STATIC ANALYSIS TOOL DISCREPANCIES AND THE PURSUIT OF A UNIFIED VULNERABILITY DATABASE

by

Brittany Morgan Boles

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Cybersecurity

MONTANA STATE UNIVERSITY
Bozeman, Montana

May 2025

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my primary advisor, Dr. Ann Marie Reinhold, for her invaluable mentorship, support, and for teaching me the essential skills of academic writing. I am also deeply thankful to my committee member, Dr. Clemente Izurieta, for his insightful guidance and support throughout my research. Additionally, I owe a great deal of my cybersecurity knowledge to Dr. Matthew Revelle, whose classes and mentorship have profoundly shaped my understanding of the field. I would also like to thank the members of the Montana State University Software Engineering and Cybersecurity Lab (MSU SECL) for their support both inside and outside the lab. Finally, I would like to express my gratitude to my partner, Russell Conti, for the many breakfasts that fueled me through this project.

This research was conducted with support from the U.S. Department of Homeland Security (DHS) Science and Technology Directorate (S&T) under contract 70RSAT22CB0000005. Any opinions contained herein are those of the author and do not necessarily reflect those of DHS ST.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

LIST OF TABLES

Table Page

LIST OF FIGURES

NOMENCLATURE

SAST	Static Application Security Testing
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
GHSA	Github Security Advisory
CVSS	Common Vulnerability Scoring System
EPSS	Exploit Prediction Scoring System
MSUSEL	Montana State University Software Engineering Lab
DHS	U.S. Department of Homeland Security
NVD	National Vulnerability Database
OSV	Open Source Vulnerabilities

ABSTRACT

When cyber attacks succeed, they affect communities, not just systems. To protect against these threats, organizations use static analysis tools to identify vulnerabilities in software components, especially within third-party dependencies. The effectiveness of these tools depends on their reliance on external vulnerability databases. The specific databases that researchers use and the way they aggregate data significantly influence the vulnerabilities they identify and report.

In this thesis, I investigate inconsistent vulnerability counts reported by two widely used static analysis tools, Trivy and Grype, across 927 Docker images. Trivy and Grype rely on different vulnerability databases. I show that differences in database selection and aggregation techniques lead to divergent vulnerability counts, classifications, and severity metrics. My findings emphasize the need for better interoperability and aggregation strategies in vulnerability management.

In response, I developed an integrated graph-based vulnerability database that integrates data from the National Vulnerability Database (NVD), GitHub Advisories, and the Open Source Vulnerability (OSV) database. By incorporating relationships such as aliases and related vulnerabilities, our graph database enables seamless cross-referencing, even across differing vulnerability identifiers. Further, our graph database includes EPSS scores and CWE mappings, offering additional security perspectives. This approach improves both coverage and collaboration, supporting informed security decisions.

Our analysis of the 2023 top ten most routinely exploited vulnerabilities demonstrates the graph database's practical value: while all ten were present in the NVD, OSV included only one, and GitHub Advisories required alias relationships for identification. Moreover, nine of the ten vulnerabilities shared linked Common Weakness Enumeration (CWE) entries, revealing exploitable patterns in vulnerability structures. This thesis advances the field by exposing inconsistencies in existing vulnerability tools that rely on disparate databases, and by introducing a scalable, unified system to enable more informed security decision-making.

1

INTRODUCTION

As critical systems grow increasingly reliant on digital technologies [1], a focused commitment to cybersecurity is essential to promote resilience against emerging threats [3]. Organizations and developers are ethically obligated to prioritize cybersecurity, recognizing that the vulnerabilities in their products can directly impact users' safety and privacy.

The number of reported software vulnerabilities increases annually [13]. As technology evolves, so do attack techniques [14], and organizations must consistently adapt, utilizing security tools to detect and manage threats effectively and at scale.

One of the most widely used techniques for detecting vulnerabilities is static analysis. It plays a crucial role in security by identifying potential weaknesses and vulnerabilities without executing the code, making it safe and efficient for early detection of problems [21]. Static analysis tools can detect common weaknesses in source code using static application security testing (SAST) [30] and can identify known vulnerabilities in a project's dependencies through vulnerability scanning.

This thesis focuses on vulnerability identification and aggregation. In contemporary software development, where code reuse is prevalent, third party software libraries often present significant risks [20, 32]. Static analysis tools identify known issues, helping developers make informed decisions about whether to patch, replace, or accept these risks. The security and reliability of systems are greatly enhanced through the use of static analysis tools.

Despite their utility, static analysis tools face significant challenges. False positives are common, and interfaces are often difficult to use [18, 19, 21, 30]. Tools can produce inconsistent results—even across versions of the same tool [22–25]. These inconsistencies

undermine trust and complicate security decision-making.

To better understand the discrepancies in vulnerability detection, we examined the differences in the count and type of vulnerabilities reported by two popular static analysis tools, Trivy and Grype, across 927 Docker images. We analyzed their vulnerability reports, comparing vulnerability counts, IDs, and severity classifications. Additionally, we investigated their underlying databases and aggregation processes to identify the causes of inconsistencies. Differences in how Trivy and Grype interact with vulnerability databases determine the vulnerabilities that each report.

Static analysis tools rely on vulnerability databases to compile and report known security flaws. However, differences in naming conventions, abstraction, and general maintenance contribute to a lack of interoperability between vulnerability databases. Further, vulnerability databases are prone to incompleteness, inaccuracies, and delays in reporting [26, 35]. To address these issues, we developed an integrated graph-based database that consolidates data from the Open Source Vulnerability (OSV)[1] database, GitHub Advisories[2], and the National Vulnerability Database (NVD)[3]. Additionally, I included the Exploit Prediction Scoring System (EPSS) database[4] and the CWE 1000 View[5], providing deeper insights into exploitability and common software weaknesses. This approach improves interoperability by enabling cross-referencing between different vulnerability taxonomies, enriches security assessments with additional context, and facilitates a more comprehensive understanding of software security risks.

I developed a graph database that unifies diverse vulnerability data sources to alleviate inconsistencies across static analysis tool reports and vulnerability databases. I demonstrate

[1]https://osv.dev/
[2]https://github.com/advisories
[3]https://nvd.nist.gov/
[4]https://www.first.org/epss/
[5]https://cwe.mitre.org/data/definitions/1000.html

how integrating multiple databases through a graph-based approach can enhance decision-making and improve security assessments. We have made our integrated vulnerability knowledge graph open-source to support further research and industry adoption.

4

QUESTIONS, CHALLENGES AND OBJECTIVES

We adopted Schlemiel's method in our research [27], structuring our investigation into questions, challenges, and objectives. We had a primary question for each chapter, along with its respective challenges and objectives.

Questions

Question 1: Why do different static analysis tools report different vulnerabilities when analyzing the same software artifacts?

Question 2: Can an integrated vulnerability graph database enhance vulnerability management?

Challenges and Objectives

Challenge 1: Different static analysis tools, such as Trivy and Grype, often report inconsistent sets of vulnerabilities for the same container images, raising questions about the sources and methods driving these discrepancies.

- Objective 1: To compare the vulnerability databases leveraged by Trivy and Grype.
- Objective 2: Analyze each tool's documentation and source code to understand how they retrieve, aggregate, and classify vulnerabilities.
- Objective 3: Systematically evaluate the differences in the vulnerabilities reported by Trivy and Grype in a large corpus of Docker images.

Challenge 2: Modern security requires relying on multiple vulnerability databases, but differences in schema, naming conventions, and abstraction making it difficult to unify and compare vulnerability information across sources.

- Objective 4: Determine the extent of overlap between Open Source Vulnerabilities (OSV), GitHub Advisories, and the National Vulnerability Database (NVD) by analyzing their shared and unique contributions.

- Objective 5: Evaluate consistency in the Common Vulnerability Scoring System (CVSS) scores by conducting a pairwise analysis of alias relationships across databases.

- Objective 6: Integrate EPSS scores and MITRE's CWE 1000 view to provide insights into the vulnerability impact and exploitability of a system.

- Objective 7: Use the graph database to investigate the ten most routinely exploited vulnerabilities from 2023.

DECIPHERING DISCREPANCIES: A COMPARATIVE ANALYSIS OF DOCKER IMAGE SECURITY

Contribution of Authors and Co-Authors

Manuscript in following chapter

Author: Brittany Boles

Contributions: Designed and implemented the study concept, performed data analysis, interpreted the results, and authored the manuscript.

Co-Author: Eric O'Donoghue

Contributions: Assisted with data collection, provided feedback and edited the manuscript.

Co-Author: A. Redempta Manzi Muneza

Contributions: Helped with early design of the study and provided feedback.

Co-Author: Garrett Perkins

Contributions: Provided feedback and helped author the manuscript.

Co-Author: Clemente Izurieta

Contributions: Obtained funding, provided feedback on the analyses, and edited the manuscript.

Co-Author: Ann Marie Reinhold

Contributions: Obtained funding, provided feedback on the analyses, and edited the manuscript.

Manuscript Information

Brittany Boles, Eric O'Donoghue, A. Redempta Manzi Muneza, Garrett Perkins,

Clemente Izurieta, Ann Marie Reinhold

2nd 2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM).

