND_TCHECK_16BITS(&ip->ip_len)*
Relationship: DOM
Destination: *ip_print(ndo,bp,EXTRACT_16BITS(&ip->ip_len))* (2.2)

This triplet represents the fact that a call to *ND_TCHECK_16BITS* must come before the call to *ip_print* in the patched version. The combination of high positive triplet score and low negative triplet allowed VGRAPH to properly label this function as a vulnerable code

```
pvscsi_dbg_dump_tx_rings_config
    (rc);
pvscsi_ring_init_data(&s->rings
    , rc);
s->rings_info_valid = TRUE;
return PROCESSING_SUCCEEDED;
```

(a) Vulnerable code with improper error checking on *pvscsi_ring_init_data*.

```
pvscsi_dbg_dump_tx_rings_config
    (rc);
if (pvscsi_ring_init_data(&s->
    rings, rc) < 0) {
  return PROCESSING_FAILED;
}
s->rings_info_valid = TRUE;
return PROCESSING_SUCCEEDED;
```

(b) Patched code with added conditional check on *pvscsi_ring_init_data*

```
if (!rc->reqRingNumPages
    || rc->reqRingNumPages >
        RINGS_MAX_NUM_PAGES
    || !rc->cmpRingNumPages
    || rc->cmpRingNumPages >
        RINGS_MAX_NUM_PAGES) {
  return PROCESSING_FAILED;
}
pvscsi_dbg_dump_tx_rings_config
    (rc);
pvscsi_ring_init_data(&s->rings
    , rc);
s->rings_info_valid = TRUE;
return PROCESSING_SUCCEEDED;
```

(c) Patched clone with a different check not based on *pvscsi_ring_init_data.*

Figure 2.10: Three versions of a code segment from the Qemu function *pvscsi_on_cmd_setup_rings* associated with CVE-2016-4952. VGRAPH generated a false positive on the patched clone.

clone.

Because of the 8 line-level modifications, the hash generated by VUDDY for this function would not match the hash of the original function, which is the reason why VUDDY did not detect this vulnerable code clone. ReDeBug would have been able to tolerate most of the modifications as they existed far away from the specific code associated with the vulnerability. However, the single line addition directly around the vulnerable code caused the ReDeBug sequence-based matching algorithm to fail and thus not detect this vulnerable code clone.

Next we will look at a case where VGRAPH generated a false positive not shared by ReDeBug and VUDDY. Figure 2.10 shows three versions of the same code segment from the Qemu program associated with CVE-2016-4952. The vulnerable code shown in 2.10a allowed an attacker to cause denial-of-service due to an out-of-bounds array access caused by improper error checking. The patched code shown in 2.10b involved adding a conditional statement checking the return value of the call to *pvsci_ring_init_data*, and returning a failed state when appropriate. Our test dataset contained a third version of the function from several months after the vulnerability was patched, shown in Figure 2.10c. We can see that

the code was updated to instead check for proper conditions prior to the function call to *pvcci_ring_init_data*. This meant that it was no longer necessary to check the return value of the call to *pvsci_ring_init_data*, and thus the conditional statement added in the original patch was removed.

Although the version of the code in Figure 2.10c is patched to the original vulnerability, VGRAPH incorrectly classified it as vulnerable to CVE-2016-4952. We can see that a large part of the code from the original vulnerable function exists in this patched version. This caused many positive triplets to match, such as:

Source: *pvscsi_ring_init_data(&s->rings, rc)*
Relationship: FLOWS_TO (2.3)
Destination: *s->rings_info_valid=TRUE*

This triplet represents the direct control flow from *pvscsi_ring_init_data* to the *rings_info_valid=TRUE*, and ultimately the successful return fo the function. Also, many of the negative triplets which would have indicated that this function was patched failed to match as well, such as:

Source: *pvscsi_ring_init_data(&s->rings,rc)<0*
Relationship: CONTROLS (2.4)
Destination: *return PROCESSING_FAILED*

This triplet describes the key control dependence in the patched code between the *return PROCESSING_FAILED* statement and the evaluation of the return value from the call to *pvscsi_ring_init_data*. This failed to match in the clone because the conditional statement no longer calls the *pvscsi_ring_init_data* function.

There were several key negative triplets that did in fact match, such as:

Source: *Condition*
Relationship: CONTROLS (2.5)
Destination: *return PROCESSING_FAILED*

This triplet represents a more generic relationship of the patch where there is a control de-

```
if (*p != 'Z')
  goto unsupported_time;
mon_len = month_lengths[mon];
...
if (year < 1970 ||
  mon < 1 ||
  mon > 12 ||
  day < 1 ||
  day>mon_len ||
  hour > 23 ||
  min > 59 ||
  sec > 59)
    goto invalid_time;
```

(a) Vulnerable code with an unsafe access to the *month_lengths* array.

```
if (*p != 'Z')
  goto unsupported_time;
if (year < 1970 ||
  mon < 1 || mon > 12)
    goto invalid_time;
mon_len = month_lengths[mon -
    1];
...
if (day < 1 ||
  day > mon_len ||
  hour > 23 ||
  min > 59 ||
  sec > 59)
    goto invalid_time;
```

(b) Patched code with conditions reordered to check value of *mon* prior to usage.

```
if (*p != 'Z')
  goto unsupported_time;
mon_len = month_lengths[mon];
...
if (year < 1970 ||
  mon < 1 ||
  mon > 12 ||
  day < 1 ||
  day > mon_len ||
  hour < 0 ||
  hour > 23 ||
  min < 0 ||
  min > 59 ||
  sec < 0 ||
  sec > 59)
  goto invalid_time;
```

(c) Vulnerable clone with a different conditional statement.

Figure 2.11: Three versions of a code segment from the Linux Kernel function *x*509*_decode_time* associated with CVE-2015-5327.

pendence between the *return PROCESSING_FAILED* statement and a conditional statement. Several triplets like this one matched in the patched code clone, however they were too few to score higher than the positive triplets. This caused VGRAPH to incorrectly label this function as vulnerable.

Because the patched code clone involves relatively significant changes to the code close to the location of the original patching process, both VUDDY and ReDeBug properly classify this function as not-vulnerable. It should be noted here that we could adjust the thresholds for the matching algorithm such that VGRAPH does properly classify this function as not vulnerable, although this would have an overall negative effect on the total performance of the algorithm, as we would generate many more false negatives.

Finally we will examine a case where VGRAPHgenerated a false negative. Figure 2.11 shows a case where VGRAPH generated a false negative not shared by ReDeBug. The Figure shows three versions of the *x509_decode_time* function found in the Linux Kernel. A vulnerability was found in the decode logic that allowed for a buffer over-read in the *month_lengths* array. The patch is shown in Figure 2.11b, which was to move some of the validation logic prior to the array access, as well as offsetting the access by 1. Figure 2.11c

shows a code clone of the vulnerable code from 4 months prior to the vulnerability being discovered. We can see in this version there are a few modifications to the lower conditional statement, however the key unchecked array access to *months_lengths[mon]* was performed, indicating this function is still vulnerable.

The main reason VGRAPH failed to detect this vulnerable code clone has to do with the fact that during the original patch process, code was removed from the large compound conditional statement at the bottom of the code excerpt, and moved to before the access of the month_lengths array. Because code was modified in the lower conditional statement, positive triplets were associated with the changes in the large conditional statement. Additionally, the way the CPG is generated for this function, the large compound conditional statement was treated as a single entity, rather than breaking it up into it's many constituent parts. This means there existed positive triplets such as:

Source: *mon_len=month_lengths[mon]*
Relationship: REACHES
Destination: *year<1970||mon<1||mon>12||day<1||day>mon_len||hour>23||min>59||sec>59* (2.6)

Source: *year<1970||mon<1||mon>12||day<1||day>mon_len||hour>23||min>59||sec>59*
Relationship: CONTROLS
Destination: *goto invalid_time* (2.7)

The first triplet describes the data dependence between the compound conditional statement and the declaration of mon_len, and the second describes the control dependence between the conditional statement and the *goto invalid_time* statement. Both of these triplets failed to match in the vulnerable code clone, due to the fact that there was a modification to the lower conditional statement. If the compound conditional statement were to be expanded into it's constituent parts, VGRAPH would have likely been able to detect this as a vulnerable code clone, as many of the components are the same in the vulnerable code clone.

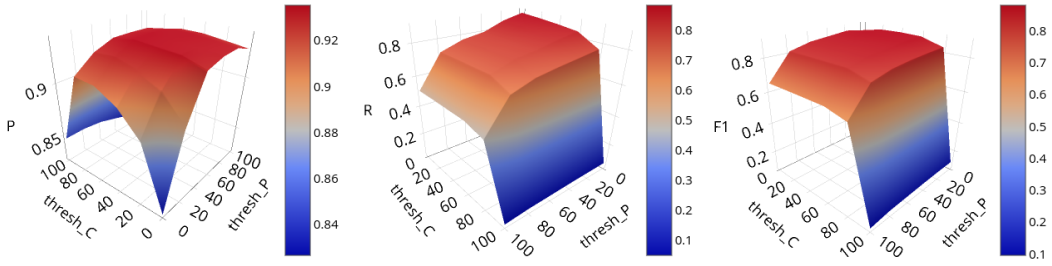
On the other hand, ReDeBug was able to properly classify this vulnerable code clone

since the modifications to the function were far enough away from the original vulnerable code. VUDDY, however, also failed to detect this function as vulnerable,as the function had changes from the original vulnerable code and the hashes would not match.

2.7.7 Parameter Sensitivity Analysis

In addition to external factors such as code modification, we were also interested in understanding how sensitive our technique was to internal factors such as our two algorithm hyperparamaters, identified as $thresh_C$ and $thresh_P$ in Algorithm 2. To that end, we used the same subset of our data as the previous sensitivity experiment, but this time varied these two hyperparameters from a threshold of 0 to a threshold of 100, and computed accuracy metrics for each configuration. Figure 2.12 shows surface plots for Precision, Recall, and F1 at each of the different threshold configurations.

In these plots we can see the obvious performance drop-offs at the extreme values of the thresholds. At threshold values of 100% for both PT and CT, our F1 score is near zero. Conversely, at thresholds near 0%, our Recall is near 100%. However, other than these extreme drop-offs near the boundary conditions, we can see in all three plots there is a large flat plane at a high level of accuracy. This means that there is in fact a wide range of threshold values that will perform well at this task. In other words, our algorithm is not overly sensitive to these tuning parameters.



(a) Precision (b) Recall (c) F1

Figure 2.12: Accuracy sensitivity analysis at varying detection threshold values.

2.7.8 Runtime Performance Comparison

In this subsection, we evaluate the runtime performance of each technique. We compute the amount of time required, starting from the raw source code, to generate match results against the full test set. Each process is run in a single threaded non-parallel implementation. Note that this is an embarrassingly parallel problem, and significant speedup for all techniques could be achieved through parallel implementations.

Table 2.6: Run-time Comparison

System	**Runtime (s)**
RATS	1
FlawFinder	74
VUDDY	1091
ReDeBug	235
VGRAPH	1918

Table 2.6 shows the results for each technique. We can see that the pattern based approaches are among the fastest, yet as seen previously, least accurate techniques. They require only a single scan of the test set, and implement relatively simple matching algorithms. ReDeBug is the next fastest, taking a little under 5 minutes to process the full test set. ReDeBug also has minimal preprocessing required, and a simple token-based matching algorithm. VUDDY takes a little under 20 minutes. The majority of this time is generating the hash representation of the functions. After the hash is generated, the actual matching takes only a few seconds through their web service, which likely involves a simple database lookup. Finally, VGRAPH is the slowest of the approaches, taking over 30 minutes to generate the results on the test set. The graph generation process using the *Joern* tool takes roughly half of the total runtime, and the matching process the remainder. This is not unexpected, as the robust source code representation and approximate matching algorithm each come at the cost of increased preprocessing and matching times. However, we believe this to be a reasonable trade-off considering the improved ability to detect highly modified vulnerable code clones.

Table 2.7: Detected Vulnerabilities in versions of FFMpeg and OpenSSL. All versions listed are not reported by the NVD as being vulnerable to the specified CVE.

Target Product	CVE	VGRAPH Repository	VGRAPH Function	Target Function	Line Mods
ffmpeg 0.11.5	CVE-2012-2776	libav	decode_cell_data	decode_cell_data	1
ffmpeg 0.11.5	CVE-2012-2791	ffmpeg	ivi_decode_blocks	ff_ivi_decode_blocks	6
ffmpeg 0.11.5	CVE-2012-2800	ffmpeg	decode_band	decode_band	11
ffmpeg 1.0.10	CVE-2012-2776	libav	decode_cell_data	decode_cell_data	1
ffmpeg 1.0.10	CVE-2012-2800	ffmpeg	decode_band	decode_band	8
ffmpeg 2.3.6	CVE-2015-3395	ffmpeg	msrle_decode_pa4	msrle_decode_pal4	0
openssl 1.0.0	CVE-2018-0739	openssl	asn1_item_embed_d2i	ASN1_item_ex_d2i	103
openssl 1.0.0s	CVE-2016-7053	openssl	asn1_template_ex_d2i	asn1_template_ex_d2i	0
openssl 1.0.1a	CVE-2018-0739	openssl	asn1_template_ex_d2i	asn1_template_ex_d2	0
openssl 1.0.1	CVE-2018-0739	openssl	asn1_template_ex_d2i	asn1_template_ex_d2	0
openssl 1.0.2s	CVE-2018-0739	openssl	asn1_template_ex_d2i	asn1_template_ex_d2	0
openssl 1.0.1u	CVE-2018-0739	openssl	asn1_template_ex_d2i	asn1_template_ex_d2	0

2.7.9 Using VGRAPH in Practice

In order to show the ability of VGRAPH to work in a real-world setting, we utilized the VGRAPH system to identify previously unknown vulnerable code clones in several popular open source programs. Undocumented vulnerabilities which are patched in the most up-to-date version of the code, but affect older versions and are undocumented as doing so, still pose a significant security risk. Many vulnerability scanners incorporate vulnerability reporting information from sources such as the NVD. If a version of a particular software package is not listed as vulnerable to a particular CVE, vulnerability scanning software will likely not alert on those versions, despite them being vulnerable.

We utilized our VGRAPH database and hunted for vulnerabilities in several versions of FFMpeg and OpenSSL. We identified 10 vulnerabilities that were silently patched in various versions of the software which were not previously documented in the NVD. Table 2.7 lists each vulnerability we discovered along with the CVE identified, and the repository and function name of both the VGRAPH and the target graph. We also include the number of source code line differences that were identified in the vulnerable code clone compared to the original vulnerable function associated with the CVE.

Here we again see VGRAPHs ability to identify several identical code clones, as well as many type-3 code clones, some with significant modification. We see that VGRAPH was

able to identify bugs across different products, in some cases using VGRAPHs from LibAV to identify vulnerable code clones in FFmpeg, and vice versa.

2.8 Limitations and Future Work

VGRAPH was designed to be a more effective code similarity based vulnerability detection system. In particular, our goal was to build a system which was more robust to modifications in the vulnerable code clones representative of type-3 clones. However, as this is a code similarity based technique, VGRAPH still struggles to accurately detect type-4 code clones. Type-4 code clones could share little to no code with the original vulnerability, and thus would be challenging for any code similarity technique to detect. We believe type-4 code clone detection is better suited to more intensive analysis based on symbolic execution and program testing [58].

This technique is also limited by the number of VGRAPHs available in the database. We developed a technique where we mine vulnerable and patched code from GitHub in order to build our VGRAPH database, however not all vulnerabilities have a representative sample in GitHub or equivalent repository. In addition, the current techniques rely on accurate documentation on behalf of the code maintainer as to what commits are related to particular vulnerabilities, which is not always the case. In the future we plan to expand our automated VGRAPH generation routines to also mine other data sources such as the NVD and Bugtraq [1].

The focus of this work was not on scalability or performance, and there are likely many areas for improvement. This problem is embarrassingly parallel, and a simple multi-threaded implementation could generate significant speedup. We leave the full scalability analysis and performance improvements to future work.

2.9 Conclusion

In this chapter we introduced a new representation of vulnerabilities in source code based on code property triplets of the vulnerable code, the patched code, and the contextual code. We designed an accurate approximate matching algorithm which is capable of detecting modified vulnerable code clones, and differentiating them from their patched counterparts. We developed the VGRAPH detection system which mines vulnerable and patched source code from GitHub and generates our VGRAPH database which we then use to identify vulnerable code clones. We built a test dataset containing vulnerable code clones ranging from identical clones to clones with many modifications. Compared with state-of-art vulnerable clone detection techniques we are able to identify significantly more vulnerable code clones, particularly for those clones with significant levels of modification, while generating as few or fewer false positives.

Chapter 3: Detecting Lateral Movement in Enterprise Networks

According to the 2019 FireEye M-Trends report [29], the median time to detection of a network intrusion was 78 days. While this is an impressive improvement from the 418 days reported in 2011, this still means an adversary would have over 2 months inside an environment to accomplish their mission prior to detection. Additionally, nearly half of all compromises are detected via external sources, indicating that the tools currently employed by enterprise-level cyber defenders are insufficient for detecting the highly sophisticated modern-day adversaries.

After a threat actor gains a foothold in a computer network, by, for example, exploiting a software vulnerability as discussed in the previous chapter, they are often required to move to different systems within the internal computer network in order to accomplish their goals. This tactic of moving laterally through a compromised computer network is called *lateral movement*, and is challenging to detect due to the fact that adversaries typically use legitimate (yet compromised) credentials, and authentication protocols that are native to the environment and used by legitimate services.

In this chapter, we will discuss network security, lateral movement, and introduce a detection technique based on unsupervised graph AI.

3.1 Introduction to Network Security

Existing systems and techniques for detecting network intrusions rely heavily on signatures of known-bad events [102], such as file hashes of malware, or byte streams of malicious network traffic. While these techniques are able to detect relatively unskilled adversaries who use known malware and common exploitation frameworks, they provide almost no utility for detecting advanced adversaries, coined Advanced Persistent Threats (APTs), who will use zero-day exploits, novel malware, and stealthy procedures.

Similarly, the state-of-the-art behavioral analytics [104] in use today by network defenders utilize relatively rudimentary statistical features such as the number of bytes sent over a specific port, number of packets, ratio of TCP flags, etc. Not only are these types of analytics relatively noisy in terms of false positives, but they are also challenging to investigate due to their limited information and scope. For example, the fact that a particular host sent 50% more network packets in a given day could be indicative of many different events, ranging from data exfiltration, botnet command & control, to a myriad of other possibilities, most of which would not indicate a compromise, such as streaming a video.

To address these challenges, our approach is to build an abstract, behavior-based, graph data model, with key elements related to the particular behavior of interest we are trying to detect. Specifically, we model a computer network using a graph of authenticating entities, and the target behavior we detect is anomalous authentication between entities indicative of lateral movement within the network. Lateral movement is a key stage of APT campaigns when an attacker will authenticate to new resources and traverse through the network in order to gain access to systems and credentials necessary to carry out their mission [78, 88]. This is very challenging to detect as attackers will often use legitimate authentication channels with valid credentials as opposed to noisy exploitation procedures.