Status of Manuscript:
____ Prepared for submission to a peer-reviewed journal
____ Officially submitted to a peer-reviewed journal
____ Accepted by a peer-reviewed journal
_X_ Published in a peer-reviewed journal

Publisher: IEEE
DOI: 10.1109/SCAM63643.2024.00034

Abstract

As the use of microservices continues to grow and become a foundational approach to architecting software solutions, ensuring the security of microservices is paramount. Docker images have emerged as the predominant solution to containerize microservices–and thus, Docker images are becoming a large attack surface. Thus, reducing vulnerabilities in Docker images will reduce microservice cyberattacks. A common way to find vulnerabilities in Docker images employs static analysis tools like Trivy and Grype. However, these tools frequently generate disparate vulnerability reports when analyzing the same Docker image, thus causing uncertainty in tool selection. We collected 927 Docker images, analyzed them with Trivy and Grype, and compared the vulnerabilities reported in each image. Among the 865 images found to have vulnerabilities, Trivy and Grype disagreed on both the number of vulnerabilities and the vulnerability IDs found therein. Since both tools interface with external vulnerability databases, some discrepancies can be attributed to how the tools interface with these external resources. The external vulnerability databases partially overlap and frequently contradict one another, thereby creating challenges for static analysis tool developers and end users alike. This New Ideas and Emerging Results (NIER) study contains new and critical information that practitioners need for selecting and using static analysis tools–given that increases in the use of Docker technologies means increases in the size of the attack surfaces.

Introduction

Microservices have emerged as an improved alternative to monolithic architectures and are commonplace in contemporary software solutions. Microservices offer many benefits, such as increased modularity, flexible configuration, simplified development, easier maintenance, and heightened productivity [29]. Prominent companies, including Netflix, Amazon, and Uber, have embraced the adoption of microservice architectures. As the use of microservice architectures grow, so to does the containerization technology, which provides details for the microservices. Docker has emerged as a front-runner containerization technology of microservices with reportedly over 75,000 company customers[1].

Docker images are used to initialize Docker containers, which in turn realize microservice solutions. Docker images layer-based architecture alleviates challenges associated with setting up development environments, making Docker highly desired. According to a survey conducted in 2023, Docker was reported as the most widely used tool with most developers expressing their intention to continue its usage in 2024[2].

As the use of Docker and microservices become the industry standard consequently security risks have emerged. Vulnerabilities within Docker images can allow bad actors to implement cyber attacks. A vulnerability is defined by the National Vulnerability Database (NVD[3]) as "A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability." Vulnerability databases such as the NVD and GitHub advisories[4] store hundreds of thousands of known vulnerability details. Many of these vulnerabilities reside in Docker images, including Docker images in active use. One study found that

[1] `www.docker.com/trust/`
[2] `https://survey.stackoverflow.co/2023/`
[3] `https://nvd.nist.gov/vuln`
[4] `https://github.com/github/advisory-database`

Table 1: The different vulnerability databases that Trivy and Grype use. Trivy uses more vulnerability databases than Grype @IEEE 2024

Data Source	Grype	Trivy
NVD	x	x
Alpine	x	x
Amazon	x	x
Debian	x	x
GitHub Advisories	x	x
Oracle	x	x
Redhat	x	x
SUSE	x	x
Ubuntu CVE Tracker	x	x
Photon Security Advisory	-	x
Arch Linux	-	x
CBL Mariner	-	x
Node js Security	-	x
GitLab Advisories Community	-	x
AlmaLinux	-	x
RubySec	-	x
PhP Security	-	x
Rocky Linux	-	x

community and official Docker images had more than 180 vulnerabilities on average, with many of the vulnerabilities being persistent between versions [28]. These findings of widespread vulnerabilities in Docker images brings light to just how large the attack surface is.

As more developers adopt Docker technology, limiting vulnerabilities is critical. Static analysis of Docker images is a common strategy for assessing cybersecurity risks [7]. Static analysis tools report known vulnerabilities within Docker image packages and artifacts. The benefits of static analysis tools include their ability to scan software for vulnerabilities without executing the software, and fast processing speeds [2]. However, static analysis tools are not without fault; these tools only report known vulnerabilities and produce a concerning number of false positives—reducing trust [9]. Thus, end users face the challenge of picking the tool that fits their needs and has trustworthy results.

Two popular static analysis tools for Docker images are Trivy[5] and Grype[6]. These open-

[5]`https://github.com/aquasecurity/trivy`
[6]`https://github.com/anchore/grype`

source static analysis tools have become an industry standard for analyzing Docker images. Both tools leverage vulnerability databases to report vulnerabilities in Docker images, software bill of materials (SBOMs), and file systems. Trivy and Grype purportedly provide the same information: reports of all known vulnerabilities in a software artifact. Despite these similarities, we found they frequently yield different results [24][23]. For instance, in a corpus of 1,151 Software Bills of Materials, Trivy reported 309,022 vulnerabilities whereas Grype reported 43,553 vulnerabilities [23]. Yet, the reasons why Trivy and Grype produce such different reports has never been investigated systematically. The absence of a systematic evaluation is problematic. Without a systematic evaluation of each tool and comparison of their results, end users of these tools are faced with numerous uncertainties. *Which tool should I trust? Why are the results so different?* These uncertainties are impediments for researchers and practitioners who rely on static analysis tools for assessing the security of Docker images. Here, we evaluate and compare the results of Trivy and Grype on a common set of targets and investigate the reasons underpinning the differences.

Our study addresses the following research goal: to systematically evaluate the differences in the vulnerabilities reported by Trivy and Grype in a large corpus of Docker images.

Background

Trivy and Grype rely on external databases to give them trustworthy vulnerability information. Transitively, the users of Trivy and Grype are also reliant on these databases. These databases contain important information about each vulnerability that is codified using a vulnerability ID. Each entry also contains metadata for each vulnerability; examples of metadata include the severity of the vulnerability and where the vulnerability is found. Knowing which set of external databases each tool uses and how each tool aggregates all vulnerability information from each database is critical.

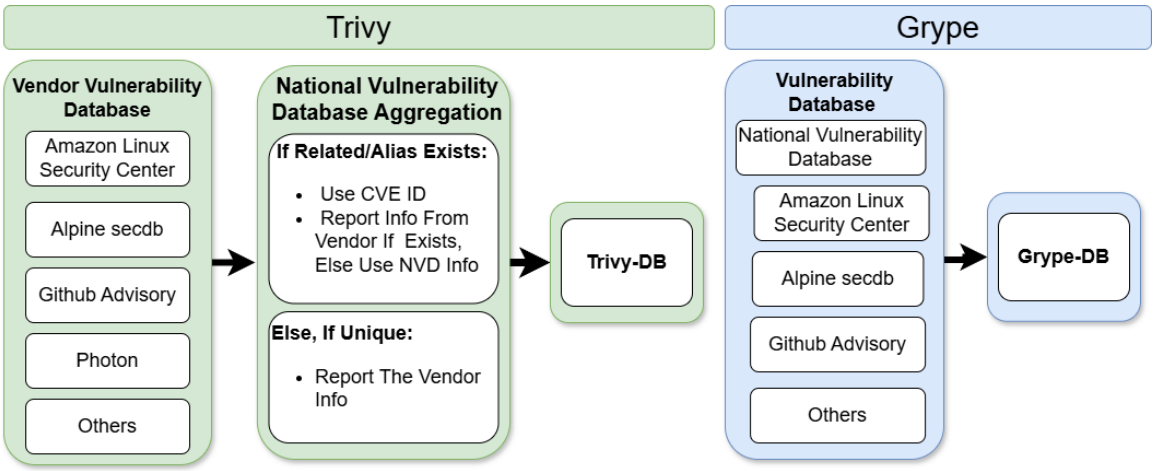

Figure 1: Schematic representing the process Grype (blue) and Trivy (green) use to create the Grype-DB and Trivy-DB, respectively. Both tools use the NVD but in different ways. Trivy aggregates information between databases into one source whereas Grype keeps the database information separate. Figure was made with draw.io. @IEEE 2024

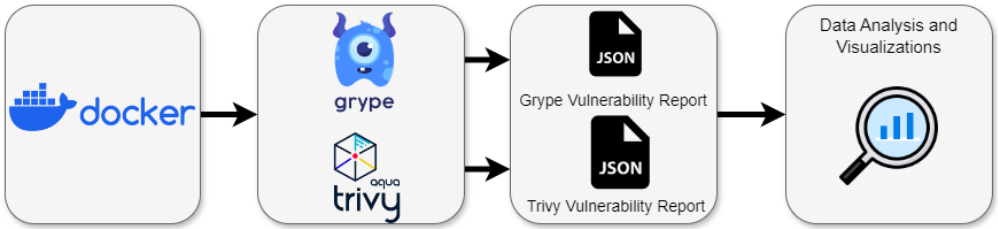

Figure 2: Process Diagram of the data pipeline. Docker images are pulled; then desired versions of Trivy and Grype are downloaded. Each image is run through each Trivy and Grype and the results are aggregated. Finally, those results are analyzed and data visuals of the reported vulnerabilities are built. Figure was made with draw.io. @IEEE 2024

External Databases

Trivy and Grype pull vulnerability details from external databases to create Trivy-DB and Grype-DB, their respective internal vulnerability databases (Fig. 1, Table 1). Trivy and Grype use many of the same external databases, including the largest, most widely used databases, GitHub Advisories, with 211,900 vulnerabilities, and the NVD, with 243,544. Though the tools share many external databases, Trivy uses nine additional databases, referred to as 'vendor vulnerability databases'.

The databases often use their own labeling conventions for vulnerabilities, creating

unique ID types. Common Vulnerabilities and Exposures (CVE) IDs are the most used ID type for vulnerabilities. CVEs are assigned by CVE-numbering authorities, and enriched by the NVD. Other vulnerability labels include GitHub Security Advisories (GHSA) IDs or Amazon Linux AMI Security Advisory (ALAS) IDs, created by GitHub Advisories[7] and Amazon[8] respectively. When two databases contain the same vulnerability, they are referred to as a 'related' vulnerability. These related vulnerabilities can either share the same ID or be labeled with unique IDs. Taking the infamous Log4j vulnerability for example, the NVD labels it CVE-2021-44228, whereas the GitHub advisories label it GHSA-jfh8-c2jp-5v3q. However, it is the same underlying vulnerability. Thus, related vulnerabilities are the same issue documented in different databases.

Internal Aggregation of Vulnerabilities

Developers of Trivy and Grype face a formidable challenge. They need to aggregate vulnerability information from multiple databases- some overlapping or unique to each database- in the creation of their internal vulnerability databases. Both tools compile lists of all the Docker image components, imported packages, operating systems, etc., all with specific versions. The tools look for matches between the components of each Docker image and known vulnerabilities contained in each tool's respective internal database.

Trivy aggregates information from the NVD and the other databases to create Trivy-DB[9] as follows. Trivy combines vulnerabilities related to entries in the NVD into one local source, i.e., Trivy-DB. Trivy uses the CVE labeling from the NVD but uses the vendor database information to populate other data fields such as severity. This relabeling process means most entries in Trivy-DB will have CVE IDs.

The aggregation process for Grype differs than for Trivy. Grype combines the

[7] `https://github.com/advisories`
[8] `https://alas.aws.amazon.com/`
[9] `https://github.com/aquasecurity/trivy-db`

vulnerability databases listed in Table 1 to create Grype-DB[10]. Each vulnerability ID in the set of external database becomes an entry in Grype-DB. Grype-DB contains a related vulnerability field that informs users when a reported ID has a related vulnerability in the NVD. The related vulnerability field is *only* populated when vendor databases or GitHub Advisories have a related vulnerability in the NVD. These relationships are either provided by the databases, or are obvious because the related IDs are identical to the CVE ID. In the case of a vendor ID being related to multiple vulnerabilities in the NVD Grype fills the related vulnerability field with multiple vulnerability IDs.

The NVD presents vulnerability information using common platform enumeration (CPE)[11]. Historically, Grype used CPE to match components of Docker images to known vulnerabilities in the NVD. In an effort to reduce the reporting of false positives, Grype now limits the use of CPE matching[12]. Grype relies on other databases like Github Advisories, and states some ecosystems could see up to an 80% reduction in false positives with this change.

Methods

We began by creating a corpus of official Docker images (Fig. 2). We built a data pipeline in Python (https://doi.org/10.5281/zenodo.13380592). We evaluated ten versions (evenly spaced through all the available versions) of the top 97 most pulled Docker Official images[13] on Docker Hub as of February 1, 2024. Only versions compatible with a Linux system were included. If fewer than ten versions of a Docker image were released, we used all available versions.

We selected Trivy and Grype because of their popularity. As of August 2024, the

[10] `https://github.com/anchore/grype-db`
[11] `https://cpe.mitre.org/about/index.html`
[12] `https://anchore.com/blog/say-goodbye-to-false-positives/`
[13] `https://hub.docker.com/search?image_filter=official`

GitHub repository for Trivy had been forked 2,200 times and Grype had been forked 542 times. In addition, Trivy and Grype were recommended by SECL (`https://www.montana.edu/cyber/`) industry and government subject matter experts. We selected the most recent versions of Grype (v0.73.0) and Trivy (v0.49.0) for tool comparison on February 1, 2024. With Trivy, we used the configuration "-timeout 30m" because several images failed with an "analyze error: timeout: context deadline exceeded". We found a timeout of 30 minutes was more than satisfactory to prevent this error.

We ran every Docker image in our collection through Trivy and Grype. Some Docker images could not be analyzed by both tools. For instance, Grype returned empty JSON files for all versions of the Docker image *Mono* as well as the images alpine:3.17.1, alpine3.18.5, and alpine 3.18.2, so these images were removed from our corpus. Trivy did not process *golang:1.4rc1*, and we removed this image from the corpus.

We implemented two primary controls for repeatability and validity. First, we downloaded the tools' databases on November 11, 2023. Thus, both tools used static databases to avoid updates throughout the study. The databases used for both tools are available at the DOI provided above. Second, both tools analyzed the exact same corpus of Docker images.

The reports from Trivy and Grype are depicted in the third step of our process diagram (Fig. 2). In the fourth step, we processed the tool reports and analyzed them.

We computed the difference in counts reported by Grype and Trivy in the Docker images. We visualized the difference in distributions with a density plot (Fig. 3). We also calculated descriptive statistics, i.e., the average count of vulnerabilities in each image and the associated standard deviations.

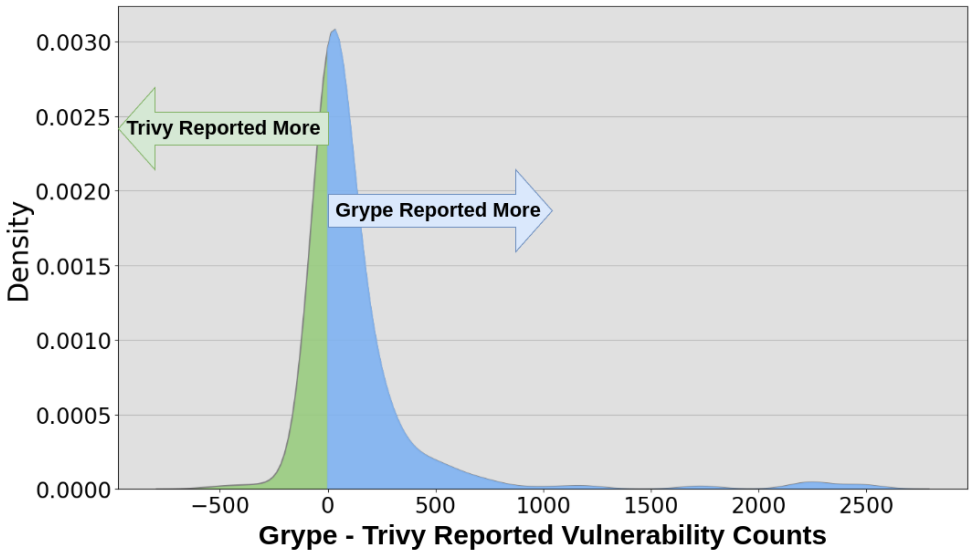

Figure 3: A density plot of the differences in Trivy and Grypes' reported vulnerabilities per image. The x-axis represents the difference between Trivy and Grypes' total count for each image. Both tools were analyzed over the corpus of 927 Docker images. @IEEE 2024

Results

Results from Trivy and Grype rarely agreed (Table 2; Fig. 3). Grype found more vulnerabilities than Trivy in 84.6% of the Docker images. Over the 927 images in the corpus, Trivy found 473,661 vulnerabilities, whereas Grype found 603,259.

Of the 603,259 vulnerabilities Grype reported in the corpus, 577,307 had a related vulnerability in the NVD. With respect to these related vulnerabilities in the NVD, Grype only double-counted vulnerabilities 675 times. These instances of double-counting are indicative of redundant reports for the same issue. However, there were relatively few instances. Thus, double-counting is not a primary reason that Grype generally reported more vulnerabilities than Trivy.

The magnitude of the differences between the tool findings was often large. On average, Trivy reported ∼140 fewer vulnerabilities than Grype (S.D. = 357.0). Furthermore, in 6.7%

of the images, Trivy and Grype reported differences exceeding 500 vulnerabilities. The largest difference was found in image python: 3.7.6-stretch, with a difference of 2,516 vulnerabilities. That is, Trivy found 3,208 vulnerabilities, whereas Grype found 5,724.

Trivy and Grype only agreed on the number of vulnerabilities in 15.3% of the images (142 of 927 images). In 62 of these 142 images, Trivy and Grype found zero vulnerabilities. Of the remaining 865 images found to contain vulnerabilities, Trivy and Grype found the same number of findings in 9.2% of those images. That is, only 80 images were found to have the same number of non-zero findings. However, even when Trivy and Grype reported the same number of vulnerabilities, the identities of the vulnerabilities (e.g., CVE IDs) *never* agreed. The only instance where the tools were unanimous in their findings was when they both found nothing.

We also observe differences in the types of IDs reported (Table 2). The external vulnerability databases have different labeling schemes, such as GHSA IDs from GitHub Advisories or CVE IDs from the NVD. Different counts of label types reported by Trivy and Grype arise because they do not use the same set of databases, and they use those databases differently (Fig. 1 and Table 1). Consequently, Trivy reported ID types Debian Linux Advisory (DLA), Debian Security Advisory (DSA), and Node.js Security Working Group (NSWG), all of which Grype never reported in our corpus. Conversely, Grype reported ID types Enterprise Linux Security Advisor (ELSA) and ALAS, which Trivy never reported. Trivy also reported a greater percentage of vulnerabilities as CVEs compared to Grype (Trivy = 98.5%, n_{Trivy} = 466,592; Grype = 95.1%, n_{Grype} = 573,953 vulnerabilities).