In order to effectively detect lateral movement, we first convert our input data, which is the in form of industry standard authentication logs, into a representation which will allow for not only learning about individual authentication events, but also the authentication behavior of the network as a whole. To that end, we construct an authentication graph, where nodes represent authenticating entities which can either be machines or users, and edges represent authentication events. Next, we utilize an unsupervised node embedding technique where latent representations are generated for each vertex in the graph. Finally, we train a link predictor algorithm on these vertex embeddings, and utilize this link predictor to identify low-probability links in new authentication events.

We apply our technique on two distinct datasets representing two contrasting computer

networks. The PicoDomain dataset is a small simulated environment we developed in-house with only a few hosts, and spanning only 3 days. The second dataset is from Los Alamos National Labs (LANL) [54] and is a real-world network capture from their internal enterprise computer network spanning 58 days with over 12,000 users and 17,000 computers. In both cases, there is labeled malicious authentication events associated with APT-style activity which were used as ground truth for evaluation purposes. We were able to detect the malicious authentication events in the real-world dataset with a true positive rate of 85% and a false positive rate of only 0.9%. In comparison, traditional heuristics, and non-graph based machine learning methods, were able to achieve at best 72% true positive rate and 4.4% false positive rate. Understanding that modern day cyber defenders are frequently receiving far too many false positives, we spent additional time building simple filters that allowed us to further reduce our false-positive rate by nearly 40% on the LANL dataset, while reducing true positives by less than 1%.

In summary, our contributions of this work are as follows:

- A graph data structure for modeling authentication behavior within enterprise-level computer networks based on information available in industry standard log files.
- An unsupervised graph-learning technique for identifying anomalous authentication events which are highly indicative of malicious lateral movement.
- Experiments on two datasets showing the strength of graph learning for this application domain.

The remaining of this chapter will be laid out as follows. Section 3.2 will provide some background into authentication protocols, the graph structure, and define the problem of lateral movement. Section 3.3 will discuss our proposed method and explain the learning algorithm. Section 3.4 will discuss our experimental evaluation and results. Section 3.5 will discuss the related work. Section 3.6 will discuss some limitations of our approach and our planned future work, and Section 3.7 will conclude.

3.2 Background & Problem Definition

In this section we will discuss some background on authentication in enterprise networks, how we build our graph structure, and define the problem of lateral movement.

3.2.1 Authentication

Modern enterprise computer networks rely on the ability to manage the permissions and privileges of users in order to maintain a safe and secure network. Users in the enterprise network will be given explicit permissions to access resources within the environment ranging from folders and network share drives, to applications and services. To make this possible, there have been many network authentication protocols developed through the years, which allow users to authenticate to resources in the network in order to verify that they have the privileges necessary to perform a certain action.

Common authentication protocols in today's enterprise computer networks include protocols such as Kerberos, NTLM, SAML, and others. Each one is designed to be a secure way to authenticate users inside an environment, and each has the ability to be abused. APT-level adversaries are well-versed in the workings of these authentication protocols, and they are often abused during an attack campaign. For example, the well-known "Pass the Hash" attack is a weakness in the NTLM implementation where the hash of a user's password, which can often be harvested from system memory, is used to authenticate to additional resources by the attacker.

Because hackers often abuse existing authentication channels, logs related to these critical protocols are valuable to the security analyst and detection algorithms. Typically these logs capture key information such as the account that is requesting to authenticate, the origin of the request, what they are attempting to authenticate to, as well as the result of that authentication request. Additionally, as authentication in the environment is network activity, we have the ability to capture this critical information from centralized network

taps, rather than requiring expensive host-based log collection.

3.2.2 Graph Structure

There were two main considerations in how we chose to build our graph data structure. First, we wanted the input data to be highly accessible to our network defenders. This means utilizing data that is likely already being collected at the enterprise scale. While some smaller enterprises may have the luxury of collecting verbose system logs from all endpoints, larger enterprises are limited to coarse feeds from centralized resources such as network sensors or domain controllers. Second, we wanted the data to provide clear and concise information related to our target detection of lateral movement. Therefore, we design our algorithm to utilize network-level authentication logs generated from Zeek sensors [120] (formerly Bro). Specifically, we utilize the Kerberos logging capability, which generates protocol specific logging on the Kerberos authentication protocol which is utilized in the majority of Microsoft Windows domains. The technique is easily adaptable, however, to other authentication logs such as host-based authentication logs, NTLM logs, Active Directory logs, or others, providing they can uniquely identify authentication events between user and system identities in the network.

For Kerberos logs, we extract the client and service principals, which are unique identifiers associated with users and services in the network, as well as the source IP address of the requesting entity, which will uniquely identify the machine from which the client is operating. The destination IP address will always be the IP of the Kerberos server itself, and thus does not add valuable information to our graph. Here is an example of content we extract from the Kerberos logs with their respective Zeek column headings:

client	**id_orig_h**	**service**
jdoe/G.LAB	10.1.1.152	host/hr-1.g.lab

This record shows that the user *jdoe* of domain *G.LAB* authenticated to service *host/hr-1.g.lab*, which is a host in the network, from IP address *10.1.1.152*.

Definition 1. *An **authentication graph (AG)** is defined as a graph $G = (V, E)$ with a node type mapping ϕ: $V \rightarrow A$ and an edge type mapping ψ: $E \rightarrow R$, where V denotes the node set and E denotes the edge set, $A = \{IP, user, service\}$ and $R = \{authentication\}$.*

A simple authentication graph generated from a small simulated computer network is shown in Figure 3.1. We can infer from this graph that there are two separate organizational units in our enterprise: the *hr* unit and the *rnd* unit, each with two user nodes (*Bob* and *Alice*, *John* and *Mary*) interacting with user workstations represented as service nodes (*hr-win7-1*, *hr-win7-2*, *rnd-win10-1*, *rnd-win10-2*), as well as some email servers and file servers (*hr-email*, *hr-fserv*, *rnd-email*, *rnd-fserv*). We can see that user *Sally* is a network administrator, as she has authentication activity to the Domain Controller service node (*DC*) in the environment, the email and file server nodes, as well as her own workstation node (*it-win10-1*). Note that for display purposes, the IP nodes have been collapsed into their representative service nodes.

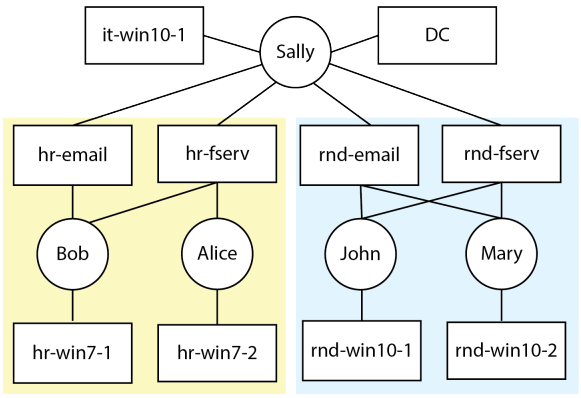


Figure 3.1: Example of an authentication graph for a small simulated network.

3.2.3 Lateral Movement

Lateral movement is a key stage of APT-level attack campaigns as seen in various attack taxonomies such as the Lockheed Martin Cyber Kill Chain [78], and the MITRE ATT@CK framework [88]. Figure 3.2 provides a simplified version of an APT-style campaign. After

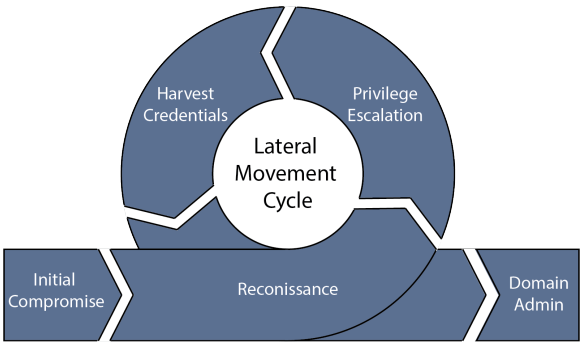


Figure 3.2: An APT-style campaign showing the cycle of lateral movement after initial compromise and prior to full domain ownership.

some initial compromise, and prior to domain ownership by the adversary, there is a cycle of lateral movement through the network. In most cases, the system that is initially compromised will be a low privileged account, typically a user workstation. This is due to the prevalence of client-side attacks (e.g., phishing), which are much more effective on typical, low-privilege users, as opposed to high-privilege IT professionals. Thus, the attacker almost always gains a foothold on a low privilege system and is thus required to move laterally through the network to achieve their goals.

Definition 2. ***Lateral movement*** *is defined as a malicious path* $< u, v >$ *conducted by an attacker in an organization's network characterized by the authentication graph, where* u, v *belong to entity set* $\{$*IP, user, service*$\}$.

For example, in Figure 3.1, if the user Alice fell victim to a phishing email and downloaded malware, the attacker would gain their initial foothold as account *Alice* on *hr-win7-2*. As *Alice* is a low-privilege account, it is unlikely that the attacker would be able to do much harm to the enterprise at large, such as installing ransomware on all the systems in the network, or exfiltrating highly sensitive business data. Therefore, the attacker would be required to move laterally to systems and accounts that have higher permissions in the environment. This can be done by exploitation of vulnerabilities, however, this is often a noisy and error prone process. More often, adversaries will harvest and abuse legitimate credentials from the set of compromised systems. In the case of our example, Alice could

harvest the domain admin *Sally*'s credentials from the file server *hr-fserv* which *Sally* had previously authenticated to, and *Alice* has privileges to access. Now, with *Sally*'s credentials, *Alice* can authenticate from *hr-win7-2* to the *Domain Controller (DC)*. This attack could be characterized by the lateral movement path: $<$*hr-win7-2*, *Sally*, *DC* $>$.

Existing techniques are not well suited to detect lateral movement within enterprise-scale environments. Most Intrusion Detection Systems (IDSs) are placed at the border of a network, and will fail to detect attacker actions after an initial foothold has been established. Even if the IDS had total visibility, an attacker using legitimate authentication channels would likely not trigger any alerts. Host-based security software relies almost exclusively on identifying signatures of known malware, and thus will prove ineffective at detecting APT-level adversaries who will move laterally through a network using novel malware or legitimate authentication mechanisms. Some environments may implement a Security Information Events Management (SIEM) System, which would allow for more complex log analytics. However, SIEMs are typically standard row or columnar data stores such as Splunk [104] which only allow for relatively basic statistical analysis of the data. Behavioral analytics implemented in SIEMs are typically simple aggregate trends of low level features such as bytes over particular ports and protocols.

3.3 Proposed Method

In this section we will discuss our proposed method for detecting lateral movement in enterprise computer networks. We will provide an overview of our machine learning pipeline, followed by detailed discussions of the node embedding process, the link predictor training, and the anomaly detection.

3.3.1 Overview

In order to detect lateral movement in enterprise computer networks, we generate authentication graphs as discussed previously and apply an unsupervised graph learning process to

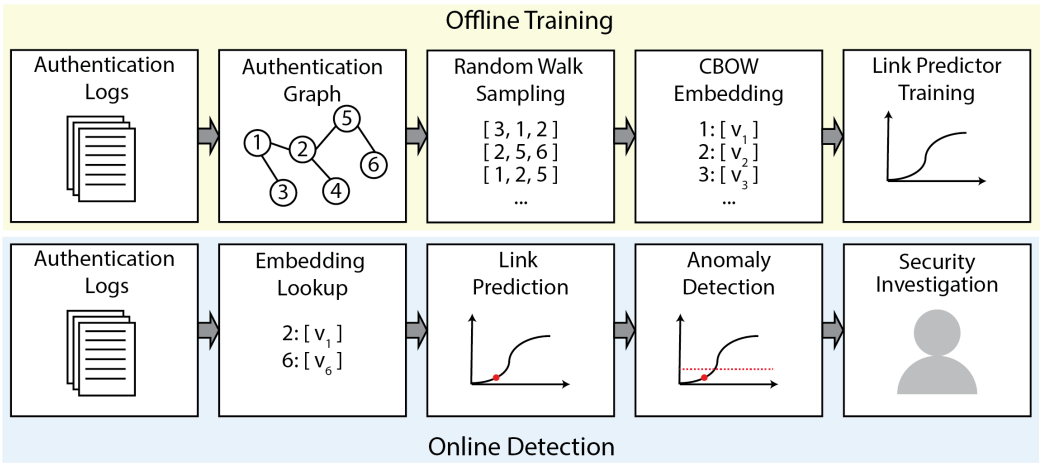


Figure 3.3: Full algorithm pipeline including offline training of node embeddings and logistic regression link predictor, as well as online detection via an embedding lookup, link prediction, and threshold-based anomaly detection.

identify low probability links. Figure 3.3 shows the algorithm pipeline. During the offline training stage (the top half of the figure), we start by generating authentication graphs, then create node embeddings via a random walk sampling and embedding process, and finally train a logistic regression link predictor using the node embeddings and ground-truth edge information from the authentication graph.

During the online detection stage (the bottom half of the figure), new authentication events are processed resulting in new edges between authenticating entities. Embeddings for these entities are generated via an embedding lookup, and link prediction is performed using the trained logistic regression link predictor. Anomaly detection is performed via a (configurable) threshold value, where links below a particular probability threshold will be forwarded to security experts for investigation.

3.3.2 Node Embedding Generation

Node embedding generation is the process by which a d-dimensional vector is learned for each node in a graph. The goal of these approaches is to generate a vector representation for each node which captures some degree of behavior within the network as a whole.

For the authentication graph, we use H to denote the set of node embeddings, $H =$

$\{h_1, h_2, ..., h_n\}$, where h_i denotes the node embedding for the ith node, and n denotes the number of nodes in the graph. In the beginning, nodes do not have embeddings, which means $h_i = \emptyset$.

In order to extract latent node representations from the graph, we utilize an unsupervised node embedding technique similar to *DeepWalk* [94], and *node2vec* [34]. We first sample our authentication graph via unbiased, fixed-length random walks. Specifically, for any node v in the graph, we will explore r random walks with a fixed-length l. For a random walk starting from node v, let v_i denote the ith node in the walk, the node sequence for this walk is generated with the following probability distribution:

$$P(v_i = x | v_{i-1} = y) = \begin{cases} \frac{1}{d_y}, & if(x,y) \in E \\ 0, & otherwise \end{cases} \tag{3.1}$$

where E denotes the edge set in the graph, and d_y is the degree of node y. This results in a set of random walk sequences $S = \{S_1, S_2, ..., S_m\}$, where S_i denotes the ith random walk sequence, and m denotes the total number of sequences.

With the sequence set of the random walks, we then tune node embeddings via a Continuous-Bag-of-Words (CBOW) model with negative sampling as proposed in [82]. In the CBOW model, we predict the target node provided context nodes from the random walk sequence. We utilize negative sampling such that we only update the vectors of a subset of nodes that were not found in the particular context window of the target node.

We use the Noise Contrastive Estimation (NCE) loss as defined in Equation 3.2:

$$L = -[logp(y = 1 | h_T, h_I) + \sum_{h_U \in N(h_I)} logp(y = 0 | h_U, h_I)] \tag{3.2}$$

where y denotes the label, h_T denotes the embedding of the target node, h_I denotes the embedding of the input node which is the average of the context nodes, h_U denotes the embedding of a noise node, and $N(\cdot)$ denotes the set of noise node embeddings for that

input. This loss function differentiates the target sample from noise samples using logistic regression [35].

Further, the probability for different labels of negative sampling is defined in Equation 3.3,

$$\begin{aligned} p(y=1|h_T,h_I) &= \sigma(h_T'^{\top} h_I) \\ p(y=0|h_T,h_I) &= \sigma(-h_T'^{\top} h_I) \end{aligned} \tag{3.3}$$

where $\sigma(\cdot)$ denotes the sigmoid function, and h'_T denotes the column vector for h_T. Therefore, the final loss value is calculated by Equation 3.4.

$$L = -[log\sigma(h_T'^{\top} h_I) + \sum_{h_U \in N(h_I)} log\sigma(-h_T'^{\top} h_I)] \tag{3.4}$$

By minimizing the loss value from Equation 3.4, we are able to tune our node embeddings such that we are more likely to predict our target node embedding h_T given the context node embeddings h_I, while simultaneously less likely to predict the negative sample node embeddings h_U given the same context h_I. We use Stochastic Gradient Descent (SGD) to minimize the loss function. In the end, we generate the output node embedding set $H' = \{h'_1, h'_2, ..., h'_n\}$, where $h_i{}'$ is the d-dimension embedding for node i.

In the context of the authentication graph, this process equates to predicting a user based on the machines and users found within at-most l-hops away. This will result in node embeddings where users who often authenticate to similar entities will be embedded in a similar region. Similarly, systems which share a user base will be found embedded in a similar region. This provides us the ability to then look at authentication events as events between two abstract vectors, as opposed to between distinct users and machines.

Figure 3.4 provides a 2-dimensional embedding space generated for the graph in Figure 3.1 using this node embedding process. We can see that the embedding of the graph corresponds nicely to the organizational units of the various users and systems. Additionally

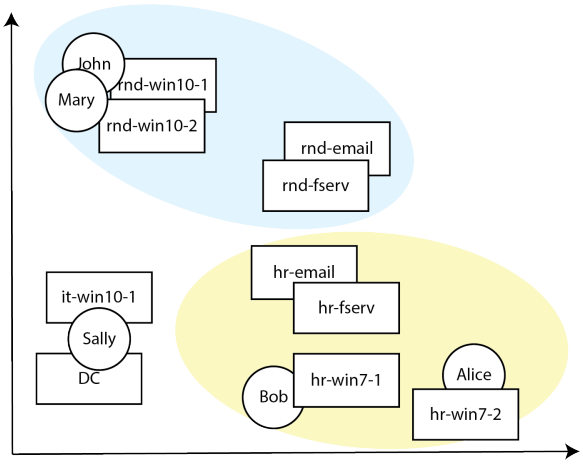


Figure 3.4: Example embedding space generated from a random-walk based node-embedding process.

we see that the servers are clearly separated from the users and their workstations. Also, the network administrator is clearly separated from both organizational units. In addition, notice that the user *Alice* does not have an edge to the *hr-email* server in the authentication graph, despite clearly being a member of the *hr* organization. Even though this is the case, we can see that *Alice* is co-located in the embedding space with other *hr* users and systems. This fact will be crucial during the link prediction process, as even though there is no explicit link between *Alice* and the *hr-email* server, we would like our link prediction algorithm to predict a high probability for the authentication event between *Alice* and *hr-email*, considering it is perfectly reasonable that *Alice* authenticates to the *hr-email* server.

3.3.3 Link Prediction

Next, we utilize a traditional logistic regression (LR) algorithm to provide us with a probability estimate that a particular authentication event occurs between two nodes a and b. Formally, our LR algorithm models:

$$P(y = 1|h') = \sigma(h') = \frac{1}{1 + e^{-w^\top h'}} \tag{3.5}$$

where y is the binary label indicating if an edge exists or not, the weight vector w contains the learned parameters, and h' is the element-wise multiplication of the node embeddings h_a and h_b defined in Equation 3.6, also known as the Hadamard product.

$$h_a \circ h_b = (h_a)_{ij} \cdot (h_b)_{ij} \tag{3.6}$$

We train the above model by generating a dataset of true and false edge embeddings from the ground truth authentication graph. The true edge set consists of all edges in the authentication graph:

$$E_T = h_a \circ h_b \; \forall (a,b) \in E \tag{3.7}$$

with each edge embedding receiving a binary label of 1. On the contrary, the false edge set consists of all edges that do not exist in the authentication graph:

$$E_F = h_a \circ h_b \; \forall (a,b) \notin E \tag{3.8}$$

with each edge embedding receiving a binary label of 0. Training on these two sets of data would cause significant over fitting as E_F contains every possible edge not in the original edge set E. Therefore, we down sample E_F via a random sampling process, and only train on the same number of false edges as found in E_T.

3.3.4 Anomaly Detection

Anomaly detection is achieved by applying our trained LR link predictor to new authentication events. First, authentication events are parsed into a set of edges between authenticating entities. Next, we perform an embedding lookup for the node embeddings generated during

the training stage. The anomaly detection function A can be expressed as:

$$A(h_a, h_b) = \begin{cases} 1, & if\ f(h_a \circ h_b) < \delta \\ 0, & otherwise \end{cases} \tag{3.9}$$

where h_a and h_b are the embeddings for nodes a and b, and the function $f(\cdot)$ is the logistic regression link predictor trained on the true and false edges generated from our training graph. The parameter δ is the threshold for generating an alert. In this paper, we use a threshold of $\delta = 0.1$, or 10%, which we will show shortly yields good performance.

3.4 Evaluation

In this section we will evaluate our technique for detecting malicious authentication in enterprise networks. First we will discuss the datasets we used for evaluation, followed by a detailed description of the various methods we evaluated, and an analysis of our results. In an effort to further reduce false positives, we make some observations about the data and our results, and update our algorithm accordingly.

3.4.1 Datasets

We apply our malicious authentication detection to two datasets generated from contrasting computer networks. Table 3.1 provides details on each dataset. We discuss both datasets in detail below.

Table 3.1: Dataset Details

	PicoDomain	**LANL**
Duration in Days	3	58
Days with Attacks	2	18
Total Records	4686	1.05 B
Total Attack Records	129	749
User and Machine Accounts	86	99968
Computers	6	17666

PicoDomain is a dataset we generated in-house for cyber security research. It is designed to be a highly scaled-down environment which contains only the most critical elements commonly found in enterprise-level domains. Specifically, the PicoDomain consists of a small Windows-based environment with five workstations, a domain controller, a gateway firewall and router, and a small-scale internet that houses several websites as well as the adversarial infrastructure. A Zeek network sensor was installed inside the environment and placed such that it had visibility of traffic entering and leaving the network from the simulated Internet (north/south), as well as traffic between local systems in the simulated enterprise network (east/west). A total of three days of network traffic was captured. During this three day period, there was benign activity performed in a typical 9-5 workday pattern, such as browsing the web, checking e-mail, etc. Additionally, on days 2 and 3, we ran an APT-style attack campaign which included all stages of the kill chain. The attack campaign started with a malicious file downloaded from an e-mail attachment. This gave the attacker the initial foothold in the network. The attacker then proceeded to perform various malicious actions typically associated with APT-level campaigns. This included exploiting system vulnerabilities for privilege escalation, registry modifications to maintain persistence, credential harvesting via the tool Mimikatz, domain enumeration, and lateral movement to new systems via the legitimate Windows Management Instrumentation (WMI) service. At the end of the campaign, the attacker was able to compromise a domain admin account, resulting in full network ownership by the attacker.

Comprehensive Cyber Security Events is a dataset released by Los Alamos National Labs (LANL) and consists of 58 consecutive days of anonymized network and host data [54]. There are over 1 billion events containing authentication activity for over 12,000 users and 17,000 computers in the network. An APT-style attack was performed during the data capture, and relevant authentication log entries were labeled as being malicious or benign. No further details were provided in the dataset as to what types of attacks were performed during the exercise. This is a limiting factor of this dataset, and, in fact, led to the generation

of the previously mentioned PicoDomain dataset.

3.4.2 Methods Evaluated

We evaluate two variants of our proposed graph learning methods, as well as four different baseline techniques, which include two non-graph-based machine learning algorithms, as well as two traditional rule-based heuristics. We will discuss each below.

Graph Learning with Local View (GL-LV). This is our graph learning technique configured in such a way as to have a more localized view in our graph. This means our embeddings and link predictor will be optimized for nodes within a close proximity. To achieve this, we generate 20 random walks of length 10 for every node, and generate a 128-dimension embedding for each node based on a context window size of 2. This means each node will only consider a neighborhood of 2-hop neighbors in the embedding process. Our anomaly detection threshold is set at $\delta = 0.1$.

Graph Learning with Global View (GL-GV). This is our second graph learning variant which is very similar to the first, however this time configured to have a more global view of the graph. This means our embeddings and link predictor will be optimized for nodes that are further apart in our graph. To that end we used the same configuration as previously, however now setting the window size to 5. This means nodes will consider at most 5-hop neighbors during the embedding and link prediction process, which will give the algorithm a much broader view of the graph.

Local Outlier Factor (LOF) [18]. For a non-graph-based machine learning comparison, we implement the LOF anomaly detection algorithm. The LOF is a density-based anomaly detection approach, where relative local densities are compared between each sample, and those which are very different from their neighbors are considered anomalous. In order to generate features for this algorithm, we 1-hot encode the authentication events into an authentication vector containing a dimension for all authenticating entities. For each event, the dimensions corresponding to the various authenticating entities for that particular record

will be set to 1, and all other dimensions will be 0. We then apply the LOF algorithm to these vectors to identify anomalies.

Isolation Forest (IF) [70]. This is a second non-graph-based machine learning comparison technique. The Isolation Forest algorithm identifies samples that can be easily isolated from the dataset by simple decision trees as being anomalous. This is applied to the same authentication vectors as in the previous LOF method.

Unknown Authentication (UA). This is a more traditional rule-based heuristic which simply identifies all first-time authentication events as anomalous. During the training period, a list of known authentications is generated for each authenticating entity in the network. During the testing phase, any authentication event which was not seen during the training phase is considered as anomalous. After an anomalous result is generated the first time, the authentication event is added to the set of known authentications for the particular entity. This way we do not generate repeated anomalies for the same event.

Failed Login (FL). This is a second traditional rule-based heuristic which considers all failed login events as anomalous. As this technique does not requiring any training data, we only evaluate it on the test portion of the datasets.

3.4.3 Detection Analysis

Next we apply the six different algorithms discussed previously and evaluate their ability to detect malicious authentication in our two datasets. For all techniques, we report the number of true positives (TP), false positives (FP), as well as the true positive rate (TPR), and false positive rate (FPR).

PicoDomain. First we apply all techniques to the simulated PicoDomain dataset. We split the dataset into training and testing, with the training data consisting of authentication activity before the APT attack began, and the testing data containing all other activity. As this is a small dataset focused on malicious activity, the majority of the time period contains malicious events. As a result, there was only roughly 20% clean training data available.

Thus our final train/test split on this data was about 20%/80%. For all 6 detection techniques, we only generate accuracy metrics on the testing dataset.

Table 3.2 shows the results for all six techniques. Not unsurprisingly, the **UA** detector performed very well, with 100% TPR, and only 1.5% FPR. This means all of the lateral movement associated with the APT campaign involved systems which did not have authentication activity during the training period, a characteristic that is likely only to hold in relatively small and simulated environments. We can also see that the failed login (**FL**) rule generated very few results, and only managed to detect a single event associated with the malicious activity. This is due to the fact that the APT campaign did not involve any brute-force password guessing attempts. The single failed login is likely due to user error during the attack.

Table 3.2: Anomaly Detection Results on PicoDomain Dataset

Algorithm	**TP**	**FP**	**TPR (%)**	**FPR (%)**
UA	**129**	11	**100**	1.5
FL	1	15	0.8	2.0
LOF	41	19	32	2.5
IF	34	62	26	8.3
GL-LV	102	**0**	80	**0.0**
GL-GV	102	**0**	80	**0.0**

Both ML techniques (**LOF** and **IF**) struggled to detect malicious events, with TPRs well below 50%, and FPRs as high as 8.3%. This indicates that the pure authentication activity between entities, without the additional information present in the graph topology, is not sufficient for detecting lateral movement.

Our graph learning techniques, **GL-LV** and **GL-GV**, performed much better than the comparison ML techniques, achieving 80% TPR. This shows the strength of the graph topology for the detection of lateral movement. Additionally, the graph-learning approaches were able to reduce the FPR to 0% compared with the 1.5% of the **UA** detector. A low false positive rate is critical for anomaly detection techniques, as will be made clear by the next experiment on the LANL dataset. Interestingly, we see that the global view and local view

had no effect on the performance. This again is likely due to the extremely small scale of this dataset. The average shortest path between any two nodes in the PicoDomain graph is slightly over 2 hops. This means the additional visibility that the **GL_GV** detector provides will not contribute significantly more information on the graph structure.

LANL. Here we apply the same 6 detectors to the LANL Comprehensive Cyber Security Events dataset. In a similar manner, we split the data into training and testing sets. The training set consists of 40 days on which no malicious activity is reported, and the testing set of 18 days with malicious activity. This is equivalent to roughly 70% training data, and 30% testing data. Due to the large scale of this dataset, it was necessary that we perform an additional down sampling for the two ML techniques **LOF** and **IF**, which was accomplished by removing timestamps from the training and testing dataset, and removing duplicate events. The TPR and FPR for these two techniques have been adjusted to account for this.

Table 3.3 shows the results for the six anomaly detectors. The impact of scale is readily evident in these results, with a significant number of false positives for all detectors, despite reasonably small false-positive rates.

We can see that the **UA** detector performs again reasonably well, with a significant 72% of the malicious authentication events detected. However, with this real-world dataset, we can see how noisy this detector is, with a FPR of 4.4% resulting in over 500,000 false positives. The **FL** detector again fails to perform, indicating that for APT style campaigns, simple failed login attempts are not suitable detectors. Similarly, both ML approaches generated

Table 3.3: Anomaly Detection Results on LANL Dataset

Algorithm	**TP**	**FP**	**TPR (%)**	**FPR (%)**
UA	542	530082	72	4.4
FL	31	116600	4	1.0
LOF	87	169460	12	9.6
IF	65	299737	9	16.9
GL-LV	503	146285	67	1.2
GL-GV	**635**	**107960**	**85**	**0.9**

many false positives, and few true positives, again showing that simple authentication events without the added information in the authentication graph are insufficient for malicious authentication detection.

The two graph learning techniques were able to provide the best TPR at the least FPR. The **GL-LV** detector, although returning less true positives than the simple UA detector, was still able to detect 67% of the malicious activity, at only 1.2% FPR compared to 4.4% by the **UA** detector. The best performing predictor on this dataset is the **GL_GV** detector, which was able to detect the most malicious authentication events with a TPR of 85%, while maintaining the lowest FPR of 0.9%. For this dataset, the increased context window of the **GL-GV** over the **GL-LV** contributed significantly to the added performance. The average shortest path between any two nodes in the LANL graph is roughly 4 hops. This explains why, in this case, the broader view of the **GL_GV** detector was able to capture more information from the graph structure in the node embeddings, resulting in a better performing link predictor.

It is important to note here that all of the previous experiments were performed on commodity server hardware. Specifically, we utilized a server with two Intel Xeon CPU E5-2683 CPUs, and 512 GB of ram. This provided enough memory and compute power to run any of the detectors discussed on the full 58-day LANL dataset in under 6 hours. We believe that the techniques used here would be supported by the infrastructure already available to our network defenders.

3.4.4 Reducing False Positives

As we can see from the previous experiment, and specifically Table 3.3, the effect of false positives on the datasets of the scale found in the real-world can be very detrimental. Even for the best performing detector, the **GL_GV** detector, a false positive rate of 0.9% resulted in over 100,000 individual false positive results in the test data. As these results will ultimately be used by cyber analysts to investigate the threats, it is important that we do

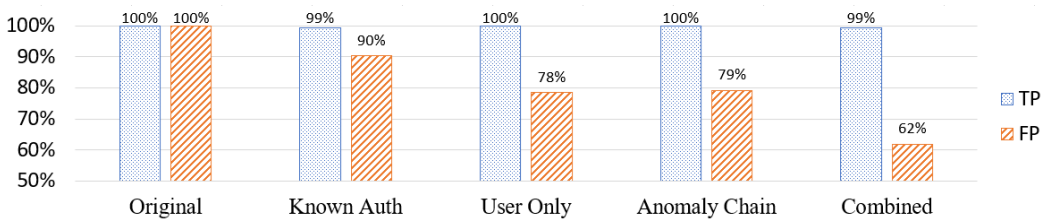


Figure 3.5: Impact of various approaches in reducing the number of false positives returned on the LANL dataset.

our best to keep the false positives to a minimum. In this section, we present some of our observations of the data and results, and design several filters to further reduce the false positive rate by nearly 40%, while reducing true positives by less than 1%.

Observation 1: The malicious authentication events are predominantly first authentication events.

This observation was made based on the fact that the simple unknown authentication (UA) detector performed very well at identifying the malicious events. However, its false positive rate was far too high to use on its own. Based on this observation, we use the inverse of this detector as a false positive filter. More precisely, all anomalies generated by the graph learning approach are passed through a filter based on the known authentication events. We discard any of the anomalous authentication events that were previously seen during the training period. This filter corresponds to the "Known Auth" filter in Figure 3.5. We can see that we achieved about a 10% reduction in false positives, while reducing true positives by less than 1%.

Observation 2: The malicious authentication events are predominantly based on user interactions.

Our authentication graph includes interactions between users and computers, but also interactions between purely computers. Some of the interactions are possibly associated with the red team exercise, however, the labeling scheme utilized by LANL only labeled authentication events involving user accounts as being malicious. Without further details

on exactly what the red team activity entailed, it is impossible to label other interactions as malicious or benign that could have been associated with the red team exercise. Based on this, we modify our anomaly detection algorithm, and again add a new filter where the results that are generated and do not involve at least one user account are discarded. This filter corresponds to the "User Only" filter in Figure 3.5. We can see this had a significant impact on the results, reducing false positives by over 20% from the original, while not reducing the true positives at all.

Observation 3: The malicious authentication events are predominantly related to specific user accounts and systems.

This observation makes sense from a practical standpoint. When an adversary gains access to a network, it is unlikely that they have multiple initial footholds. Typically a single foothold would be established, and then access throughout the network would expand from there. This means that all of the malicious edges in our authentication graph should be close together, or even form a connected component in the graph. Based on this observation, we build a third filter, where all of the anomalous results are chained together based on their shared nodes and edges. Any anomalous results which do not form a chain with at least one other anomalous event is discarded. This filter corresponds to the "Anomaly Chain" filter in Figure 3.5. This resulted again in about a 20% reduction in false positives from the original, and no reduction in true positives.

To summarize, the last bars labeled as "Combined" in Figure 3.5 represent the results when combining all of the previous filters together. We can see this resulted in the best performance, and was able to reduce the number of FPs on the LANL dataset by nearly 40%, while losing $< 1\%$ of the true positives.

3.5 Related Work

This section studies the related works in terms of anomaly detection and node embedding methods.

Anomaly detection for APT identification has been extensively studied. However, the majority of works are based on expensive host-based log analysis, with the goal of anomalous process activity, indicative of malware or exploitation [98] [31] [101] [77]. Some go so far as mining information from user-driven commands for anomaly detection [64]. While host logs may be available in some environments, it would be a significant burden for most large enterprises to capture and store verbose host-based logs such as system call traces.

At the network level, there are techniques for detecting web-based attacks [61], as well as botnet activity [13] utilizing anomaly detection algorithms. A highly related technique [27] combines host information with network information to detect lateral movement. However, they require process-level information from hosts, making this technique a poor fit at the enterprise scale. As lateral movement detection is such a hard problem, some approaches instead focus on detecting the degree to which environments are vulnerable to lateral movement attacks [52].

There are also approaches that look for deviations from known, specification-driven, rules of how an environment should behave, such as Holmes [85] and Poirot [83]. While these work reasonably well and are able to reduce false positives by explicitly defining what behavior is deemed malicious, they are still based on knowledge derived from a human, and thus risk circumvention by new and novel attack paths. In addition, these techniques require constant maintenance and upkeep to develop new specifications for the constantly evolving attack surface.

Node embedding methods aiming at learning representative embeddings for each node in a graph have been successfully applied to various downstream machine learning tasks,

such as node classification [94], link prediction [34], and node recommendation [119]. Existing methods usually take two steps to generate node embeddings. First, they sample meaningful paths to represent structural information in the graph. Second, they apply various data mining techniques from domains such as natural language processing (NLP), utilizing technologies such as word2vec [82] for learning meaningful vector embeddings.

The major difference between existing methods lie in the first step, i.e., how to mine better paths to capture the most important graph information. In this context, the early work DeepWalk [94] applies random walks to build paths for each node. In order to give more importance to close-by neighbors, Line [106] instead applies a breadth-first search strategy, building two types of paths: one-hop neighbors and two-hop neighbors. Further, the authors of node2vec [34] observe that the node embeddings should be decided by two kinds of similarities, homophily and structural equivalence. The homophily strategy would embed the nodes closely that are highly interconnected and in similar cluster or community, while the structural equivalence embeds the nodes closely that share similar structural roles in the graph. Based on these strategies, node2vec implements a biased random walk embedding process which is able to model both similarity measures.

There are additionally many other graph neural network architectures recently proposed, such as the convolution-based GCN [59], attention-based GAT [108], and many variants based on both [37]. However, they are mostly designed for semi-supervised or supervised tasks, and are not as suitable for unsupervised learning as the random-walk based approaches mentioned previously.

3.6 Limitations & Future Work

Although our results are promising, there are several limiting factors of our approach. The first limitation is the problem of explainability, which is not specific to our technique, but rather a limitation of machine learning techniques in general. When our graph learning algorithms label an event as an anomaly, it is relatively challenging to determine why it

has done so. There is current and active research on explaining machine learning and AI algorithms [23], and many even specific to explaining the results of graph learning algorithms in particular [117]. We may be able to use some of these techniques in the future which would allow us to identify what nodes were most important when generating both the embedding, and ultimately the link prediction scores.