The discrepancies between Trivy and Grype extended to the metadata associated with vulnerabilities. The severity of the vulnerabilities is an important example of this discord. While we did not systematically evaluate severities, we found extensive anecdotal evidence of differences between the tool reports. In total, the tools disagreed 60,799 times on the severity of vulnerability IDs that were identical. Taking CVE-2019-17594 as an example, Grype

Table 2: Counts of the vulnerability IDs reported by Trivy and Grype. The 'Other' column contains DLA, DSA and NSWG. @IEEE 2024

Tool	CVE	GHSA	ALAS	ELSA	Other	Total
Trivy	466,592	965	0	0	6104	437,661
Grype	573,953	26,028	1,462	1,816	0	603,259

reported a 'medium' severity for this CVE, whereas Trivy reported a 'low' severity. We even found instances (e.g., CVE-2019-8457) where Grype labeled a vulnerability as 'negligible' whereas Trivy reported the same vulnerability as 'critical'.

Discussion

Trivy and Grype disagreed on the count, IDs, and severity of the vulnerabilities in the corpus, begging the question, *What is causing these differences?* Here, we unpack the reasons for the differences in the vulnerabilities reported by Trivy and Grype. We consider how the tools interact with external databases (see Subsection 3) and how they aggregate information from those databases internally (see Subsection 1).

Different Sets of External Vulnerability Databases

The external databases used by Trivy and Grype impact the IDs reported by the tools (Table 2). However, differences in external databases are not primary factors driving the differences in vulnerability counts. Evidence for this observation includes that Trivy found fewer vulnerabilities despite pulling information from nine more external databases than Grype. This observation was surprising because we expected having more information would result in finding more vulnerabilities.

In contrast, the differences in the sets of external databases used to create Trivy-DB and Grype-DB lead to different types of IDs being reported. Trivy, for instance, reports

NSWG vulnerability IDs sourced from Node js Security[14], a database not utilized by Grype (Table 2). On the other hand, Grype reports the ID type ELSA, which Trivy does not. The documentation from Trivy indicates that ELSA IDs were likely relabelled to their related CVE IDs, causing Trivy to never report ELSA IDs. This data aggregation leads to the high percentage of CVE IDs reported by Trivy. Thus, a primary driver of the differences reported by Trivy and Grype is *how the tools aggregate information from external databases* to create their internal vulnerability databases.

Different Internal Aggregation Processes

How the tools aggregated information from the external vulnerability databases, specifically handling related vulnerabilities, greatly impacted their results. Their distinct processes affected vulnerability IDs reported, their severities, and the total number of vulnerabilities reported.

Grype reported 95.7% vulnerabilities had a related vulnerability ID in the NVD, illustrating the extent to which the vendor vulnerability databases overlap with the NVD. Further, external vulnerability databases may also overlap with each other. We speculate that Grype reported more vulnerabilities (Fig. 3) because of how they handles related IDs.

Further research is needed to understand if Grype is reporting multiple IDs for the same underlying vulnerability and the degree to which redundancies may impact overall count. Validation into the aggregation processes used by Trivy is also needed; it remains unclear how Trivy handles multiple related vulnerabilities. While the aggregation process used by Grype can cause redundancies, it is transparent.

Vulnerability databases often disagree on the severity of vulnerabilities [12]. Disagreement between the databases caused Trivy and Grype to report different severities for the same vulnerability IDs. This disagreement arose because the tools can pull from different

[14] `https://github.com/nodejs/security-wg`

databases to populate metadata for the same vulnerability ID (Fig. 1).

Disagreement in severities is particularly problematic. End users of Trivy and Grype often filter the reported vulnerabilities by severity to tackle the most severe issues first. Thus, disagreement in severities creates a problem for the developers of Trivy and Grype, as well as for their end users who rely on these tools for vulnerability assessment.

It is also important to note that many vulnerability databases contain inaccuracies and duplications [11]. These inaccuracies and duplications present an additional source of uncertainty. Determining which source of vulnerability information static analysis tools should use–and how that information should be aggregated–is challenging when databases disagree.

NIER Considerations & Challenges

While we found variation in the vulnerability reports produced by Trivy and Grype, this research does not imply that these tools should not be used. Instead, we highlight challenges associated with their use–in particular, the challenges that arise from discrepancies between the tools and inaccuracies in their underlying vulnerability databases. Ultimately, if Trivy and Grype are pulling inaccurate data from the vulnerability databases, their results will also be inaccurate.

Some research shows that NVD has robust and trustworthy severity ratings [17]. Using the NVD is mandated for federal contractors to use as an authoritative source of threats by many federal government programs such as the Federal Risk and Authorization Management[15] program. However, the NVD is far from perfect.

While both Trivy and Grype use the NVD, these tools favor information from vendor databases and GitHub Advisories. Grype limits matching with the NVD. Trivy specifies choosing vendor severity ratings over the NVD. Trivy documentation states that vendor

[15] `www.fedramp.gov/2024-02-16-rev-5-additional-documents-released/`

severity ratings are more accurate than the NVD[16] and Red Hat agrees [17]. We also suspect that the developers of Trivy and Grype minimize the use of the NVD because of notoriously long lag times and pervasive inaccuracies [4]. These lags and inaccuracies can contribute to the tools reporting false positives, one of the most significant complaints of end users of static analysis tools [30]. Lags and inaccuracies are not unique to the NVD. However, the impact of these flaws is exacerbated by the NVD's prominence, large size, and processing procedure. A comprehensive database loses value if it cannot be trusted.

Each vulnerability database compiles, updates, and validates a rapidly increasing number of vulnerabilities. However, keeping pace with the volume and velocity of this data is a formidable challenge. Simplifying and aggregating the information across these databases is crucial, as manually gathering and verifying vulnerability information is both time-consuming and tedious. Our current work takes aim at surmounting these challenges directly.

We are currently consolidating the vulnerability information in these diverse databases into a unified, searchable graph database that links shared vulnerabilities. Our graph database will (1) streamline the vulnerability detection process by facilitating collaboration among database creators/maintainers, (2) aid static analysis developers by presenting comprehensive information in aggregate, and (3) enable end users and static analysis tool developers to make informed decisions when sources conflict. Our goal is to make the secure choice, the easy choice.

Threats to Validity

Our study was highly controlled – using stringent data processing steps to ensure replicability. Moreover, our study design minimizes threats to internal validity. However,

[16] `https://aquasecurity.github.io/trivy/v0.49/docs/scanner/vulnerability/`
[17] `www.redhat.com/en/blog/security-flaws-and-cvss-rescore-process-nvd`

we examine three other potential threats to validity: construct validity, content validity, and external validity using the classification scheme of Cook et al. [10] and Campbell et al. [8].

We consider potential threats to the construct and content validity of our study. In particular, our study assumes that a single version of Trivy and Grype produces a meaningful and comprehensive assessment of the vulnerabilities in each Docker image in the corpus. However, other studies have found that different versions of the same static analysis tool can produce different results even when holding the corpus of software artifacts constant [24, 25]. Although this "version variation" is known to exist for Trivy and Grype [24], Trivy and Grype have proven to produce more consistent and reliable results across versions than some binary analysis tools [25]. Thus, we acknowledge these potential threats but do not perceive them to undermine the overall conclusions of our study. Moreover, our systematic design can easily be extended to include multiple versions of Trivy and Grype.

The external validity of our study primarily hinges on the breadth of the corpus of Docker images. We analyzed 10 version of the 97 most downloaded Official Docker images that run on Linux. Therefore, our study is representative of Official Docker images that run on Linux. Trivy and Grype may produce different results when analyzing Docker images that are not Official or that run on other operating systems. Thus, a modicum of caution is warranted when extrapolating these results beyond the scope of the corpus.

Conclusions & Future Directions

Our work here introduces several challenges. The disagreement between results from Trivy and Grype bring the following questions to the forefront: *How reliable and accurate are Trivy and Grype? Additionally, how can we improve the reliability and accuracy of the vulnerability databases on which these tools depend?* These questions point to critical research challenges. Surmounting these challenges will reduce the attack surfaces of microservices. The high number of vulnerabilities we found in the corpus of popular, Official Docker images

suggests that microservices are far from secure; this statement would hold true even if half of the reported vulnerabilities were false positives. Therefore, microservices are–and will continue to be–prime targets for malicious actors.

We need to come together as a community to discuss how to surmount these challenges. Moreover, we advocate for building connections among the creators and managers of vulnerability databases, the developers of cybersecurity static analysis tools, and the researchers and practitioners who depend on these databases and tools.

Acknowledgment

This research was conducted with support from the U.S. Department of Homeland Security (DHS) Science and Technology Directorate (S&T) under contract 70RSAT22CB0000005. Any opinions contained herein are those of the author and do not necessarily reflect those of DHS S&T. Thanks to Gabe Cowley, Sabrina Hendricks, Madie Munro, Russell Conti, Yvette Hastings, Zach Wadhams, Ashley Boles, Emma Sheppard, Tom McElroy, and the Anchore office hour team for their contributions to this work.