Our detection algorithm is based on the assumption that we will have historic data for each entity we plan to perform link prediction on in the future. If we have never seen an entity authenticate before, then we will not have an embedding generated for that entity, and thus we will be unable to perform the link prediction. There are many ways to handle this problem, such as assigning new entities a generic "new node" embedding, or assigning the new node embedding to the average embedding of its neighbors (provided that they have embeddings themselves), however we have not explored the impact of these various approaches. We believe that, at least in the case of enterprise network authentication, it is a fair assumption to believe that for the vast majority of user accounts in the network, there should be some history of their behavior provided a sufficiently long historic window.

In this work we focused specifically on log data pertaining to authentication events. However, there is a myriad of additional data that we could add to our graph and ultimately to our graph learning algorithms. In the future we plan to add finer grained detail of actions performed by users, such as DNS requests and file-share accesses. This will allow us to also expand our detection algorithm to identify other stages of the kill chain beyond lateral movement, such as command and control traffic, which would likely cause anomalous DNS requests.

3.7 Conclusion

In this chapter we discussed the challenging problem of detecting lateral movement of APT-level adversaries within enterprise computer networks. We explained why existing signature-based intrusion detection techniques are insufficient, and existing behavioral

analytics are too fine grained. We introduced our technique of abstracting a computer network to a graph of authenticating entities, and performing unsupervised graph learning to generate node behavior embeddings. We discussed how we use these embeddings to perform link prediction, and ultimately anomaly detection for malicious authentication events. We applied our techniques to both simulated and real-world datasets and were able to detect anomalous authentication links with both increased true positive rates, and decreased false positive rates, over rule-based heuristics and non-graph ML anomaly detectors. We analyzed the results of our algorithm, and developed several simple filters to further reduce the false positive rate of our technique.

Chapter 4: NetHawk: Hunting for Advanced Persistent Threats via Structural and Temporal Graph Anomalies

Advanced Persistent Threats, or APTs, perform highly complex, multi-stage attack campaigns which involve zero-day exploits, lateral movement, as well as a myriad of other tactics, techniques, and procedures (TTPs) in order to accomplish their mission. Existing approaches relying on signature detection are unable to detect new and stealthy adversaries, and behavioral analytics are frequently too granular, or too strict to capture an accurate and complete account of the APT attack campaign. This leads to an increased mean-time-to-detection, mean-time-to-remediation, and accordingly a higher likelihood of attack success.

In this chapter we will discuss our system called NETHAWKwhich applies threat hunting techniques in a real time and continuous manner to detect and correlate the many anomalous and malicious actions performed by advanced threat actors inside a network.

4.1 Introduction to NetHawk

The existing cyber security tools utilized by enterprise defenders suffer from two major challenges. The first is that they are overwhelming based on detecting signatures of known bad events, such as malicious file hashes, domain names, IP addresses, etc. While these methods can provide a high fidelity indicator of compromise, they are inherently flawed as they will never be able to detect a sophisticated adversary who will not reuse known-bad attacks. For example, the recent high profile *SolarWinds* hack was not only based on new malware, but in fact malware embedded in previously trusted software [30]. Many advanced attack campaigns will perform so-called *living-off-the-land* attacks, utilizing known and trusted software native to the environment to accomplish their mission, making them very challenging to detect [11].

The second challenge comes from the fact that security tools often generate a high volume of weak indicators of comprise, and require a human analyst to then "connect-the-dots" to understand the full impact of a security event. This challenge is exacerbated by the fact that many security alerts are in fact false positives [10, 103, 95, 41] leading to so called "alert fatigue" of the cyber analysts. Differentiating true positives from false positives, while simultaneously stitching together the true positive alerts into a security incident requires a significant level of expertise, as well as time and effort, and often results in attacks succeeding due to the full scope of the attack not being properly understood in time by the defenders. For example, during the Target hack in 2013, where credit card data was stolen from some 1,700 stores, there were multiple security alerts generated by the malicious activity. However, the security team did not successfully assemble the various alerts into a representation that illustrated the full scope of the security event, resulting in millions of dollars in damages [76].

In this work, we introduce our system, NETHAWK, which can accurately and scalably model user and system activity within enterprise-grade networks. At the heart of our system is our *Cyber Activity Graph* data structure which captures and learns network-wide dynamics between users and systems. In addition we have devised a novel anomaly detection algorithm based on aggregating temporal and structural anomalies across various cyber security relevant activity types such as network activity, file activity, authentication activity, etc. Unlike prior graph-based APT detection systems [40, 39, 86, 84] NETHAWK operates at the network level as opposed to the host level, and therefore it can stitch together related events that span hosts and users in the enterprise. NETHAWK is also entirely based on anomalies, and does not require expensive creation or maintenance of rules, patterns, or signatures of attacks. In addition, the NETHAWK architecture is easily extensible to support many different activity types with minimal modification.

We evaluate our system on the OpTC dataset [92] generated during the DARPA Transparent Computing program, and the Comprehensive Multi-Source Cyber Security Events

dataset [55] from the Los Alamos National Labs (LANL) enterprise network. The former dataset is a simulated enterprise-grade network including roughly 600 hosts, 600 users, benign activity, as well as three days of malicious red team attack campaigns. The latter dataset is a real-world data capture from the internal LANL network, also including labels from red team activity. Our system was able to learn the benign activity, and identify the malicious attack campaigns with 90% precision and 98% recall in OpTC, and 72% precision and 93% recall in LANL. Notably, NETHAWK accomplishes all of this while maintaining a small memory footprint of under 300MB, and a processing time of less than one minute for every hour of data for both the OpTC and LANL datasets.

It is worthy to note that we have deployed NETHAWK in a real-world operational network. During a month-long evaluation, NETHAWK generated four incident reports, one of which is directly related to a red team attack campaign. An independent evaluation shows that NETHAWK achieved around 65% precision, 36-42% recall, and 46-52% F1 score.

In summary, the contributions of this paper are as follows:

- A graph data model capturing cyber relevant activity that is scalable to enterprise-wide analysis.
- An anomaly detection algorithm that learns per-entity baselines, combines and correlates temporal and structural anomalous activity, and produces accurate and concise security incidents.
- A comprehensive evaluation of our system on two datasets representative of real-world environments containing malicious activity, as well as results from a deployment in a live enterprise network.

The remaining of this chapter will be organized as follows: Section 4.2 will provide background on enterprise network security and graph analysis. Section 4.3 will introduce our system and discuss our data model and anomaly detection algorithm. Section 4.4 will

discuss our experiments and evaluations. Section 4.5 will highlight the related work in this area. Section 4.6 will conclude.

4.2 Background

4.2.1 Cybersecurity Monitoring

The threat from ransomware, APTs, and insiders to large organizations is higher than ever, as demonstrated by the frequent high-profile hacks that occur seemingly daily. In order to manage this risk, enterprise networks employ around-the-clock security operations centers, or SOCs, which maintain 24/7/365 "eyes-on-glass" support[44].

Unfortunately, as indicated by the frequent breaches, adversaries are succeeding in circumventing these security controls. The reason for this is that the vast majority of security controls are focused specifically on identifying patterns or signatures of known-bad activity. For example, identifying a file that is known to be malware, or an IP address that is known to be malicious, are traditional ways to detect malicious activity. However, adversaries are refining their techniques, and are less often re-using known-malicious code and infrastructure. Thus, in order to detect malicious activity, the defensive strategy needs to shift focus to detecting malicious behaviors, rather than signatures. For example, identifying an unusual running program that is exhibiting behavior similar to ransomware. This is exceedingly challenging in large and complex computer environments due to the fact that often these abstract behaviors can be the result of either benign or malicious activity. For example, ransomware encryption programs have behavior that can look very similar to the *zip* program [56]. This can lead to many false-positive alerts that then require human analysts to sift through in order to extract and correlate the meaningful activity needed identify real security incidents.

4.2.2 Threat Hunting

In addition to the constant security monitoring that happens inside the SOC, many organizations also employ an advanced defensive technique called Threat Hunting. In this approach, threat hunting teams analyze a network with the assumption that an adversary has already gained access to the environment, and it is their task to find and eradicate them. This approach is almost exclusively a human driven effort which relies heavily on the intuition and expertise of the threat hunters. A common approach threat hunters utilize is long-tail analysis [65], or, in other words, analyzing the long tail of infrequent events in a network. The intuition here is that attacker related activity is likely to cover at least some events that are unusual or infrequent for any given computing environment. For example, a user authenticating to a system for the first time, logging in at an unusual hour, or interacting with a large amount of files. Using these unusual and infrequent events as a starting point, threat hunters then cross correlate information from other sources in order to uncover advanced adversaries that have infiltrated the network.

4.2.3 Graphs for Cybersecurity

The majority of off-the-shelf security tools and analytics focus on traditional row or columnar data representation and analysis. This means events are typically analyzed through SQL-style query interfaces. Representing and analyzing the data in this way imposes limitations on the types of analysis that can be performed [63]. Some prior works design purpose-built query systems entirely for cybersecurity analysis [33, 32]. In this work, we represent and process cybersecurity data in the graph domain. A graph $G = (V, E)$ is a collection of vertices V and edges E that represent data as a set of relationships. As presented in some other works [16, 89, 50], the graph domain provides some significant algorithmic advantages over traditional database representations for a variety of cybersecurity relevant tasks.

In this work we combine aspects of security monitoring, threat hunting, and graph

analysis, to build a system that hunts for threats in real time, and aggregates and correlates events without the need for brittle signatures, while also not overwhelming analysts with a deluge of alerts.

4.3 NETHAWK System

In this section we will discuss the details of our detection and monitoring system which we call NETHAWK. We will start with a high level overview of the components of the system, followed by a deep dive of each element.

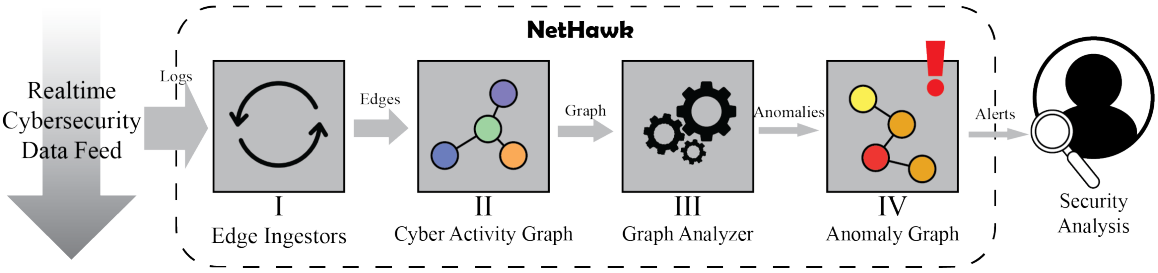


Figure 4.1: NETHAWK Detection and Monitoring System Architecture

4.3.1 System Overview

Figure 4.1 shows the workflow illustrating how the system operates. Each component is summarized below.

I. Edge Ingestors are the first stage of our system which parse cybersecurity log data and convert the information into a set of timeseries edges in our Cyber Activity Graph. This will require custom parsers for each log type in use. The edge ingestors will either pull from an existing data store, such as a Security Information and Events Management System (SIEM) (e.g., Splunk [104], ELK [26]), or from a streaming data feed (e.g., Apache Kafka [9]), convert the raw logs into a set of node and/or edge updates, and apply those updates to the Cyber Activity Graph.

II. Cyber Activity Graph is the primary data structure of our system that contains the

real time graph information pertaining to how users and systems are interacting across an enterprise network. The graph contains information that is both structural in nature (e.g., who is talking to who?), as well as behavioral (e.g., what are they saying?). The graph edges as well as the features are further attributed with timestamps of the first observation of a particular behavior on each edge. This fine-grained timeseries information per entity in the graph is used by the downstream anomaly detection algorithm.

III. Graph Analyzer will process a snapshot of the Cyber Activity Graph and assign an anomaly score to all edges in the graph using our anomaly detection algorithm.

IV. Anomaly Graph receives anomalous edges from the Graph Analyzer, and will commit a subset of those edges to the Anomaly Graph data structure for inspection by a security analyst when the Anomaly Score exceeds a certain threshold.

4.3.2 Edge Ingestors

The Cyber Activity Graph is only as expressive as the data on which it is generated. As we are focused on granular, user-driven behaviors and activity, it is necessary that we have an equally granular data source. As a result, we focus on analyzing host-based telemetry. Fortunately, this type of information is already generated by many enterprise solutions such as *Endpoint Detection and Response* (EDR) solutions [79], *Antivirus* solutions (AV) [109], *Sysmon* [80], or even native solutions provided by operating systems, such as *Event Logs* in Windows [81], or *auditd* in Linux [69]. In most of these cases, the information such as which user is logged in, and what actions they are performing on a particular system, will be logged and optionally forwarded to a centralized system for analysis, e.g., a Security Information and Events Management system (SIEM) such as Splunk [104], or general purpose data lake such as ELK [26].

The responsibility of the Edge Ingestors is to pull log data from the SIEM or similar centralized log source, transform the content into a time series edge list, and update the Cyber Activity Graph. The type of information that is parsed from the log data is discussed

in the next section where we describe the edges in the Cyber Activity Graph.

4.3.3 Cyber Activity Graph

The Cyber Activity Graph is a graph structure $G = (V, E)$ where V is a set of vertices of type $\{system, user\}$ and E is a set of edges of type $\{file, process, network\}$. All nodes and edges are attributed with a timestamp corresponding to the first time NETHAWK observed the node or edge. Edges are further attributed with behavioral features which themselves are also each associated with a timestamp of the first observation. Each node and edge type is discussed below.

User Nodes are the entities which correspond to an account, persona, or credential. These nodes could correspond to user accounts tied to human identities, or to service credentials tied to software services within an environment. These nodes could be generated from various different data sources, such as host logs, authentication logs, Active Directory logs, etc. These nodes are deigned to capture *identities* within an enterprise network.

System Nodes are the entities which correspond to systems or services and are attributed with information such as IP address and/or hostname. These nodes could be generated from various different data sources, such as application logs, network logs, authentication logs, Active Directory, etc. These nodes are designed to capture *services* within an enterprise network.

Network Edges characterize how users interact with network resources. These edges contain edge attributes which correspond to what ports were utilized in the network communication. Valid attributes are the lower 1024 non-ephemeral ports, a single attribute indicating communication over an ephemeral port, as well as a direction specifier to indicate if the user has originated flow activity from the system, or is involved in flow activity to a system. Additionally each port is tagged with the time it was first observed on each edge.

Process Edges characterize what processes are used by users on particular systems, and are attributed with the name of the process and the time of the first observation.

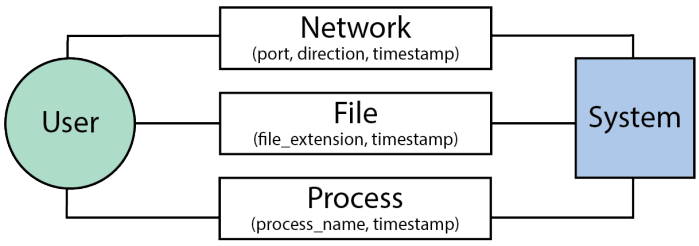


Figure 4.2: Cyber Activity Graph Schema

File Edges characterize how users interact with files on a system. Specifically, we analyze what types of file extensions are in use by the users, as these have significant behavioral implications. Each file extension observed for a particular user is again attributed with the timestamp of the first observation.

Figure 4.2 provides a diagram representation of the information captured in the Cyber Activity Graph. Note that all graph elements are tagged with timestamps. This is important as our anomaly detection algorithms which we will discuss in the coming sections are heavily focused on identifying temporal anomalies, and correlating them via the graph structure.

4.3.4 Graph Analyzer

The purpose of the Graph Analyzer component is to periodically analyze the Cyber Activity Graph and assign anomaly scores to edges of interest. The analysis consists of temporal and structural analysis of the aforementioned Cyber Activity Graph, in addition to a training phase.

Temporal Analysis. The Graph Analyzer is powered by a custom anomaly detection algorithm which was influenced by the threat hunting technique of long-tail analysis. The idea of long-tail analysis is that the most infrequent events are a good place to start when looking for potentially malicious activity. As NETHAWK is a system designed for monitoring real-time enterprise network data feeds, we relax long-tail analysis slightly - instead of just focusing on infrequent events, we focus on new, previously unobserved events.

Equation 4.1 shows the most basic form of the edge feature anomaly calculation in our system. This equation is an exponential decay function which returns values between $(0,1]$ based on the age of an edge feature f.

$$A(f) = e^{-\alpha * age(f)} \tag{4.1}$$

The $age()$ function is simply the number of time units that have elapsed between the current time of analysis, and the first observation of edge feature f. The α parameter allows us to adjust the rate of decay of the anomaly score. As the value of α increases, the anomaly score of a new observed event will decay more quickly. A value of α that is too high will cause the system to "forget" about activity too quickly, meaning it will not properly stitch together related malicious events. A value of α that is too small will cause the system to "remember" activity for too long, resulting in the increased likelihood of unrelated but anomalous activity being incorrectly associated together, ultimately resulting in false positives.

A more intuitive way to tune this function is to introduce the idea of anomaly half-life $t_{1/2}$. This is the amount of time that must pass for the anomaly score of a particular edge feature to decrease by half. We can adjust the equation as shown in Equation 4.2 and introduce the parameter $t_{1/2}$ which represents the number of time units that must pass before the anomaly score will be halved. In our experiments, we have found a half life of 6 hours to provide good performance.

$$\begin{aligned} A(f) &= e^{\frac{ln(0.5)}{t_{1/2}} * age(f)} \\ &= 0.5^{\frac{age(f)}{t_{1/2}}} \end{aligned} \tag{4.2}$$

Structural Analysis. A common challenge with temporal based threat hunting analysis is that there are many new or infrequent events that occur in large-scale and diverse computing

environments that are in fact not malicious. Because of this it is not sufficient to only look at new or infrequent events as this would cause far too many false positives. To handle this challenge we utilize the graph structure in order to reduce the anomaly score of edges that are unlikely to be malicious based on the neighborhoods of the nodes involved.

The *Neighbor Similarity Scaling* technique aims to *reduce* the anomaly score of an edge feature based on the similarity of the anomalous edge feature to other edges that have been previously observed and are no longer considered anomalous. In other words, if new activity is observed between node a and b, we reduce the anomaly score based on if that same activity was observed between nodes (a,b') where $b \approx b'$. There are many ways to compute node similarity, however in this work we use a simple function based on nodes with a similar degree that are in the 1-hop neighborhoods of the nodes involved in the original edge. Equation 4.3 shows the similarity function. The β parameter controls how strict this equation is, and the $D()$ function returns the degree of the node.

$$sim(a,a') = \begin{cases} true & |D(a')-D(a)|<\beta * \frac{D(a')+D(a)}{2} \\ false & otherwise \end{cases} \tag{4.3}$$

Pseudocode for scaling a particular anomaly score s, based on a feature f between two nodes (a,b) in a graph G can be seen in Algorithm 3. After identifying a suitable nodes based on our similarity function in the 1-hop neighborhood, the anomaly score of the same feature (e.g. port, process name, etc) is used as a scaling factor on the original anomaly score s. As the values returned by the anomaly function $A()$ are necessarily $<= 1$, this will always result in either no change, or a reduction of the anomaly score s.

It is important to note that we do not want to blindly apply the neighbor similarity scaling technique for all edges in the graph. For edges involving low-degree nodes (e.g., users and their workstations), new activity, despite its similarity to previously observed activity on other edges, is inherently interesting. For example, a user logging into their co-workers workstation for the first time may look exactly like how they log into their own

Algorithm 3 Neighbor Similarity Scaling

```
1: procedure NEIGHBORSCALE(G, a, b, f, s)
2:     for b' ∈ G.neighbors(a) do
3:         if b ≠ b' and sim(b, b') then
4:             for f' in G(a, b').features do
5:                 if f == f' then
6:                     s = s * A(f')
```

workstation, however it is still extremely interesting from a security perspective. Conversely, if we consider an edge involving nodes with high degree, such as a web server or file server, utilizing these server resources for the first time is less interesting from a security perspective. Therefore, edges are divided into two groups we call: individual resources and shared resources.

Intuitively, individual resources are those which are tied to specific users and/or services. For these nodes, we *do not* apply the neighbor similarity scaling technique to reduce anomaly scores. On the other hand, the shared resources are intuitively services that are used by large groups of users, or even the entire enterprise. For these nodes, we *do* apply the neighbor similarity scaling technique in order to reduce the anomaly score for edges involving these entities.

In order to separate the two types of entities, we use a metric based on the mean and standard deviation of node degrees across the enterprise. Nodes which have more than the average plus one standard deviation number of edges are labeled as shared resource modes.

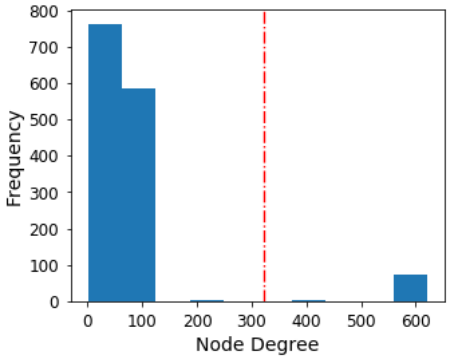


Figure 4.3: OpTC Node Degree Histogram

In the datasets we analyzed, we found that there was a clear distinction between these two distributions. Figure 4.3 shows a histogram of node degrees in the OpTC dataset, with a clear separation between shared vs. individual resources.

The pseudocode for the edge anomaly scoring algorithm can be seen in Algorithm 4. The input graph G is a snapshot of the Cyber Activity Graph at a particular point in time. On lines 2 and 4 we are iterating over each feature of each edge in the graph. We compute the edge anomaly score for the particular feature on line 5, followed by the neighbor similarity scaling technique on lines 7 and 8 if either of the nodes is shared resources. Finally on line 10 the edge anomaly scores are summed and added to the list of edge scores stored in the graph.

Algorithm 4 Cyber Activity Graph edge scoring routine

```
1:  procedure COMPUTEEDGESCORES(G)
2:      for edge ∈ G.edges do
3:          scores = []
4:          for f ∈ edge.features do
5:              s = A(f)
6:              if (a or b) ∈ shared resources then
7:                  s = neighborScale(G, a, b, f, s)
8:                  s = neighborScale(G, b, a, f, s)
9:              scores.push(s)
10:         G(edge).scores.push(sum(scores))
```

Training. Since the edge anomaly algorithm is based on first observation time, it is necessary that we have some type of a training window as otherwise everything that is new will be identified as anomalous as soon as the system starts. Additionally, our goal is not to simply detect new entities within the network, but rather we are trying to detect new behaviors between known entities. To that end, we build and maintain a data structure which indicates if a particular node is in training or not. No edges involving nodes that are in the training phase will be added to the anomaly graph.

In order to handle new nodes that we have never seen before, as well as nodes that may require different amounts of training time, we have designed a training algorithm that allows

for per-node training that can train online and handle new nodes never previously observed. To accomplish this, we maintain a list of all nodes as well as their edge anomaly scores, and a binary indicator which specifies if a node is in training or not. All nodes start in a training mode when they are first observed, with the exception of external IP nodes, which are inherently untrusted and can generate anomalies on first observation. For nodes from internal entities, a new node is in training until two conditions are met: (1) a minimum number of observations have been recorded, and (2) the anomaly score reaches equilibrium. The first condition is trivial to implement, and for the second we choose to calculate the variance of the previous N observations, and declare that equilibrium has been met once the variance of the anomaly terms falls below some threshold γ. In our experiments we have found a value of $\gamma = 100$ performed well. If this equilibrium has been achieved, the node is labeled as no longer training, and will be incorporated into the Anomaly Graph if anomalous edges are identified.

Algorithm 5 Training Update Routine

```
1: procedure UPDATETRAININGSTATUS(G)
2:     for n ∈ G.nodes do
3:         if n.isTraining and n.scores.length() > N and variance(n.scores) < γ then
4:             n.isTraining = False
```

The psuedocode for the training update loop can be seen in Algorithm 5. The per-node scores accessed as *n.scores* are simply a list of the summation of all edge anomaly scores from each iteration of the edge scoring routine.

4.3.5 Anomaly Graph

The Anomaly Graph is a structure which stores the anomalous components identified by the Graph Analyzer and presents them to a human analyst for investigation. At this point in our algorithm, individual edge anomalies are still relatively weak indicators of compromise, and a human analyst would still be required to do a lot of manual inspection in order to piece together an attack campaign. In order to address this, we again develop a technique which

utilizes the graph structure to further weed out false positive events from the true malicious activity. We accomplish this by analyzing the structure of the graph formed by connecting the anomalous edges, and amplifying the most anomalous graph structures.

In graph theory, connected components are a traditional way to analyze graph structures, and are defined as subgraphs in which there is a path that connects each node. We modify this type of analysis slightly, and define our anomalous connected components as those components which share edges that are additionally identified as anomalous by our edge anomaly detection algorithm discussed previously. Additionally, this part of the algorithm also has the ability to baseline the anomaly score of edges based on their history of anomaly scores. For edges that contain an anomaly score history of at least the number of minimum training samples N, anomalous edges are identified as those edges with an anomaly score which is τ standard deviations away from the mean for that particular edge. For edges that do not have sufficient history, anomalous edges are simply labeled with the anomaly score as computed at that time step. This way, the system has a way to baseline anomalous behavior for particular edges such that they won't make their way into the Anomaly Graph if it is typical for a particular edge.

After identifying the anomalous connected components, we use the size of the component as a way to further amplify the anomalous signal. The intuition here is that a single edge anomaly is a relatively weak indicator of malicious activity, however, as anomalies connect together both structurally via anomalous connected components, as well as temporally by our time-based anomaly detection algorithm, they become increasingly more likely to be malicious. Therefore we use the size of the connected anomalous component as a force-multiplier of the anomalous activity. After the anomaly amplification, we have a threshold based calculation that determines if the activity should be published to the anomaly graph, ultimately escalating the activity to a human for investigation.

One full analysis iteration is shown in Algorithm 6. First the realtime Cyber Activity Graph has the edges scored via the *computeEdgeScores* function on line 3 which is detailed

in Algorithm 4. Next, the training update occurs on line 4 which will remove nodes from the training phase and into the evaluation phase as appropriate, which is also detailed in Algorithm 5. Next, the edge anomaly scores are analyzed and if the score exceeds a threshold based on deviation from the mean, the edge is added to a temporary graph structure. After all edges are added to the temporary graph for the current time step, we apply the connected component anomaly amplification by scoring entire components of the graph as opposed to individual edges. If an anomalous component exceeds a predefined threshold ν then the results are added to the anomaly graph on line 16.

Algorithm 6 Anomaly Graph Update Routine

```
1:  procedure UPDATEANOMALYGRAPH(AG, RG)
2:      tmpGraph = newGraph()
3:      computeEdgeScores(RG)
4:      updateTraining(RG)
5:      for (a,b) ∈ RG.edges do
6:          lastScore = RG(a,b).scores.last
7:          mean = mean(RG(a,b).scores)
8:          std = std(RG(a,b).scores)
9:          if lastScore > mean + τ * std then
10:             expected = lastScore − (mean + τ * std)
11:             tmpGraph.addEdge(a,b,score = expected)
12:     ccs = connectedComopnents(tmpGraph)
13:     for cc ∈ ccs do
14:         score = |cc.nodes| * |cc.edges| * sum(cc.scores)
15:         if score > ν then
16:             AG.add(cc)
```

4.4 Evaluation

In this section, we set out to answer several research questions as outlined below:

- Accuracy (RQ1): Can the VGRAPH system accurately detect malicious activity?
- Parameters (RQ2): What is the impact of the various system and algorithm configuration parameters?

- Scalability (RQ3): What is the performance of the VGRAPH system in terms of compute and memory requirements?
- Practicality (RQ4): Is this a viable technique for real-world deployments?

In order to answer these questions, we evaluate VGRAPH on two public datasets: OpTC [92] and Comprehensive Multi-Source Cyber Security Events [55]. High-level dataset details are shown in Table 4.1. In addition, we deploy our system in a real-world network for a live experiment. The VGRAPH system is developed entirely in Python, utilizing the highly efficient NetworkX graphing library [36], in about two thousand lines of code. While our experiments are performed on a server with 56 Intel Xeon E5-2683 CPUs, and 512 GB of RAM, currently the VGRAPH system was developed as a single threaded application. Thus the reported runtime in the experiments below are from a single CPU core.

Table 4.1: Evaluation Datasets

	OpTC	**LANL**
Duration (Days)	8	58
Attacks (Days)	3	18
Computers	625	17666
Users	627	10941

We report three accuracy metrics: (P)recision, (R)ecall, and (F1)-score, where

$$P = \frac{TP}{TP+FP} \qquad R = \frac{TP}{TP+FN} \qquad F1 = 2 * \frac{P*R}{P+R}$$

and TP, FP, and FN are the number of true positives, false positives, and false negatives, respectively. The finer details of the labeling scheme may vary depending on the quality and granularity of the red team labels, and is discussed for each evaluated dataset.

4.4.1 OpTC Evaluation

The OpTC dataset [92] was produced during the DARPA Transparent Computing program, and contains data from an orchestrated network of roughly 600 hosts and 600 users. Each

host was installed with a monitoring agent which generated highly verbose and granular telemetry which is similar to the type of information available from *Windows Event Logs*, *Sysmon*, or *auditd*. The data gathering exercise ran for roughly eight days, with simulated user activity on all days, and three days of malicious red team activity. Each day of the red team activity was a different attack campaign involving largely different hosts and users, as well as different tactics, techniques, and procedures. All attacks are highly documented, and, when combined with the high-fidelity host logs, allows for a very granular analysis of the NETHAWK system. We use this dataset in particular to analyze accuracy metrics (RQ1), configuration parameters (RQ2) and scalability (RQ3).

Labeling. Due to the high granularity of events and labels, we take a strict labeling approach and score a true positive where our anomaly graph contains an edge between nodes (a,b) and the ground truth red team notes also contain information that would indicate that malicious activity took place between nodes (a,b) on the day that the anomaly graph alert was generated. In some cases, our system generated alerts which were not explicitly in the ground truth red team notes, however, upon further investigation, we determined many were malicious as well, by identifying obviously malicious activity (e.g., running malware, connecting to the C2 server, etc). In these cases we recorded our detections as true positives. Further details of these cases will be discussed below when we analyze the attack detections. A false positive is recorded for every edge in our anomaly graph (a,b) where either node a or node b are not found to be in the red team notes. False negatives are identified by looking at all edges in the Cyber Activity Graph for a particular day, and identifying any edges (a,b) that are in the Cyber Activity Graph, found in the red team notes, but missing in the anomaly graph. In this dataset, unfortunately, there were no logs captured on the Domain Controller, which was involved in 2 of the 3 attack days. Because we do not have visibility into this system we do not count the missing malicious domain controller activity as false negatives.

Preprocessing. We utilize a subset of the host telemetry from the OpTC dataset corresponding to network, file, and process events across all users and hosts in the environment.

Specifically, we process records with the event types of: *FILE_READ*, *FLOW_START*, and *PROCESS_CREATE*. Each record is further attributed with a security principal which represents the user associated with the record. We only process records that can be directly attributed to a domain user account, which, in the case of the OpTC dataset, are records that involve principals of the *systemiacom* domain. Further details of the specific elements of the red team campaigns will be found in the attack analysis section below. For even more detail on this dataset we recommend the reader to the notes found in the data repository [92], or a recent analysis of the dataset [8].

For testing on the OpTC dataset we configured our system as shown in Table 4.2. We will take a closer look at the impact of these parameters in the parameter analysis section below.

Table 4.2: NETHAWK System Configuration

Parameter Name	**Variable**	**Value**
Anomaly Half-Life	$t_{1/2}$	6
Neighbor Similarity Factor	β	0.3
Minimum Training Samples	N	24
Training Variance Threshold	γ	100
Anomalous Edge Factor	τ	5
Anomaly Graph Threshold	ν	100

4.4.1.1 Accuracy Analysis

First we will look at the detection accuracy of our anomaly detection algorithm. Table 4.3 shows results on the OpTC dataset. The first row contains the score for all results generated on the full dataset. This includes results generated during the entirety of the dataset, including the three days that contain attack data, as well as the days preceding that contain no attack data. From the results in the first row, we can see that our anomaly detection algorithm did not generate a significant amount of false positives on days outside of the red team activity. The bottom three rows of Table 4.3 are scored specifically based on the known ground-truth malicious activity for each individual day of the attack campaign.

We will dive into each day below, however at a high level we can see that on each day NETHAWK was able to detect the majority of the malicious activity while remaining highly precise.

Table 4.3: Accuracy results on OpTC Dataset

Data	**Precision (%)**	**Recall (%)**	**F1 (%)**
All	90	98	94
Attack Day-1	92	96	94
Attack Day-2	86	99	92
Attack Day-3	100	100	100

Next we performed a similar analysis, however this time only considering specific edge types in isolation. We generate the F1 score with only a single edge type present in the graph. Table 4.4 shows that, on different attack days, different edge types contributed to the successful detection of the malicious activity. For example, on day-1, the *process* and *file* edges brought the F1 score from 85% to 94% when all combined. Conversely, on day-2, the *process* and *file* activity actually detracted from the F1 score by introducing false positives, and reducing it from 97% to 92%. Day-2, as discussed below, is characterized by a large amount of anomalous network activity spawned by a automated hacking tool, which is the reason why the *network* edges on this day specifically are so indicative of the malicious activity. On day-3, we can see that when only considered in isolation, *file* and *process* edges fail to detect any of the malicious activity, however when combined with *network* edges, we are able to detect 100% of the malicious activity. Importantly, we see that the best average F1 score across all attack days is achieved when utilizing all of edge types in concert.

Table 4.4: F1 Accuracy (%) results for isolated edge types

Edges	**Day-1**	**Day-2**	**Day-3**	**Avg**
Network	85	**97**	92	92
File	76	25	0	34
Process	79	27	0	35
All	**94**	92	**100**	**95**

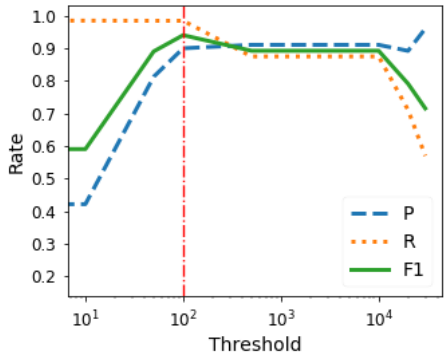


Figure 4.4: Accuracy of attack detection at different Anomaly Graph threshold values

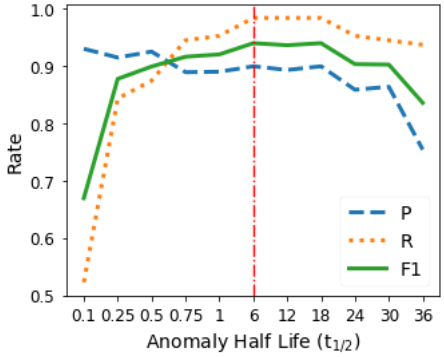


Figure 4.5: Accuracy of attack detection at different Anomaly Half-Life values

4.4.1.2 Parameter Analysis

There are several configuration parameters in our system that can impact the ability to detect malicious activity. In order to understand these parameters better, we vary each individually and capture accuracy metrics of the system.

One of the most important parameters in our system is the Anomaly Graph threshold ν which is the final threshold that must be exceeded before an anomalous connected component is added to the Anomaly Graph and an alert is effectively generated. Figure 4.4 shows the accuracy metrics when varying this threshold. The large plateau in performance in the middle of the plot indicates that there is a wide range of values that lead to reasonable accuracy with F1-scores near 90%. Our chosen threshold of 100 is shown as the vertical red line.

Another important parameters in our system is the anomaly half-life parameter $t_{1/2}$. Figure 4.5 shows the accuracy of our system when varying the half life parameter $t_{1/2}$ across a range of values. From the plot we can see that we have the lowest recall values at the smallest anomaly half-life values below 1 hour. This is because the system will effectively "forget" anomalies too quickly before it can properly stitch the malicious activity together. Similarly, a value above 30 hours results in a sharp decline in precision due to the system retaining and erroneously connecting unrelated anomalous activity causing false positives.

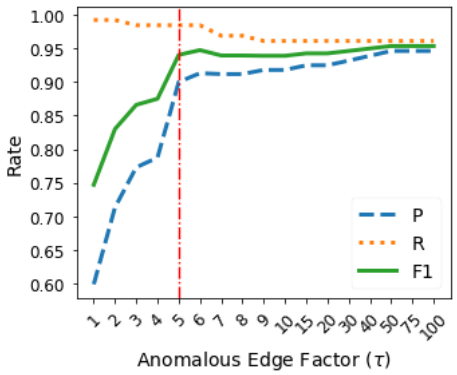


Figure 4.6: Parameter Analysis of Anomalous Edge Factor (τ)

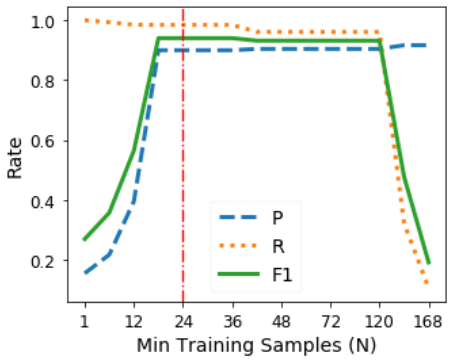


Figure 4.7: Parameter Analysis on Minimum Training Samples (N)

Our chosen threshold of 6 hours is shown as the vertical red line.