CONNECTING THE DOTS: AN INTEGRATED VULNERABILITY KNOWLEDGE GRAPH FOR SECURITY PRACTITIONERS

Contribution of Authors and Co-Authors

Manuscript in following chapter

Author: Brittany Boles

Contributions: Developed and implemented integrated graph database. Preformed data collection, analysis and interpretation of results, and wrote the manuscript

Co-Author: Clemente Izurieta

Contributions: Obtained funding, helped develop the research concept, provided feedback on the tool, and edited the manuscript.

Co-Author: Ann Marie Reinhold

Contributions: Obtained funding, helped develop the research concept, provided feedback on the tool, and edited the manuscript.

Manuscript Information

Brittany Boles, Clemente Izurieta, Ann Marie Reinhold

2024 IEEE Information Reuse and Integration for Data Science

Status of Manuscript:
____ Prepared for submission to a peer-reviewed journal
_X_ Officially submitted to a peer-reviewed journal
____ Accepted by a peer-reviewed journal
____ Published in a peer-reviewed journal

Publisher: IEEE
Submission Date: April 1st 2025

Abstract

Vulnerability databases are essential to cybersecurity, providing developers with critical information about software security flaws. However, inconsistencies among vulnerability databases pose challenges for integration. To address this, we created a graph database that consolidates data from the National Vulnerability Database (NVD), GitHub Advisories, the Open Source Vulnerability (OSV) database, the Exploit Prediction Scoring System (EPSS), and the CWE-1000 View. Our graph database revealed inconsistent vulnerability severity vectors across the databases. To illustrate the utility of our graph database, we investigated how the databases reported the "top ten" most routinely exploited vulnerabilities. Our analysis revealed differences in vulnerability identifiers, and the Common Weakness Enumeration (CWE) mappings of the top ten vulnerabilities. By aggregating vulnerability information from disparate sources, this graph database supports cross-validation, increases transparency, and enables efficient complex queries.

Introduction

Cybersecurity risks in code are enumerated as vulnerabilities. Vulnerability databases catalog known security flaws. Consequently, vulnerability databases have emerged as a cornerstone of cybersecurity.

Creators and maintainers of each vulnerability database face a Sisyphean challenge. New vulnerabilities are reported daily and must be vetted for accuracy. Vulnerability submissions increased 32% from 2023 to 2024, and are estimated to continue to rise[1]. The volume and velocity of new vulnerabilities being reported make maintaining the databases more difficult. Consequently, the reliability, accuracy, and completeness of the databases are impacted [11, 16].

In February 2024, the National Vulnerability Database (NVD) paused updates for several months as the National Institute of Standards and Technology (NIST) worked to address a growing backlog of submissions[2]. The sudden disruption left the cybersecurity community scrambling for alternative databases[3,4]. The incident highlighted the dangers of over-reliance on a single source and emphasized the necessity for a more robust, multi-database strategy in cybersecurity.

Security tools already embrace this multi-source philosophy. Many static analysis tools aggregate multiple vulnerability databases to improve vulnerability coverage[5,6,7]. However because multiple databases will often report the same vulnerability, the way these tools combine vulnerability data impacts which—and how many—vulnerabilities the tools report

[1] https://www.nist.gov/itl/nvd
[2] https://www.nist.gov/itl/nvd/nvd-news
[3] https://www.darkreading.com/vulnerabilities-threats/fall-of-national-vulnerability-database
[4] https://anchore.com/blog/navigating-the-nvd-quagmire/
[5] https://github.com/anchore/grype
[6] https://github.com/aquasecurity/trivy
[7] https://github.com/intel/cve-bin-tool

[6, 24]. Security professionals and researchers often face the challenge of receiving different vulnerability reports when analyzing the same software using static analysis tools [22, 23, 25]. Additionally, the number of vulnerabilities identified by static analysis tools influences how Software Quality Assurance (SQA) models evaluate overall system quality [15]. Discrepancies in the vulnerability databases used by these tools can lead to inaccurate assessments of system quality. Tools keeping the databases separate can lead to redundant reportings of the same vulnerability. However, aggregating vulnerability reports across databases can cause information loss, reduce transparency, and can be problematic when vulnerability databases disagree.

Aggregation and visualization across vulnerability databases is complex due to schema differences, inconsistencies in metadata, and varying levels of abstraction. For instance, the same vulnerability may have a single identifier in one database but multiple identifiers in another. Further, vulnerability databases are constantly evolving, as are the vulnerabilities they track[8].

Until now, no technology has enabled the analysis and visualization of vulnerabilities across vulnerability databases. Here, we created a graph database that integrates three of the most prominent vulnerability databases: the NVD, GitHub Advisories, and the Open Source Vulnerability (OSV) database. This graph database promotes cross-validation and enables the visualization of cliques of varying sizes while allowing for variable levels of abstraction in the underlying vulnerability databases. We improved overall vulnerability coverage and enabled security practitioners to make informed decisions based on diverse perspectives.

[8]https://media.blackhat.com/us-13/US-13-Martin-Buying-Into-The-Bias-Why-Vulnerability-Statistics-Suck-WP.pdf

Related Work

Cybersecurity researchers have explored the use of graph databases for vulnerability management. A graph database can be advantageous for mapping vulnerabilities because it does not require a single schema. Moreover, graph databases allow for the integration of diverse data sources and support complex queries (with multiple joins based on attributes such as severity, ecosystem, and software version).

Bhalker et al. [5] demonstrated how a graph database can be used to visualize relationships among vulnerabilities. The researchers [5] constructed a graph database focused on relationships, such as "Breach Types". This study advanced the use of graph databases in vulnerability analysis, but was based on a relatively small collected dataset of Cyber Security Breaches from 2009 to 2014 from varied sources.

In contrast, our graph database incorporates three extensive databases, each containing over 200,000 vulnerabilities. Our graph database interconnects vulnerabilities using both "aliased" and "related" vulnerabilities in our graphs. We also incorporate CWE, Common Vulnerability Scoring System (CVSS), and Exploit Prediction Scoring System (EPSS) data for a more holistic approach.

We are not the first to create a graph database containing the NVD. Wang et al. [34] used a knowledge graph to improve the analysis of security vulnerabilities. These researchers sourced vulnerability information from the NVD to address limitations such as poor readability and inadequate visualization of correlations between vulnerabilities. By leveraging Neo4j, they built a common vulnerabilities and exposures (CVE) knowledge graph incorporating raw data, ontology modeling, and data extraction. Their research allowed for deeper vulnerability analysis across dimensions like Common Weakness Enumerations (CWEs) and severity.

Our approach expands on previous research by incorporating a more diverse set of

Table 3: Execution time in milliseconds for Neo4j example queries. Demonstrates examples of multi-joint searches to our graph database. Queries were preformed on Ubuntu 22.04.5 and with Neo4j version 1.61.

Neo4j Query	**Execution Time (ms)**

GitHub and OSV alias pairs
`MATCH (g:GitHub)-[:alias]-(o:OSV)`
`RETURN g, o`
	174 ms

NVD and GitHub alias pairs published after 2018
`MATCH (n:NVD)-[:alias]-(g:GitHub)`
`WHERE datetime(n.published) > datetime('2018-01-01T00:00:00')`
`AND datetime(g.published) > datetime('2018-01-01T00:00:00Z')`
`RETURN n, g`
	2115 ms

Get top ten most routinely exploited vulnerabilities
`WITH ["CVE-2023-3519","CVE-2023-4966","CVE-2023-20198",`
`"CVE-2023-20273","CVE-2023-27997","CVE-2023-34362",`
`"CVE-2023-22515","CVE-2021-44228","CVE-2023-2868",`
`"CVE-2022-47966"] AS top_vulns`
`MATCH (v) WHERE v.id IN top_vulns`
`RETURN v`
	456 ms

databases. Additionally, our graph database facilitates security analysis across different identification conventions (e.g., CVE, GitHub Security Advisory [GHSA], Ubuntu security notice[USN]).

Methods

Database Selection

We obtained vulnerability data from the NVD[9], GitHub Advisories[10], and OSV[11], retrieved on March 11, 2025. We selected these vulnerability databases for their widespread adoption and integration with static analysis tools[12,13,14]. We incorporated all three

[9] https://nvd.nist.gov/vuln/
[10] https://github.com/advisories
[11] https://osv.dev/
[12] https://github.com/anchore/grype
[13] https://github.com/aquasecurity/trivy
[14] https://github.com/intel/cve-bin-tool

vulnerability databases into our graph database.

Unlike vendor-specific databases, which report vulnerabilities relevant only to their ecosystems, these three sources provide broad coverage. The NVD, maintained by the NIST, enriches vulnerabilities as CVEs. The GitHub Advisory Database aggregates data from eight sources, such as GitHub security advisories, the NVD, and the npm Security Advisories database, and labels them with GHSA identifiers[15]. The OSV integrates filtered subsets of data from 18 sources, including the GitHub Advisory Database, PyPI Advisory Database, and Go Vulnerability Database, leading to the use of multiple vulnerability identifiers. In total, the OSV has 27 different vulnerability identifier types in its database (e.g., CVE, GHSA, USN, CGA, RSEC). While GitHub Advisories and OSV are maintained by GitHub and Google respectively, both are open-source, allowing public contributions.