Figure 4.6 shows the detection performance of the system on the OpTC dataset when varying the anomalous edge factor τ. This factor is used to determine when an existing edge with sufficient history has become anomalous. When τ becomes large, it becomes less likely that existing edges will ever get into the anomaly graph. We can see that this occurs around $\tau = 100$, as the performance stabilizes and does not change after this point. This indicates that a large amount of edges in the red team activity were in fact new edges in our graph that did not have any anomaly history, which is why the accuracy is so high even when this factor is effectively not contributing. However, we also noticed that the loss in recall from this approach is significant, as typically what this parameter impacts is the ability to detect patient 0 (e.g. on Day-1 of the red team activity in OpTC, the malicious activity of user *zleazer* with their own workstation *sysclient0201*). If τ is too low, however, there is a significant loss in precision, as many existing edges can easily become anomalous. We chose a threshold here of 5 in order to optimize for recall, so that we would be able to detect patient 0 activity, as this is often critical to knowing the full scope of the attack.

Figure 4.7 shows the accuracy when adjusting the minimum training samples parameter N. With a value too small, the system will generate a large number of false positives early on results as nodes will leave the training phase before a sufficient amount of behavior has

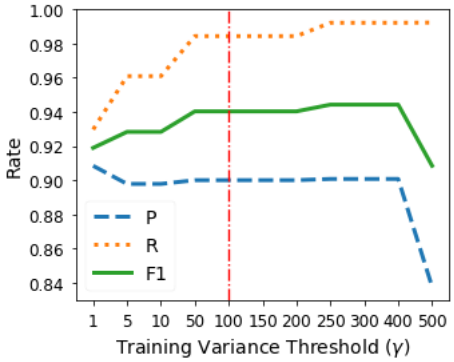


Figure 4.8: Parameter Analysis on Training Variance Threshold (γ)

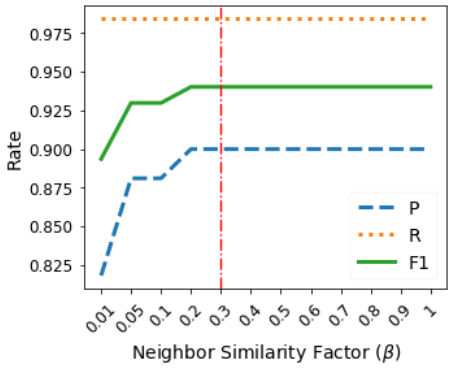


Figure 4.9: Parameter Analysis on Neighbor Similarity Factor (β)

been observed, which leads to poor performance. After about 20 samples (or equivalently hours in our experiments), The performance improves significantly. Our chosen threshold of 24 hours/samples is shown as the vertical line. After about 120 samples/hours, we are requiring too much training time and our accuracy sharply declines as our training bleeds into the red team exercise which causes the system to miss the malicious activity.

Figure 4.8 shows the accuracy when varying the training variance threshold. This threshold is only checked after the minimum number of training samples is satisfied, which is why the performance is relatively high across the board. For very low values, we can see that the system has the lowest recall, as some nodes will never leave training due to anomaly scores not reaching equilibrium. As we increase the value, we see a steady rise in recall. Once we get to a variance threshold of 500, we are allowing nodes to leave the training state too early, which introduces false positives into the results, and a loss of precision.

Figure 4.9 shows the effect of the neighbor similarity factor, which controls how similar two nodes have to be to use their edges as proxy edges during neighbor similarity scaling stage of the anomaly detection algorithm. This is largely a false-positive reduction technique, which is illustrated by the fact that recall is essentially static in the figure. For small values of β, the algorithm is very strict in determining similarity, which leads to nodes not being scaled by similar neighbors, which leads to higher anomaly scores. A value of greater

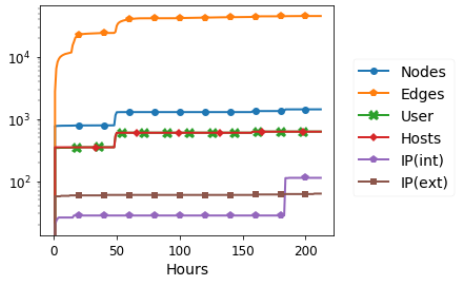


Figure 4.10: OpTC Graph Size vs Time

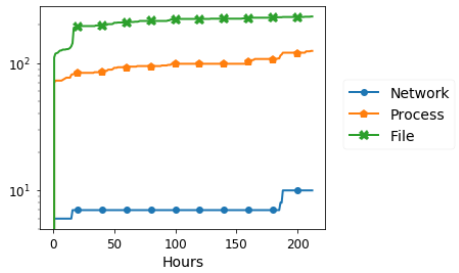


Figure 4.11: OpTC Edge Features vs Time

than 0.2 results in reducing false positives via the neighbor similarity scaling technique, which reduces the anomaly scores of nodes that are found to be similar and within the same neighborhood.

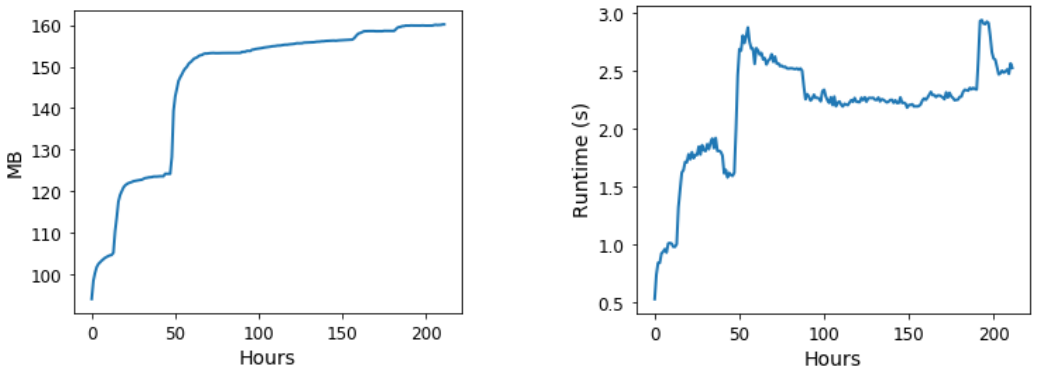
4.4.1.3 Scalability

Next we will look at the scalability characteristics of NETHAWK on the OpTC dataset.

Graph Size. Figure 4.10 shows the size of the Cyber Activity Graph over time. Notice the super linear growth in the first few hours of the dataset, tailing off after about 50 hours of system operation. During the first hours of operation, the system is observing a large quantity of new activity, which leads to this aggressive growth in size. After roughly 50 hours, the size of the graph is relatively static, with only a small upward trend. The large spike of internal IP nodes near the end of the chart is a manifestation of the noisy day-2 red team attack campaign which we will discuss further below in the attack analysis section.

Feature Size. The graph structure is accompanied by a set of features on each edge corresponding to the type of activity observed and the timestamp it was first seen. Figure 4.11 shows the number of unique features observed across all edges in the graph for each edge type. Again we see the initial super linear growth followed up by a relatively slow and steady increase over time.

Memory. Figure 4.12a shows the memory consumed by our system over the course of the exercise. We can see a similar behavior with a large increase in memory consumption



(a) Memory

(b) Runtime

Figure 4.12: OpTC Memory and Runtime Characteristics

in the first 50 hours, followed by the steady state period with only a minor increase. The overall memory footprint is less than 160 MB for this dataset, indicating that this technique could scale to a much larger network.

Runtime. Figure 4.12b shows how long the system took to run one analysis iteration per every hour of the dataset. In general the analysis runtime is short, less than 3 seconds of analysis per one-hour of realtime data that has elapsed. There is an initial increase in runtime in the first few days as new activity as aggregated into the Cyber Activity Graph, and after this point the runtime only shows a slight trend upwards. This again indicates that this technique can scale to a much larger network with ease.

4.4.1.4 Training Analysis

One of our core hypothesis in this work is that the activity captured in our Cyber Activity Graph will reach a steady-state that will allow for our anomaly detection algorithm to correctly identify the malicious activity. In our training algorithm as discussed previously, nodes in our graph start out in a training phase where they do not generate any alerts, and only after a steady-state is reached do they leave the training phase. Figure 4.13 shows a histogram plot of node training duration. The plot shows three clear distributions. The first distribution of nodes leaves the training phase at or nearly after the minimum number of

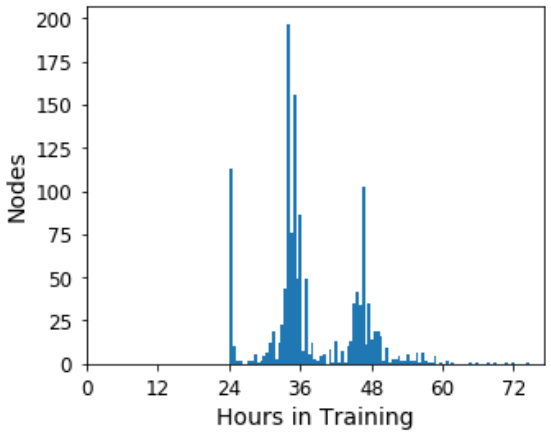


Figure 4.13: Node Training Duration Histogram

observed samples, which in our case was 24 hours of activity. Following this, there are two distributions centered around 35 hours and 48 hours. These distributions illustrate the need for a flexible training strategy, as nodes will have different roles in the network and thus require different amount of history to baseline their activity.

In addition to understanding the duration of the training phase, it is also important that we understand when, in terms of the OpTC dataset, the nodes in our graph are being evaluated for anomalies. Figure 4.14 shows the ratio of nodes in the evaluation phase (i.e. no longer training) across the time span of the dataset. We can see from this plot that roughly 90% of all nodes in the dataset are in evaluation mode roughly 3 days into the exercise, After this point, we see a slow but steady increase in evaluation nodes as new nodes are observed throughout the duration of the exercise. The plot also highlights the redteam activity period which we can see begins long after the majority of nodes are being evaluated for anomalies.

4.4.1.5 Attack Analysis

Next, we will dig into the results on a per-day, per-attack basis. We will analyze where our system succeeded, and where it failed, and how it could be improved.

Attack Day 1. The first day of malicious activity involved a user account *zleazer* and their workstation *sysclient0201* which was compromised and attacker controlled. The

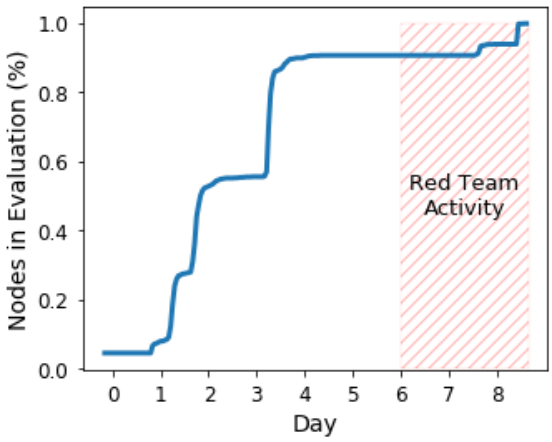


Figure 4.14: Percentage of nodes in the evaluation phase over the duration of the OpTC dataset.

attacker was controlling the compromised system through the C2 server at IP *132.197.158.98*. The attacker first moved laterally to *sysclient0660* and *sysclient0402*. From these systems, a domain admin account *administrator* login credentials were harvested from memory with the well-known tool Mimikatz [22]. At this point, the attacker was able to pivot to the Domain Controller where they further spread to a variety of systems using the compromised account *zleazer*.

On the first day of attack activity our system achieved an F1 score of 94% as shown in Table 4.3, achieving 92% precision and 96% recall. Figure 4.15 shows the Anomaly Graph for the activities that were detected during the day-1 attack campaign. Nodes are colored based on the time when they were aggregated in the Anomaly Graph. The graph was updated three times throughout the day, as indicated by the three individual colors. At the center is the compromised user account *zleazer*. We can see that *zleazer* had anomalous activity detected on their own workstation: *sysclient0201*, in addition to many other systems in the environment. The C2 communication is clear here as the only external IP node in the Anomaly Graph. We can also see clearly the *administrator* account that was moved to laterally, as well as the systems involved in that lateral movement: sysclient0660 and sysclient0402. Despite not having visibility on the domain controller, we were able to detect

the large amount of lateral movement that occured after domain controller compromise due to the usage of the compromised *zleazer* account.

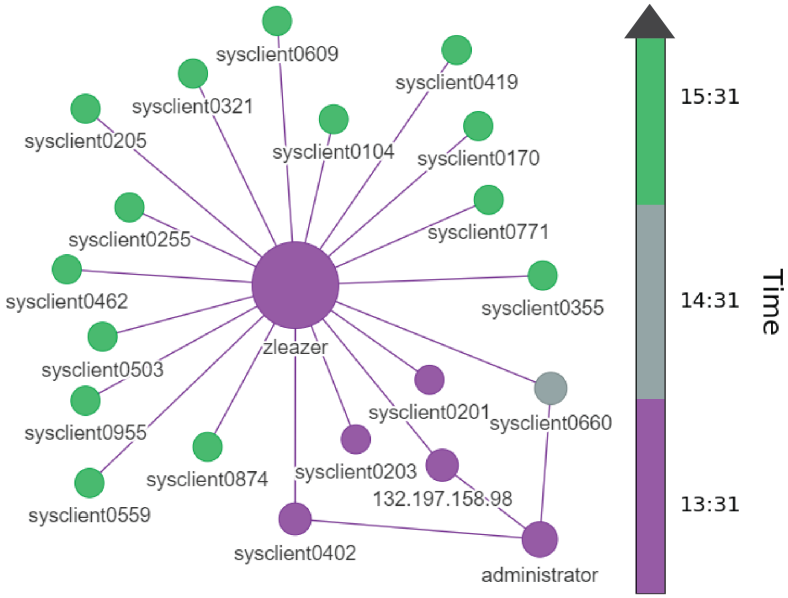


Figure 4.15: OpTC Attack Day-1 Anomaly Graph

In addition to this large connected component displayed in Figure 4.15, there is also a smaller, 3 node component that was detected on day-1 (omitted in the figure) that was counted as a false positive. This component involved a large amount of new activity between a couple of users in the system. This component, despite making its way into the anomaly graph, is easily distinguishable from the malicious activity as it covers only a small amount of activity. The only false negative our system incurred involved the compromised user account *hdorka*. This account was compromised on the domain controller, which as mentioned prior, we do not have logs from. This points to a strong need for complete information for this type of approach to work. If there is a system or user that has no visibility and an attacker happens to laterally move through that entity, it could result in the graph components not being connected accurately, and ultimately the threshold not being exceeded and the attack going either entirely or partially undetected.

Attack Day 2. The activity on day-2 was arguably the noisiest, most obvious activity,

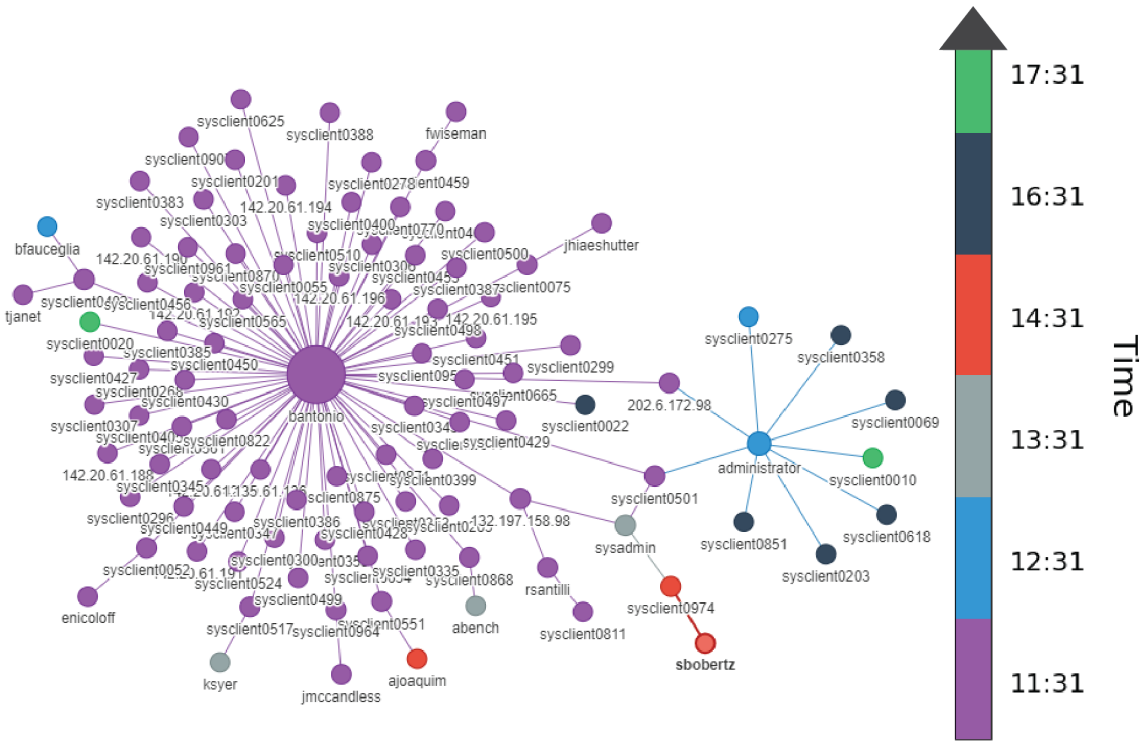


Figure 4.16: OpTC Attack Day-2 Anomaly Graph

with a compromised user *bantonio* running an automated exploitation tool called *DeathStar* [19] on their workstation *sysclient0501*. The purpose of this tool is to automatically discover vulnerabilities and weaknesses in Active Directory, and exploit them to achieve Domain Admin privileges. After the *bantonio* user successfully ran the *DeathStar* tool, they again moved laterally to the Domain Controller, where they deployed additional agents on various systems using the compromised *administrator* account. The C2 server for this attack campaign was located at an IP address of *202.6.172.98*.

On the second day of the attack our system achieved an F1 score of 92% as shown in Table 4.3. The Anomaly Graph for the activity that occurred during the day-2 attack campaign is shown in Figure 4.16. Nodes are colored according to the time in which they were added to the Anomaly Graph. The attack began with the large cloud of connections around the *bantonio* user as the result of running the *DeathStar* automatic exploitation tool. It is clear that this level of activity is very noisy, and very obvious when doing this type

of graph analysis. We can also see the other compromised user accounts of *administrator* and *sysadmin* which were added to the graph after the domain admin was achieved by the *DeathStar* tool. We can again see the C2 server in the graph shared between *bantonio* and the *administrator* account.

This day introduced the most false positives, and consequently had the lowest precision of 86%. This was due largely to the fact that the *bantonio DeathStar* process touched so many unique systems that it inadvertently amplified some activity that would have never been escalated into the anomaly graph. These manifest in the graph as weakly connected leaf nodes around the perimeter of the large *bantonio* cloud. There is likely some pruning that we could do to remove some of these false positives, which we leave to future work.

Attack Day 3. The last day of the exercise had the least amount malicious activity. The attack was based on a malicious update of a well known text editing program *Notepad++*. Two users, *dcoombes* and *bbateman* performed an update which provided attackers a backdoor into their system. Once on the system, the attackers interacted with the local workstations of the compromised users, only performing some enumeration and local persistence actions. No further lateral movement or exploitation was performed. The C2 server for this exercise was located at *53.192.68.50*. In addition to the two users that were compromised, we found several other users that also were communicating with the malicious C2 server at this time. We assume that these were systems that were compromised but not utilized by the attackers. For this reason we consider them as also malicious.

On the final day of activity we achieved a F1 score of 100% as shown in Table 4.3. The fact that we did so well on day-3 is related to the fact that the last day of the exercise had the least amount of activity in general. Based on the data, it seems likely that the exercise was winding down on this last day, and this attack was only half-baked. That being said, our system was still able to detect the malicious activity. Figure 4.17 shows the full attack graph for day-3, with the nodes colored according to the time they were added to the Anomaly Graph. We can see that the system captured the anomalous and malicious activity

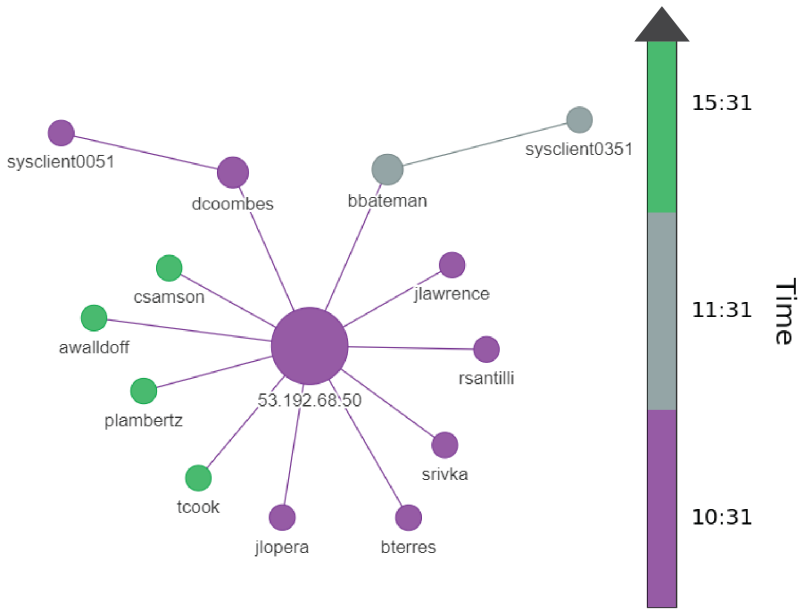


Figure 4.17: OpTC Attack Day-3 Anomaly Graph

between *dcoombes*, *bbateman*, and both of their respective workstations *sysclient0051* and *sysclient0351*. We can also see the attacker C2 server at *53.192.68.50*. All of the various other users interacting with the C2 IP address were not further exploited by the attackers, although we assume they were compromised as well based on the fact that they were calling back to the C2 infrasturcture. Had all of these various accounts not also been compromised, the activity on this day would have likely been much more challenging to detect.

4.4.2 LANL Evaluation

The second dataset we utilized was the Comprehensive Multi-Source Cyber Security Events dataset [55] published by Los Alamos National Labs (LANL), which contains 58 days of live data captured from the real-world LANL network. Additionally, red team activity was performed over the duration of the exercise, and the related malicious authentication events are labeled in the dataset. However, due to the fact that this dataset is from a real-world environment, the data is highly anonymized, which makes granular interpretation of the results challenging if not infeasible. Additionally, this dataset is orders-of-magnitude larger

than the previous, making it much more challenging and time consuming to analyze. For these reason we use this dataset to evaluate accuracy metrics (RQ1), scalability (RQ3), and practicality (RQ4), and omit the fine-grained attack analysis, and configuration parameter analysis, as done previously.

Labeling. In order to evaluate accuracy on this dataset we relaxed our labeling scheme slightly from the previous experiment, and instead label an edge (a,b) in our Anomaly Graph as a true positive when either node a **or** node b is found in the red team labeled activity on a particular day. We did this due to the relatively coarse labeling of the red team activity, which only specified authentication events as either benign or malicious, and did not indicate what other activity may have been performed on the compromised systems. Edges in the Anomaly Graph for which there is no corresponding entry for either node in the red team events on the day of detection are labeled as false positives. For computing false negatives, we again modify our labeling scheme from the previous dataset and instead look at the set of entities involved in the red team campaigns, and count a false negative for every entity that was omitted in an Anomaly Graph. This was again due to the coarse red team labels, and highly anonymized data.

Preprocessing. In order to analyze the LANL dataset, we extracted process and flow events, and added them to our graph as appropriate. This dataset did not contain file information, however, it did contain authentication information. Therefore, we modified the Cyber Activity Graph structure slightly, and added a new edge type for authentication events. The authentication events were added to the graph in a manner similar to previously described, where the edge is attributed with the type of authentication and the first observation time. We utilized explicitly NTLM authentication events in our experiments. This illustrates how easy it is to add new event types to the Cyber Activity Graph.

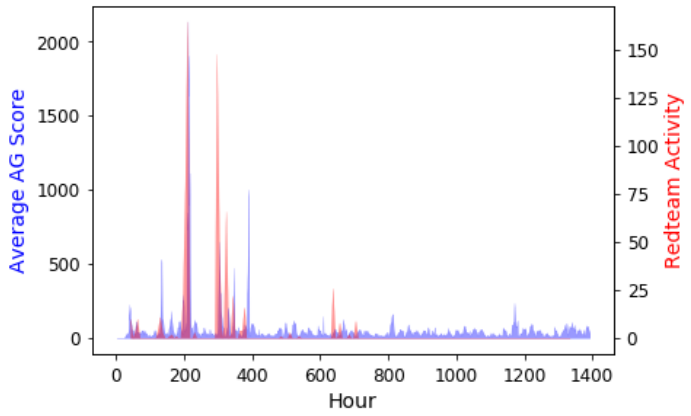


Figure 4.18: Average Anomaly Graph score and red team activity over duration of LANL dataset

4.4.2.1 Accuracy Analysis

During analysis, we found that an individual system always dominated our results, and generated a large Anomaly Graph component that dwarfed all others. Upon investigation, we found that this was related to a system labeled as *C15244*. This is a shared resource in the computing environment, which around the time of the red team events started providing a new service on port 69 (typically associated with the TFTP protocol) which was utilized by many systems in the environment. This is highly anomalous, and possibly malicious, however it does not coincide with any of the labeled red team events. Therefore, we decided to note the activity, and whitelist the system in order to best analyze the other results without the noise added by this individual system.

Figure 4.18 shows the average anomaly graph score over the full 58 day period. We can see that, during the red team activity, there is an elevated average Anomaly Graph score, indicating that there is malicious activity occurring and that it is being captured in our Anomaly Graph representation. If we zoom in to the time range containing the most red team activity as shown in Figure 4.19, we can see there is a visible spike in the average Anomaly Graph score for each corresponding spike of red team activity. This is a good indicator that our system was able to detect the malicious activity.

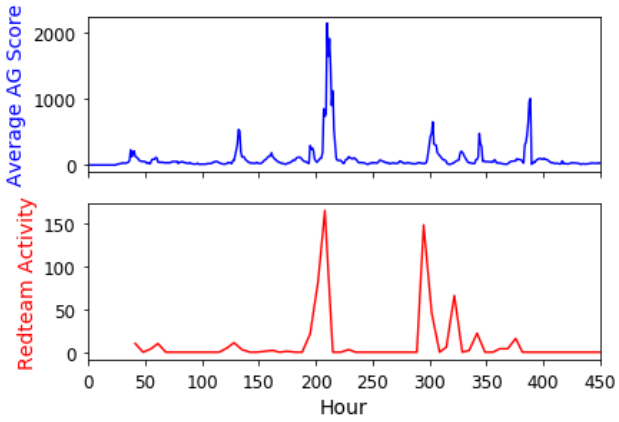


Figure 4.19: Average Anomaly Graph score and red team activity during most active red team hours in LANL dataset

To further analyze what is contained in these Anomaly Graph spikes, we captured the top-k highest scoring Anomaly Graphs generated by the system. The results are shown in Table 4.5. The top-1 Anomaly Graph captures the vast majority of entities involved in the red team activity, while also remaining highly precise at 72%. When we expand to the top-100 Anomaly Graphs, we capture over 99% of the red team entities, retaining a precision of 61%. This illustrates the ability of NETHAWK to detect malicious red team activity in much larger networks. Notice, however, the significant difference in Anomaly Graph threshold, which is measured in millions in Table 4.5, compared with the OpTC experiment with an optimal threshold of only 100. This is due to the significant size difference of these two networks. However despite the increased threshold size, the red team activity still manifested as large connected anomalous components which was detected by the system.

Table 4.5: Top-k Anomaly Graph Accuracy Results on LANL

Top-k	**Threshold (M)**	**Precision (%)**	**Recall (%)**	**F1 (%)**
1	94	72	93	81
2	92	71	93	80
5	72	69	94	79
10	48	65	98	78
100	9	61	99	78

4.4.2.2 Scalability

Next we will again look at the scalability of the system when applied to this dataset which is orders of magnitude larger than the OpTC dataset. Figure 4.20 and 4.21 show plots of the size of the graph and edge features. These plots mirror the previous plots in the OpTC experiments, showing a quick, almost immediate growth, followed by a small upward trend.

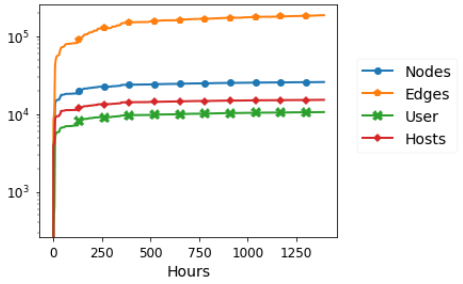


Figure 4.20: LANL Graph Size vs Time

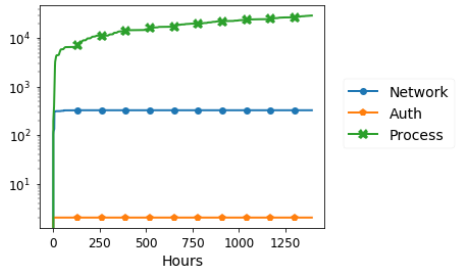
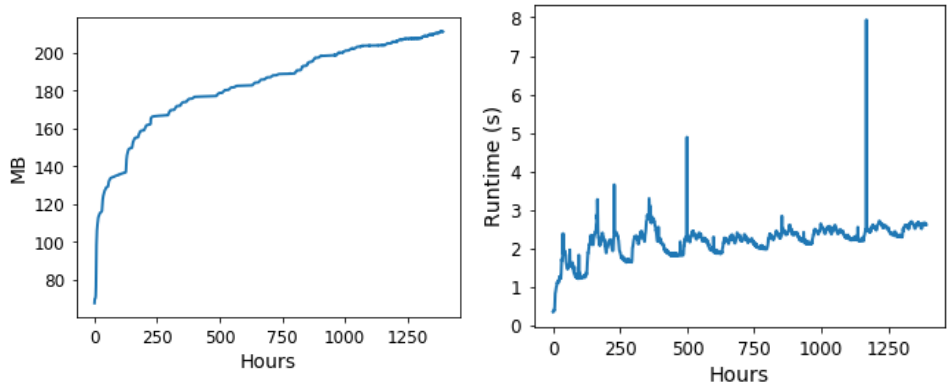


Figure 4.21: LANL Edge Features vs Time

Figures 4.22a and 4.22b show memory usage and runtime for the LANL dataset respectively, which also mirror the previous plots from the OpTC experiment. The memory usage is more in the case of LANL, although not by a considerable amount, with total memory consumption of less than 300 MB for the full 58-day graph. The runtime plot shows a very obvious weekly cycle that is typical of enterprise network activity. There are some occasional spikes that we found were due to new activity related to high degree nodes undergoing the neighbor scaling technique as described in Section 4.3, which in the worst case requires scanning through all neighbors of a particular node. Regardless, the run time still averaged around three seconds per one-hour of realtime activity, with the spikes only reaching about 8 seconds.

4.4.3 Live Experiment

We had the opportunity to deploy our system in a real-world operational network and monitor real-time security event feeds across network and host based telemetry. Our system moni-



(a) Memory

(b) Runtime

Figure 4.22: LANL Memory and Runtime Characteristics

tored the network for one month of time, during which a red team exercise was performed. During this exercise, a professional red team was given access to a compromised user credential, and over the course of roughly one week proceeded to exploit the environment, ultimately resulting in a domain admin account takeover.

During the month-long deployment, NETHAWK generated a total of four incident reports, or, in other words, four anomalous connected components in the Anomaly Graph. Three of these were determined to be unrelated to the malicious activity, despite containing a large amount of unusual authentication behavior. The incident report that we generated relating to the red team attack campaign was scored by a third party based on known ground-truth results, and was found to have the accuracy metrics displayed in Table 4.6.

Table 4.6: Accuracy Results on Live Experiment

Entity Type	**Precision (%)**	**Recall (%)**	**F1 (%)**
User	68	42	52
Host	64	36	46

From these results, we can see that the NETHAWK system was able to detect a large amount of the malicious red team activity, despite not utilizing any signatures of known malicious activity, and instead focusing on identifying connected temporal and structural anomalies to identify the red team attack campaign. At the enterprise scale, we do generate

some false positives related to other connected anomalous activity, however we believe that four total alerts over the course of a month for a large enterprise is within reason for a security analyst to triage.

4.5 Related Work

There are a myriad of commercial and open source options for detecting malicious activity in enterprise computer networks, however the vast majority of them focus primarily on signature-based detentions. Snort [102] and Suricata [105] are two popular open source intrusion detection system (IDS), but they both require rules that define what malicious activity looks like, and thus will always be vulnerable to APT circumvention.

On the other hand, Endpoint Detection and Response (EDR) systems, such as the open source OSSEC [93], run software on hosts and can detect malicious activity and some behavioral anomalies. Unfortunately, they often lack the big picture visibility across the full enterprise, and also are highly dependent on signatures.

In addition, the Security Information and Events Management Systems (SIEM) such as Splunk [104] or ELK [26] aim to collect and correlate activity across an enterprise, however they are also typically rule-driven when it comes to detecting malicious activity. Some of these tools contain anomaly detection capabilities, however they are frequently limited to time series analysis on individual columns of data, and again lack the full enterprise context.

In the academic space there are many works that model and detect malicious activity via provenance or causal analysis of host events. For example, a recent work Holmes [86] builds graphs of host-level events, and then correlates them to identify likely stages of an attack campaign. Similarly, Poirot [84] models attacker behavior as a query graph, and searches for attacks on a system via the kernel audit records. Another system RapSheet [39] is another technique where attacker activity is modeled as a graph and identified in a host provenance graph. A common thread of these related approaches, as well as many other provenance-based works [42, 118, 38, 116], is that they typically focus on a set of

pre-defined attacker behaviors on individual hosts. In addition, the output of these systems can be cumbersome for an analyst to inspect, as host-based provenance graphs can be quite large and often suffer from dependency explosion. A recent paper WATSON [121] focuses on abstracting behaviors from these graphs to reduce the burden on analysts.

Other related works utilize anomaly detection as their primary detection method. The NoDoze system [40] is based on identifying anomalous paths in host-based provenance graphs in order to detect malicious activity. Also, the PrioTracker [74] system identifies anomalous causal relationships via graph-based structural and statistical anomalies. The Kitsune [87] system is based on identifying anomalous network activity to detect network intrusions. A similar system [17] utilizes graph-based link prediction algorithms to identify anomalous authentication activity. The previous two systems, however, only analyze a subset of behavior visible across the network (flow, and authentication, respectively) and thus are not able to capture and describe the full extent of the malicious activity.

Other approaches, such as DeepLog [24] and others [91] forgo the conversion to the graph domain, and instead learn to distinguish normal from abnormal log events via machine learning and statistical analysis. Others simply provide an effective SQL-style query mechanism purpose-built for cybersecurity log analysis [32, 33].

By comparison, the NETHAWK system does not rely on any signatures, or definitions of malicious activity, which means that it will be better suited to identifying zero-day malware, compromised credentials, and insider threats. Further, the NETHAWK system models network-wide dynamics, as opposed to individual hosts, which allows for the system to provide a more complete account of malicious activity that spans multiple systems and services across an enterprise. And lastly, due to the compact yet expressive representation of the Cyber Activity Graph, the NETHAWK system is able to monitor large-scale real-world enterprise computer networks with minimal memory and compute requirements.

4.6 Conclusion

In this chapter we introduced NETHAWK, a system for monitoring and detecting Advanced Persistent Threats in enterprise computer networks based entirely on anomalous activity. We discussed the pitfalls of traditional signature based approaches, and how behavioral detection techniques are the key to our future security. We discussed our Cyber Activity Graph and the Anomaly Graph data structures, as well as our anomaly detection algorithm based on structural and temporal graph anomalies. We applied our system to two datasets representing real-world enterprise computer networks, and demonstrated how NETHAWK can detect sophisticated malicious activity spanning multiple users and systems with high precision and recall. Further, we showed how our system can be used to analyze large-scale networks with minimal memory and compute requirements. Finally, we deployed our system in a real-world network and demonstrating it's ability to detect attacks in a real environment.

Chapter 5: Conclusion & Future Work

In this dissertation we introduced and discussed some of the key challenges we are facing in cybersecurity today. In Chapter 2, we discussed software security, and the challenge of identifying vulnerable code clones accurately and scalably. We introduced our VGRAPH vulnerability representation and detection system based on code property triplets of the vulnerable code, the patched code, and the contextual code, and used this system to accurately identify highly modified vulnerable code clones. In Chapter 3, we discussed network security, and the challenge of identifying malicious authentication activity associated with lateral movement, a critical step in APT attack campaigns. We introduced our graph structure and graph AI algorithm for identifying anomalous authentication edges in order to detect lateral movement activity. In Chapter 4, we discussed threat hunting and APT attack campaign detection, and the challenges associated with stitching together related malicious events in order to fully capture the attack. We introduced our NETHAWK system for identifying APT attack campaigns by correlating temporal and structural anomalies in our Cyber Activity Graph data structure.

Despite making significant research contributions, the three state-of-the-art techniques discussed in this dissertation are still largely reactionary, relying on effective detection of malicious activity in order to provide improved security. While monitoring for malicious activity will always be important, it is also critical that we begin to implement more proactive defensive measures. This means not only detecting, but predicting the most likely attacks that could result in the most catastrophic results if successful, and remediating these underlying vulnerabilities prior to any exploitation efforts.