In our graph database, we also incorporated scores from the Exploit Prediction Scoring System (EPSS). We included EPSS to promote the evaluation of exploitability metrics concomitant with severity metrics (i.e., CVSS). EPSS assigns a score (0–1) to CVE-labeled vulnerabilities, estimating the likelihood of exploitation in the next 30 days. It also provides a percentile ranking, indicating the proportion of vulnerabilities with an equal or lower score. Integrating EPSS offers developers an additional perspective on risk assessment beyond CVSS.

We also augmented the graph with Common Weakness Enumerations (CWEs) from the CWE 1000 view[16]. CWEs classify common software weaknesses that can lead to vulnerabilities. For example, the Log4j vulnerability (e.g., 'CVE-2023-44228' in the NVD and OSV, 'GHSA-jfh8-c2jp-5v3q' in GitHub Advisories and OSV) maps to CWE-502, which describes unsafe deserialization of untrusted data. While vulnerabilities do not always map

[15] `https://docs.github.com/en/code-security/security-advisories/working-with-global-security-advisories-from-the-github-advisory-database/about-the-github-advisory-database`

[16] `https://cwe.mitre.org/data/definitions/1000.html`

to CWEs, many do.

Building The Graph Database

We integrated the vulnerabilities from the three vulnerability databases using Neo4j (version 1.61) and Python (version 3.13). All code used to build and analyze the graph database can be found at (https://github.com/MSUSEL/msu_Vuln_Database_Graph.git). Each vulnerability database was represented as a unique node type (NVD, GitHub, OSV), with each vulnerability entry corresponding to individual nodes. The attributes of these nodes were determined by the schema of the respective source databases. These attributes included information such as CVSS vectors and vulnerability descriptions.

Next, we established relationships between vulnerability nodes. The OSV schema, used by GitHub and OSV, defines an 'alias' and 'related' relationships.

A 'related' vulnerability relationship connects two or more vulnerabilities that are (1) similar but not unique, (2) multiple similar vulnerabilities codified in the same entry, or (3) do not satisfy the strict definition of the alias.

An 'alias' relationship connects two or more vulnerabilities that affect any software component the same way; i.e., either both vulnerabilities affect the software component or neither do[17]. A patch addresses all vulnerabilities with the same alias. In our graph database, we created alias relationships for vulnerabilities with identical identifiers across the databases.

To add EPSS data to the ontology, we built nodes 'EPSS', where each node is the score from a CVE ID and has attributes of the EPSS score, including the percentile. Corresponding nodes with a CVE ID were connected using a relationship to the EPSS node called 'epssScore'.

Each CWE in the CWE-1000 view was built into nodes, with attributes such as

[17] `https://ossf.github.io/osv-schema/`

'description', 'id', and 'type'. CWE nodes have relationships to vulnerability nodes via a 'weakness' relationship. The 'weakness' relationship details are provided by the three vulnerability databases. The CWE nodes also have their hierarchal mappings with levels of abstraction represented through 'parent' and 'child' relationships between the CWE nodes.

Analysis

Our analyses were three-pronged. First, we investigated the overlap of vulnerabilities across the three databases. In particular, we investigated discrepancies in attributes between alias nodes.

Second, we conducted a pairwise investigation for each alias relationship to determine consistency in CVSS vectors across the databases. Vectors contain manually inputted data that is used to calculate a CVSS score for a vulnerability. To promote a valid comparison, we only compared vulnerability pairs when each pair used the same version of CVSS. Different versions of CVSS are expected to generate different CVSS vectors. Therefore, we compared the CVSS vector provided by each underlying vulnerability database. If all databases had identical vulnerability data, then the CVSS vectors for all vulnerabilities that are aliases of one another would be identical.

Third, we analyzed the top ten most routinely exploited vulnerabilities in 2023 reported by DHS CISA[18] to determine the connectedness and similarities in reporting of the top ten vulnerabilities across the databases. Note that the top ten for 2024 were not available in February 2025, when this paper was written. Our exploration of the top ten vulnerability subgraph included traversing vulnerability node relationships 'related', 'alias', 'weakness', 'epssscore' and the CWE nodes 'parent' and 'child' relationships. This exploration of the subgraph gave us a holistic view of the top ten vulnerabilities.

We also tracked execution time for common queries (Table 3) to demonstrate the

[18] `https://www.cisa.gov/news-events/cybersecurity-advisories/aa24-317a`

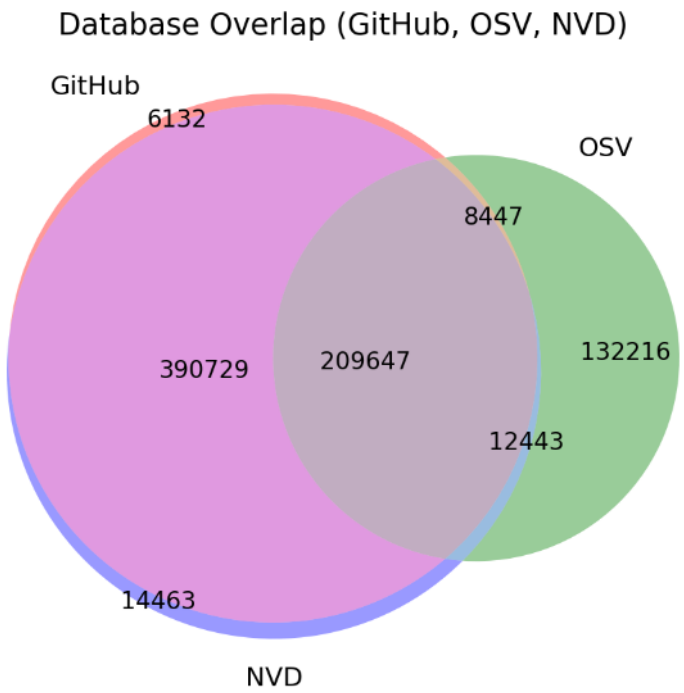

Figure 4: Shows the distribution of the total 774,077 vulnerability nodes across databases. Overlap occurs when nodes in different databases have an alias relationship between them, implying that these are the same vulnerabilities in different databases.

response time of our graph database from a user perspective.

Results

We found an increase in total vulnerability coverage by integrating the NVD, OSV, and GitHub Advisories. Further, we found that the databases overlap in vulnerabilities (Fig. 4), allowing for cross-referencing across vulnerability databases. Such cross-referencing revealed differences in CVSS vectors for aliased vulnerabilities.

Vulnerability Database Relationships

Our graph database successfully integrated data from three major vulnerability databases. Each database contributed a distinct subset of the overall vulnerability landscape. In total, the NVD contributed 276,339 nodes, GitHub Advisories contributed 268,644

vulnerability nodes, and OSV contributed 229,094 nodes for a total of 774,077 vulnerability nodes. Additionally, we had 281,448 EPSS nodes and 988 CWE nodes for a total of 1,056,513 nodes.

The NVD uses only one ID type (CVE). Similarly, GitHub only uses GHSA identifiers. Thus, the NVD has no alias or related relationships among NVD nodes; similarly, GitHub has no alias or related relationships among GHSA nodes. In contrast, the OSV database has 27 different identifiers from 18 different data sources. Therefore, the OSV database has interconnected nodes representing alias and related relationships. In total, the OSV contains 43,675 vulnerabilities with an alias and 114,882 vulnerabilities with a related vulnerability.

When comparing the three vulnerability databases against each other, GitHub contributes 6,132 distinct vulnerabilities without aliases in the NVD or OSV databases. The NVD database contributes 14,463 distinct vulnerabilities, and the OSV database contributes 132,216 distinct vulnerabilities (Fig. 4). In total, 621,266 of the 774,077 vulnerability nodes have an alias.

Beyond aliases, our graph database nodes contained 625,439 'related' vulnerability relationships. Additionally, 672,124 nodes had an EPSS score. Vulnerabilities mapped to CWEs 1,022,272 times (as a single vulnerability can be associated with multiple CWEs). Overall, the graph database comprises 2,932,444 relationships.

Discrepancies in Severity Scores

Discrepancies in CVSS vectors were common. In our pairwise comparison, we identified 281,817 alias relationships where only one node had a CVSSv2 vector. Notably, there were zero instances where both nodes in a pair had CVSSv2 vectors (Table 4), making the comparison of CVSSv2 vectors impossible.

There were 65,898 alias relationships in which only one node had a CVSSv3 vector, preventing direct comparison for these pairs. However, 16,729 vulnerability pairs had

Table 4: For alias pairs with the same CVSS metric version, we analyze vector differences. "Total Matched" counts such pairs, while "No Match" indicates pairs where only one node uses that CVSS version. The "%" column shows the percentage of alias pairs with differing CVSS vectors.

	Disagreement	**Matched**	**%**	**No Match**
CVSSv2	0	0	NA	281,817
CVSSv3	65	16,729	0.38%	65,898
CVSSv3.1	27,408	214,696	12.76%	99,882
CVSSv4	392	6,971	5.62%	22,221

CVSSv3 vectors for both nodes, with 65 instances of disagreement in vectors (0.38%).