In future work, we plan to incorporate more information about the attack surface into our graph structures and algorithms. Specifically, there is a wealth of data pertaining to the vulnerability level of systems and software, reputation scores of IP addresses and other

Internet entities, risk levels of users, etc., that could be used to act in a proactive manner by identifying the most likely attack scenarios that would cause the biggest impact. This will require innovation in the form of graph-based algorithms that can compute complex reachability metrics of entities and paths in the network based on some ground truth understanding of the network as a whole (e.g. vulnerable systems, untrusted IPs, etc), as well as an understanding of the attacker (e.g. typical lateral movement techniques, common command-and-control methodology, etc). This way, critical flaws in the security controls of an environment can be flagged for inspection such that they can be patched prior to exploitation by a malicious actor.

The goal of this dissertation is to provide scientifically grounded motivation for the usage of graph data structures and graph algorithms for cybersecurity. Our cyber defenders are in need of techniques that go beyond simple signature detection, and can better utilize the wealth of information that is being generated in enterprise computer networks. Our current cybersecurity techniques rely far too much on the expertise of human analysts, which benefits our attackers who utilize programmatic approaches to identify weaknesses that they can then exploit with ease. With tools and techniques such as those discussed in this dissertation, we believe our cybersecurity defenders will be better able to stay ahead of critical threats and keep our networks safe from attack.

Bibliography

[1] Bugtraq. https://securityfocus.com.

[2] Github. https://github.com. Accessed: 2018-07-27.

[3] Joern. http://www.mlsec.org/joern/.

[4] National vulnerability database. https://nvd.nist.gov.

[5] Rough audit tool for security. https://github.com/andrew-d/rough-auditing-tool-for-security.

[6] Vuddy web service. https://iotcube.korea.ac.kr/. Accessed: 2018-07-27.

[7] Lillian Ablon and Andy Bogart. *Zero days, thousands of nights: The life and times of zero-day vulnerabilities and their exploits*. Rand Corporation, 2017.

[8] Md Monowar Anjum, Shahrear Iqbal, and Benoit Hamelin. Analyzing the usefulness of the darpa optc dataset in cyber threat detection research. In *Proceedings of the 26th ACM Symposium on Access Control Models and Technologies*, pages 27–32, 2021.

[9] Apache. Kafka. https://kafka.apache.org/. Accessed:2021-01-16.

[10] Stefan Axelsson. The base-rate fallacy and the difficulty of intrusion detection. *ACM Transactions on Information and System Security (TISSEC)*, 3(3):186–205, 2000.

[11] Frederick Barr-Smith, Xabier Ugarte-Pedrero, Mariano Graziano, Riccardo Spolaor, and Ivan Martinovic. Survivalism: Systematic analysis of windows malware living-off-the-land. In *Proceedings of the IEEE Symposium on Security and Privacy*. Institute of Electrical and Electronics Engineers, 2021.

[12] Bibek Bhattarai, Hang Liu, and H Howie Huang. Ceci: Compact embedding cluster index for scalable subgraph matching. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1447–1462, 2019.

[13] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 129–138. ACM, 2012.

[14] Benjamin Bowman and H Howie Huang. Nethawk: Hunting for advanced persistent threats via structural and temporal graph anomalies. Submitted.

[15] Benjamin Bowman and H Howie Huang. Vgraph: A robust vulnerable code clone detection system using code property triplets. In *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 53–69. IEEE, 2020.

[16] Benjamin Bowman and H Howie Huang. Towards next-generation cybersecurity with graph ai. *ACM SIGOPS Operating Systems Review*, 55(1):61–67, 2021.

[17] Benjamin Bowman, Craig Laprade, Yuede Ji, and H Howie Huang. Detecting lateral movement in enterprise computer networks with unsupervised graph {AI}. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses ({RAID} 2020)*, pages 257–268, 2020.

[18] Markus M Breunig, Hans-Peter Kriegel, Raymond T Ng, and Jörg Sander. Lof: identifying density-based local outliers. In *ACM sigmod record*, volume 29, pages 93–104. ACM, 2000.

[19] byt3bl33d3r. Deathstar. https://github.com/byt3bl33d3r/DeathStar. Accessed:2021-01-16.

[20] R. Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *IEEE Transactions on Software Engineering*, 34(5):579–596, Sept 2008.

[21] Qian Chen and Robert A Bridges. Automated behavioral analysis of malware: A case study of wannacry ransomware. In *2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 454–460. IEEE, 2017.

[22] Benjamin Delpy. mimikatz. https://github.com/gentilkiwi/mimikatz. Accessed:2021-01-16.

[23] Mengnan Du, Ninghao Liu, Qingquan Song, and Xia Hu. Towards explanation of dnn-based prediction with guided feature inversion. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1358–1367, 2018.

[24] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. Deeplog: Anomaly detection and diagnosis from system logs through deep learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, 2017.

[25] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 60–71, Piscataway, NJ, USA, 2019. IEEE Press.

[26] Elastic. Elastic. https://www.elastic.co/. Accessed:2021-01-16.

[27] A. Fawaz, A. Bohara, C. Cheh, and W. H. Sanders. Lateral movement detection using distributed data fusion. In *2016 IEEE 35th Symposium on Reliable Distributed Systems (SRDS)*, pages 21–30, Sep. 2016.

[28] FireEye. How many alerts is too many to handle?

[29] FireEye. M-trends 2019. https://content.fireeye.com/m-trends/rpt-m-trends-2019, 2019.

[30] FireEye. Highly evasive attacker leverages solarwinds supply chain to compromise multiple global victims with sunburst backdoor. https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html, 2020. Accessed:2021-01-16.

[31] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 120–128, May 1996.

[32] Peng Gao, Xusheng Xiao, Ding Li, Zhichun Li, Kangkook Jee, Zhenyu Wu, Chung Hwan Kim, Sanjeev R Kulkarni, and Prateek Mittal. Saql: A stream-based query system for real-time abnormal system behavior detection. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 639–656, 2018.

[33] Peng Gao, Xusheng Xiao, Zhichun Li, Fengyuan Xu, Sanjeev R Kulkarni, and Prateek Mittal. Aiql: Enabling efficient attack investigation from system monitoring data. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 113–126, 2018.

[34] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864. ACM, 2016.

[35] Michael Gutmann and Aapo Hyvärinen. Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 297–304, 2010.

[36] Aric Hagberg, Pieter Swart, and Daniel S Chult. Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[37] William L Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *arXiv preprint arXiv:1709.05584*, 2017.

[38] Wajih Ul Hassan, Lemay Aguse, Nuraini Aguse, Adam Bates, and Thomas Moyer. Towards scalable cluster auditing through grammatical inference over provenance graphs. In *Network and Distributed Systems Security Symposium*, 2018.

[39] Wajih Ul Hassan, Adam Bates, and Daniel Marino. Tactical provenance analysis for endpoint detection and response systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1172–1189. IEEE, 2020.

[40] Wajih Ul Hassan, Shengjian Guo, Ding Li, Zhengzhang Chen, Kangkook Jee, Zhichun Li, and Adam Bates. Nodoze: Combatting threat alert fatigue with automated provenance triage. In *Network and Distributed Systems Security Symposium*, 2019.

[41] Cheng-Yuan Ho, Yuan-Cheng Lai, I-Wei Chen, Fu-Yu Wang, and Wei-Hsuan Tai. Statistical analysis of false positives and false negatives from real traffic with intrusion detection/prevention systems. *IEEE Communications Magazine*, 50(3):146–154, 2012.

[42] Md Nahid Hossain, Sadegh M Milajerdi, Junao Wang, Birhanu Eshete, Rigel Gjomemo, R Sekar, Scott Stoller, and VN Venkatakrishnan. Sleuth: Real-time attack scenario reconstruction from cots audit data. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 487–504, 2017.

[43] Yang Hu, Hang Liu, and H Howie Huang. Tricore: Parallel triangle counting on gpus. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 171–182. IEEE, 2018.

[44] SANS Institute. Sans 2020 threat hunting survey results. Technical report, 2021.

[45] Md Rakibul Islam, Minhaz F. Zibran, and Aayush Nagpal. Security vulnerabilities in categories of clones and non-cloned code: An empirical study. In *Proceedings of the 11th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '17, pages 20–29, Piscataway, NJ, USA, 2017. IEEE Press.

[46] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, SP '12, pages 48–62, Washington, DC, USA, 2012. IEEE Computer Society.

[47] Yuede Ji, Lei Cui, and H Howie Huang. Buggraph: Differentiating source-binary code similarity with graph triplet-loss network. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 702–715, 2021.

[48] Yuede Ji, Hang Liu, and H Howie Huang. ispan: Parallel identification of strongly connected components with spanning trees. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 731–742. IEEE, 2018.

[49] Yuede Ji, Hang Liu, and H Howie Huang. Swarmgraph: Analyzing large-scale in-memory graphs on gpus. In *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 52–59. IEEE, 2020.

[50] Yan Jia, Yulu Qi, Huaijun Shang, Rong Jiang, and Aiping Li. A practical approach to constructing a knowledge graph for cybersecurity. *Engineering*, 4(1):53–60, 2018.

[51] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105, Washington, DC, USA, 2007. IEEE Computer Society.

[52] John R Johnson and Emilie A Hogan. A graph analytic metric for mitigating advanced persistent threat. In *2013 IEEE International Conference on Intelligence and Security Informatics*, pages 129–133. IEEE, 2013.

[53] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering*, 28(7):654–670, Jul 2002.

[54] Alexander D. Kent. Comprehensive, Multi-Source Cyber-Security Events. Los Alamos National Laboratory, 2015.

[55] Alexander D. Kent. Comprehensive, Multi-Source Cyber-Security Events. Los Alamos National Laboratory, 2015.

[56] Amin Kharaz, Sajjad Arshad, Collin Mulliner, William Robertson, and Engin Kirda. Unveil: A large-scale, automated approach to detecting ransomware. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 757–772, 2016.

[57] S. Kim, S. Woo, H. Lee, and H. Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 595–614, May 2017.

[58] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[59] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[60] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. Ddos in the iot: Mirai and other botnets. *Computer*, 50(7):80–84, 2017.

[61] Christopher Kruegel and Giovanni Vigna. Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 251–261. ACM, 2003.

[62] Pradeep Kumar and H Howie Huang. G-store: high-performance graph store for trillion-edge processing. In *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 830–841. IEEE, 2016.

[63] Pradeep Kumar and H Howie Huang. Graphone: A data store for real-time analytics on evolving graphs. *ACM Transactions on Storage (TOS)*, 15(4):1–40, 2020.

[64] Wenke Lee, Salvatore J Stolfo, and Kui W Mok. A data mining framework for building intrusion detection models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No. 99CB36344)*, pages 120–132. IEEE, 1999.

[65] Joshua Lewis. Incident identification through outlier analysis. Technical report, SANS, 2016.

[66] Z. Li, S. Lu, S. Myagmar, and Y. Zhou. Cp-miner: finding copy-paste and related bugs in large-scale software code. *IEEE Transactions on Software Engineering*, 32(3):176–192, March 2006.

[67] Zhen Li, Deqing Zou, Shouhuai Xu, Hai Jin, Hanchao Qi, and Jie Hu. Vulpecker: an automated vulnerability detection system based on code similarity analysis. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 201–213. ACM, 2016.

[68] Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. Vuldeepecker: A deep learning-based system for vulnerability detection. In *Proceedings of the 2018 Network and Distributed System Security Symposium (NDSS)*, 2018.

[69] Linux. auditd. https://man7.org/linux/man-pages/man8/auditd.8.html. Accessed:2021-01-16.

[70] Fei Tony Liu, Kai Ming Ting, and Zhi-Hua Zhou. Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE, 2008.

[71] Hang Liu and H Howie Huang. Enterprise: breadth-first graph traversal on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[72] Hang Liu and H Howie Huang. Graphene: Fine-grained {IO} management for graph computing. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 285–300, 2017.

[73] Hang Liu, H Howie Huang, and Yang Hu. ibfs: Concurrent breadth-first search on gpus. In *Proceedings of the 2016 International Conference on Management of Data*, pages 403–416, 2016.

[74] Yushan Liu, Mu Zhang, Ding Li, Kangkook Jee, Zhichun Li, Zhenyu Wu, Junghwan Rhee, and Prateek Mittal. Towards a timely causality analysis for enterprise security. In *NDSS*, 2018.

[75] Zhen Liu, Qiang Wei, and Yan Cao. Vfdetect: A vulnerable code clone detection system based on vulnerability fingerprint. In *2017 IEEE 3rd Information Technology and Mechatronics Engineering Conference (ITOEC)*, pages 548–553. IEEE, 2017.

[76] Nathan Manworren, Joshua Letwat, and Olivia Daily. Why you should care about the target data breach. *Business Horizons*, 59(3):257–266, 2016.

[77] Emaad Manzoor, Sadegh M Milajerdi, and Leman Akoglu. Fast memory-efficient anomaly detection in streaming heterogeneous graphs. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1035–1044. ACM, 2016.

[78] Lockheed Martin. The cyber kill chain. https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html, 2020. Accessed: 2020-01-16.

[79] Microsoft. Overview of endpoint detection and response. https://docs.microsoft.com/en-us/microsoft-365/security/defender-endpoint/overview-endpoint-detection-response. Accessed:2021-01-16.

[80] Microsoft. sysmon. https://docs.microsoft.com/en-us/sysinternals/downloads/sysmon. Accessed:2021-01-16.

[81] Microsoft. Windows events. https://docs.microsoft.com/en-us/windows/win32/events/windows-events. Accessed:2021-01-16.

[82] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[83] Sadegh M. Milajerdi, Birhanu Eshete, Rigel Gjomemo, and V.N. Venkatakrishnan. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1795–1812, New York, NY, USA, 2019. Association for Computing Machinery.

[84] Sadegh M Milajerdi, Birhanu Eshete, Rigel Gjomemo, and VN Venkatakrishnan. Poirot: Aligning attack behavior with kernel audit records for cyber threat hunting. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1795–1812, 2019.

[85] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, R Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1137–1152. IEEE, 2019.

[86] Sadegh M Milajerdi, Rigel Gjomemo, Birhanu Eshete, Ramachandran Sekar, and VN Venkatakrishnan. Holmes: real-time apt detection through correlation of suspicious information flows. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1137–1152. IEEE, 2019.

[87] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: an ensemble of autoencoders for online network intrusion detection. *arXiv preprint arXiv:1802.09089*, 2018.

[88] MITRE. Mitre att@ck. https://attack.mitre.org/, 2020.

[89] Steven Noel, Eric Harley, Kam Him Tam, Michael Limiero, and Matthew Share. Cygraph: graph-based analytics and visualization for cybersecurity. In *Handbook of Statistics*, volume 35, pages 117–167. Elsevier, 2016.

[90] United States Government Accountability Office. Actions taken by equifax and federal agencies in response to the 2017 breach. 2018.

[91] Alina Oprea, Zhou Li, Ting-Fang Yen, Sang H Chin, and Sumayah Alrwais. Detection of early-stage enterprise infection by mining large-scale log data. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 45–56. IEEE, 2015.

[92] OpTC. Darpa. https://github.com/FiveDirections/OpTC-data, 2021. Accessed: 2021-07-21.

[93] OSSEC. Ossec. https://www.ossec.net/, 2021. Accessed:2021-01-16.

[94] Bryan Perozzi, Rami Al-Rfou, and Steven Skiena. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 701–710. ACM, 2014.

[95] Tadeusz Pietraszek. Using adaptive alert classification to reduce false positives in intrusion detection. In *International workshop on recent advances in intrusion detection*, pages 102–124. Springer, 2004.

[96] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.

[97] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 1157–1168. IEEE, 2016.

[98] R Sekar, Mugdha Bendre, Dinakar Dhurjati, and Pradeep Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 144–155. IEEE, 2000.

[99] Lwin Khin Shar, Lionel C Briand, and Hee Beng Kuan Tan. Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Transactions on Dependable and Secure Computing*, 12(6):688–707, 2015.

[100] Xiaokui Shu, Ke Tian, Andrew Ciambrone, and Danfeng Yao. Breaking the target: An analysis of target data breach and lessons learned. *arXiv preprint arXiv:1701.04940*, 2017.

[101] Xiaokui Shu, Danfeng Yao, and Naren Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 401–413. ACM, 2015.

[102] Snort. Snort. https://www.snort.org/, 2020. Accessed:2020-01-16.

[103] Georgios P Spathoulas and Sokratis K Katsikas. Reducing false positives in intrusion detection systems. *computers & security*, 29(1):35–44, 2010.

[104] Splunk. Splunk. https://www.splunk.com, 2020.

[105] Suricata. Suricata. https://suricata.io/, 2021. Accessed:2021-01-16.

[106] Jian Tang, Meng Qu, Mingzhe Wang, Ming Zhang, Jun Yan, and Qiaozhu Mei. Line: Large-scale information network embedding. In *Proceedings of the 24th international conference on world wide web*, pages 1067–1077. International World Wide Web Conferences Steering Committee, 2015.

[107] Tenable. Nessus. https://www.tenable.com/products/nessus.

[108] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[109] VirusTotal. Virustotal. https://www.virustotal.com/. Accessed:2021-01-16.

[110] David A Wheeler. Flawfinder, 2011.

[111] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.

[112] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck. Automatic inference of search patterns for taint-style vulnerabilities. In *2015 IEEE Symposium on Security and Privacy*, pages 797–812, May 2015.

[113] Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Conference on Offensive Technologies*, WOOT'11, pages 13–13, Berkeley, CA, USA, 2011. USENIX Association.

[114] Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 359–368, New York, NY, USA, 2012. ACM.

[115] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*, CCS '13, pages 499–510, New York, NY, USA, 2013. ACM.

[116] Runqing Yang, Shiqing Ma, Haitao Xu, Xiangyu Zhang, and Yan Chen. Uiscope: Accurate, instrumentation-free, and visible attack investigation for gui applications. In *NDSS*, 2020.

[117] Rex Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. Gnn explainer: A tool for post-hoc explanation of graph neural networks. *arXiv preprint arXiv:1903.03894*, 2019.

[118] Le Yu, Shiqing Ma, Zhuo Zhang, Guanhong Tao, Xiangyu Zhang, Dongyan Xu, Vincent E Urias, Han Wei Lin, Gabriela Ciocarlie, Vinod Yegneswaran, et al. Alchemist: Fusing application and audit logs for precise attack provenance without instrumentation. 2021.

[119] Xiao Yu, Xiang Ren, Yizhou Sun, Quanquan Gu, Bradley Sturt, Urvashi Khandelwal, Brandon Norick, and Jiawei Han. Personalized entity recommendation: A heterogeneous information network approach. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 283–292, 2014.

[120] Zeek. The zeek network security monitor. https://zeek.org, 2020. Accessed: 2020-01-16.

[121] Jun Zeng, Zheng Leong Chua, Yinfang Chen, Kaihang Ji, Zhenkai Liang, and Jian Mao. Watson: Abstracting behaviors from audit logs via aggregation of contextual semantics. In *Network and Distributed Systems Security Symposium (NDSS)*, 2021.