In total, 214,696 alias relationships had CVSSv3.1 vectors, with 27,408 instances of disagreement (12.76%). In total, 99,882 alias relationships were not comparable for CVSSv3.1 due to differences in how CVSS vectors were constructed.

For CVSSv4, there were 22,221 alias pairs that could not be compared due to differences in how CVSS vectors were constructed. Among the 6,971 vulnerability pairs where both nodes had CVSSv4 vectors, 392 pairs (5.62%) had differing CVSSv4 vectors.

Severity vectors for a vulnerability often depended on the underlying vulnerability database. The discrepancies in CVSS vectors across alias pairs, specifically with version CVSSv3.1, highlight important discrepancies across the NVD, GitHub, and OSV databases. Moreover, these discrepancies indicate the subjectivity inherent in quantifying the severity of vulnerabilities, even when using the same scoring system [33].

Top Ten Routinely Exploited Vulnerabilities

The utility of our graph database is evident from our analysis of the database coverage of the top ten most frequently exploited vulnerabilities, their CVSS and EPSS scores, and their mappings to CWEs. The top ten vulnerabilities were reported using CVE IDs. The NVD included all ten vulnerabilities, whereas the OSV database contained only one, CVE-2021-44228. Although GitHub does not label vulnerabilities with CVE IDs, we utilized the

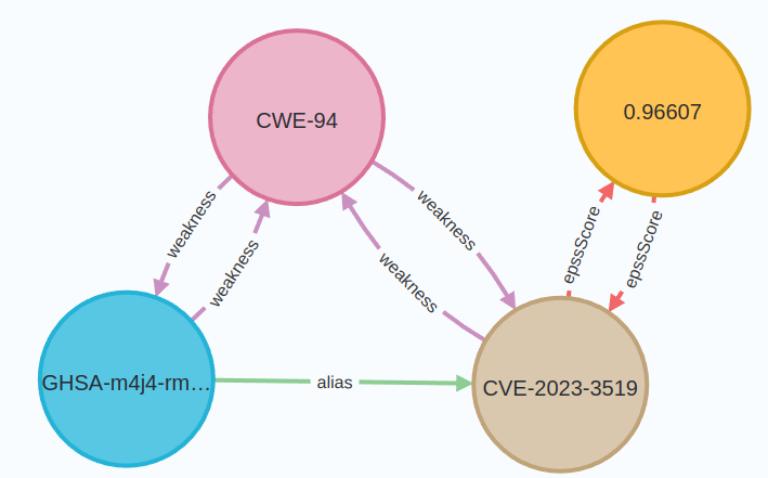

Figure 5: Graph of the top exploited vulnerability, CVE-2023-3519. The blue GHSA node GHSA-m4j4-rmj5-w5gp represents an alias listed in GitHub Advisories, while the CVE node originates from the NVD. Both databases classify this as a CWE-94 weakness, and CVE-2023-2519 has an EPSS score of 0.96607

alias relationships of the CVE nodes to identify all ten vulnerabilities under GHSA IDs in the GitHub advisories database.

Severity vectors were consistent for the top ten vulnerabilities. The CVSSv3.1 severity scores were consistent across the databases for the top ten vulnerabilities. The lowest severity score was CVE-2023-20273 at 7.2.

EPSS scores were surprisingly low for three of the ten most exploited vulnerabilities. Whereas seven of the top ten vulnerabilities had EPSS scores greater than 0.88, the remaining 3 top ten routinely exploited vulnerabilities had low EPSS scores: CVE-2023-20273 had an EPSS score of 0.07, CVE-2023-27997 had an EPSS score of 0.10651, and CVE-2023-2868 had an EPSS score of 0.07893. These low EPSS scores contradict the exploitability ranking provided by DHS CISA.

When we queried all weaknesses associated with the top ten vulnerabilities, we found that many of the top ten vulnerabilities share CWEs (Fig. 6). The most shared weakness was CWE-20, Improper Input Validation, with 4 of the top ten mapping to it.

CWE mappings can be inconsistent across vulnerability databases. One illustrative

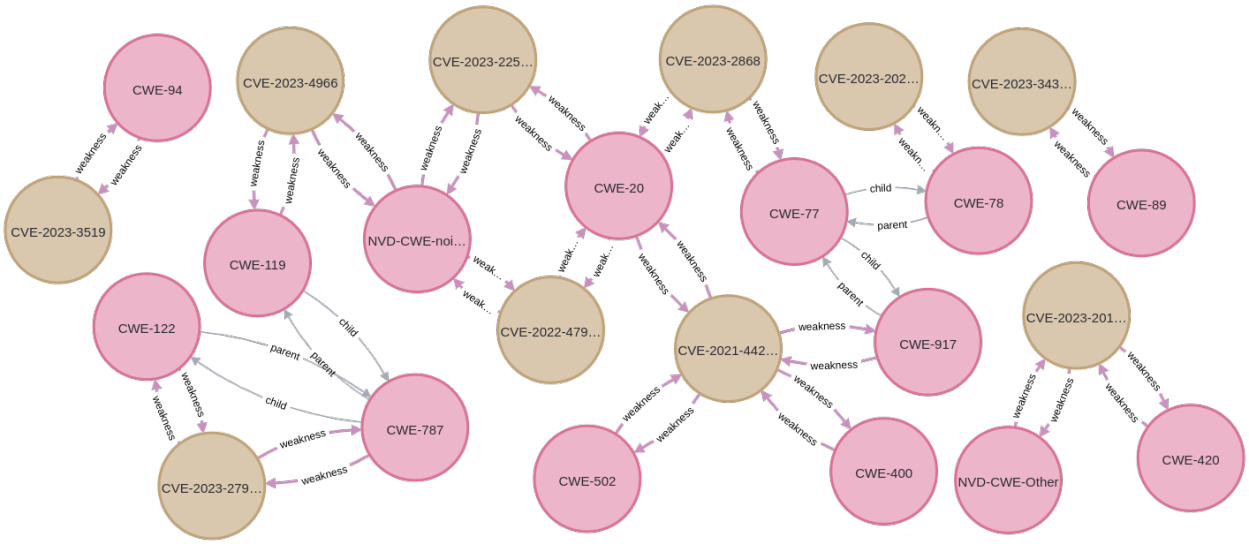

Figure 6: Top exploited vulnerabilities in the NVD database mapped to CWEs. Here, we can see many of the top vulnerabilities and shared CWEs. The top ten vulnerabilities map to 14 CWEs. Two of those CWEs are NVD-CWE-noinfo and NVD-CWE-Other. The graph image was generated by Neo4j.

mapping is for CVE-2023-4966, where the NVD maps it to CWE-199 and NVD-CWE-noinformation. Upon further investigation, we know this occurs because the NVD uses a secondary source, 'secure@citrix.com', that maps the id to CWE-199, but the NVD, as a primary source, has no information on what weakness the id maps to.

By including the CWE 1000 view of parent and child relationships, we added another layer of detail to our graph database. Using Neo4j we can specify the depth of parent-child relationships we want visualized. When looking at the vulnerability that mapped to the most CWEs, CVE-2021-44228, one parent-child relationship deep we found two of the CWEs, CWE-502 and, CWE-400,mapped to the same parent CWE, CWE-664.

Looking one parent-child CWE relationship deep for the top ten vulnerabilities the resulting graph has only two path components. Nine of the top vulnerability nodes have a connected chain of parent-child relationships between their CWE nodes. Only one vulnerability, CVE-2023-20198, had no path of relationships connecting it to the other top ten vulnerabilities.

Discussion

In this research, we proposed unifying multiple vulnerability databases—NVD, GitHub Advisories, and OSV—into a single integrated graph database. We expanded the database with additional layers of vulnerability information by including the EPSS scores and the CWE-1000 view. Although the integration of information from these databases improved our overall domain coverage, it also revealed that there are significant overlaps and inconsistencies of information between databases. The overlaps are exposed through the examination of the relationships (e.g., aliases) between the vulnerability nodes, and inconsistencies are uncovered through variations in attribute measurements (e.g., differing CVSS scores for the same vulnerability).

Making vulnerability management fast and easy for developers and practitioners is imperative. Our analysis of the top ten most routinely exploited vulnerabilities identified by CISA reveals the utility of our integrated database solution. For example, the mapping of CWEs with corresponding CVEs revealed significant patterns in the top ten vulnerabilities, showcasing how a graph database can reveal valuable insights quickly.

Graph Construction and Connectivity

An integrated graph database solution offers interconnected, varied perspectives for vulnerability management. By incorporating three databases maintained by prominent organizations, GitHub, Google, and NIST, we improve overall ecosystem coverage and robustness.

In total, our database has 774,077 (Fig. 4) vulnerability nodes and 1,056,513 nodes when including EPSS and CWE nodes. The integrated graph database has 2,932,444 relationships informing users of EPSS scores, associated weaknesses, aliases, related vulnerabilities, and parent-child relationships with corresponding CWEs. Over 621,266 vulnerabilities from the

combined datasets have alias relationships, allowing for cross-database comparisons. To further highlight the utility of an integrated graph database solution, this practical approach allows practitioners to easily access information with fast and easy-to-create queries.

Discrepancies in Cross-Database Analysis

Filtering vulnerability reports by severity is a common practice in vulnerability management [31], and practitioners often use CVSS vectors to prioritize vulnerabilities. A key finding from our analysis is the inconsistency in CVSS vectors reported by different databases. Our analysis of CVSSv3.1 vectors from pairwise comparisons between two distinct databases revealed discrepancies in 12.76% of the cases, highlighting the subjective nature of severity assessments. Previous research has shown that CVSS vectors are subjective, with up to 68% of evaluators assigning different severity ratings for the same vulnerabilities, demonstrating inconsistencies in the scoring process [36].

Our integrated graph database solution does not determine which source provides the 'correct' severity score; instead, it allows practitioners to decide which source they trust, by offering alternatives that can enable informed decisions.

Furthermore, integrating additional EPSS score data provides an additional perspective. EPSS offers an empirical measure of the likelihood that a vulnerability will be exploited. However, EPSS scores only map to vulnerabilities with CVE IDs. Utilizing the 'alias' relationships in our graph database, vulnerabilities with different IDs, such as GHSA or USN, can still access this extra layer of information. This allows developers the autonomy to choose the sources that fit their needs without incurring the extra effort of scouring the internet.

Using the Graph Database

Using DHS CISA's top ten most frequently exploited vulnerabilities as an example, we see how the relationships between vulnerabilities and their associated CWEs are suggestive of

common patterns in exploit techniques. More specifically, many of the top ten vulnerabilities share CWEs.

By incorporating CWE parent-child relationships, we found that 9 out of the top 10 vulnerabilities were related. The connectivity between the CWEs of the top ten most routinely exploited vulnerabilities highlights the benefit of this integrated view. Importantly, patterns of weakness that are commonly exploited emerge without the need to scan multiple vulnerability databases or click one's way through the CWE-1000 view.

Conclusion

Our database provides a computational solution that decreases cognitive workloads and the chance that a relationship will be missed via manual error. Moreover, query execution time is fast despite containing over a million nodes and nearly three million relationships (Table 1). Thus, by integrating multiple vulnerability databases into a single graph database, we provide an elegant and efficient solution for the onerous and vexing problem of vulnerability mapping and management.

Our future work will focus on expanding data sources, incorporating real-time updates, and providing an API for accessibility. By strengthening this solution, we aim to further support security practitioners in mitigating threats with greater accuracy and efficiency.

Acknowledgment

I want to thank Gage Nesbit for his contributions to the graph database, Eric O'Donoghue for his table beautification skills, as well as Madison Munro and Ryan Cummings for their feedback. This research was conducted with support from the U.S. Department of Homeland Security (DHS) Science and Technology Directorate (S&T) under contract 70RSAT22CB0000005. Any opinions contained herein are those of the author and do not

necessarily reflect those of DHS S&T.

CONCLUSION

I provide an analysis of static analysis tools to understand discrepancies in vulnerability detection (Chapter 3). I found that Trivy's broader selection of external databases did not lead to more vulnerabilities being detected. However, the external source selection and aggregation techniques did affect what vulnerabilities were reported and how many. Additionally, differences in how each tool handles related vulnerabilities and assigns severities resulted in the same vulnerabilities being reported with differing severities. This manuscript contributes to our understanding of how the design choices of different static analysis tools impact vulnerability reports.

The findings on discrepancies in static analysis tools underscore the challenges posed by inconsistencies in vulnerability databases and the need for improved aggregation strategies. I created a unified graph database solution that addresses aggregation issues by integrating OSV, GitHub Advisories, and NVD, enabling seamless cross-referencing across multiple sources. This approach preserves critical information by linking vulnerabilities under different naming conventions, including EPSS scores, CVSS ratings, and CWE classifications. This structure eliminates the need for time-consuming manual reconciliation and provides a comprehensive view of vulnerability data.

My analysis of the top ten routinely exploited vulnerabilities demonstrates the value of this approach. While the NVD contained all ten vulnerabilities, OSV included only CVE-2021-44228, and GitHub Advisories required alias mapping to identify them all. Beyond database coverage, my investigation into CWE relationships revealed a striking trend: nine vulnerabilities shared linked weaknesses, exposing patterns in the techniques used to exploit them. These findings highlight how a graph database streamlines vulnerability analysis, equipping security professionals with the tools to detect trends and make more informed security decisions.

This thesis contributes to the field of vulnerability management by uncovering the effects of vulnerability database aggregation on static analysis tool reports and proposing an integrated graph database solution. I identified discrepancies in vulnerability detection and severity reporting by analyzing Trivy and Grype (Chapter 3), emphasizing the need for improved aggregation strategies. In response, Chapter 4 presents a unified graph database that integrates data from OSV, GitHub Advisories, and NVD, providing a comprehensive and easily accessible view of vulnerabilities. This approach links vulnerabilities across different naming conventions and preserves critical data such as EPSS scores, CVSS ratings, and CWE classifications, simplifying the reconciliation of conflicting information. Our analysis of the top ten exploited vulnerabilities further demonstrates the utility of this system in identifying patterns and trends, ultimately enabling security professionals to make more informed and efficient decisions.

REFERENCES CITED

[1] Sobhy Abdelkader, Jeremiah Amissah, Sammy Kinga, Geofrey Mugerwa, Ebinyu Emmanuel, Diaa-Eldin A. Mansour, Mohit Bajaj, Vojtech Blazek, and Lukas Prokop. Securing modern power systems: Implementing comprehensive strategies to enhance resilience and reliability against cyber-attacks. *Results in Engineering*, 23:102647, 2024.

[2] Ashish Aggarwal and Pankaj Jalote. Integrating static and dynamic analysis for detecting vulnerabilities. In *30th Annual International Computer Software and Applications Conference (COMPSAC'06)*, volume 1, pages 343–350, 2006.

[3] Tawfiq M. Aljohani. Cyberattacks on energy infrastructures as modern war weapons—part i: Analysis and motives. *IEEE Technology and Society Magazine*, 43(2):59–69, 2024.

[4] Afsah Anwar, Ahmed Abusnaina, Songqing Chen, Frank Li, and David Mohaisen. Cleaning the nvd: Comprehensive quality assessment, improvements, and analyses. *IEEE Transactions on Dependable and Secure Computing*, 19(6):4255–4269, 2022.

[5] Pratima M. Bhalekar and Jatinderkumar R. Saini. Comprehensive exploration of the role of graph databases like neo4j in cyber security. In *2024 International Conference on Emerging Smart Computing and Informatics (ESCI)*, pages 1–4, 2024.

[6] Brittany Boles, Eric O'Donoghue, A. Redempta Manzi Muneza, Garrett Perkins, Clemente Izurieta, and Ann Marie Reinhold. Deciphering Discrepancies: A Comparative Analysis of Docker Image Security . In *2024 IEEE International Conference on Source Code Analysis and Manipulation (SCAM)*, pages 254–259, Los Alamitos, CA, USA, October 2024. IEEE Computer Society.

[7] Kelly Brady, Seung Moon, Tuan Nguyen, and Joel Coffman. Docker container security in cloud computing. In *2020 10th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0975–0980, 2020.

[8] D.T. Campbell and J.C. Stanley. *Experimental and Quasi-experimental Designs for Research*. R. McNally, 1966.

[9] Foteini Cheirdari and George Karabatis. Analyzing false positive source code vulnerabilities using static analysis tools. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 4782–4788, 2018.

[10] T.D. Cook and D.T. Campbell. *Quasi-experimentation: Design & Analysis Issues for Field Settings*. Houghton Mifflin, 1979.

[11] Roland Croft, M. Ali Babar, and M. Mehdi Kholoosi. Data quality for software vulnerability datasets. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 121–133, 2023.

[12] Roland Croft, M. Ali Babar, and Li Li. An investigation into inconsistency of software vulnerability severity across data sources. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 338–348, 2022.

[13] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-scale vulnerability analysis. In *Proceedings of the 2006 SIGCOMM Workshop on Large-Scale Attack Defense*, LSAD '06, page 131–138, New York, NY, USA, 2006. Association for Computing Machinery.

[14] Chang-Hoi Hur, Seong-Pyo Kim, Yoon-Soo Kim, and Jung-Ho Eom. Changes of cyber-attacks techniques and patterns after the fourth industrial revolution. In *2017*

5th International Conference on Future Internet of Things and Cloud Workshops (FiCloudW), pages 69–74, 2017.

[15] Carlos Izurieta, Darius Reimanis, Eoghan O'Donoghue, Kanishka Liyanage, Ange Robert Manzi Muneza, Benjamin Whitaker, and Alexander M. Reinhold. A generalized approach to the operationalization of software quality models. *PeerJ Computer Science*, 10:e2357, 2024.

[16] Yuning Jiang, Manfred Jeusfeld, and Jianguo Ding. Evaluating the data inconsistency of open-source vulnerability repositories. In *Proceedings of the 16th International Conference on Availability, Reliability and Security*, ARES '21, New York, NY, USA, 2021. Association for Computing Machinery.

[17] Pontus Johnson, Robert Lagerström, Mathias Ekstedt, and Ulrik Franke. Can the common vulnerability scoring system be trusted? a bayesian analysis. *IEEE Transactions on Dependable and Secure Computing*, 15(6):1002–1015, 2018.

[18] R. Kalaiselvi, Shruthi Ravisankar, Varun M, and Dharanipriya Ravindran. Enhancing the container image scanning tool - grype. In *2023 2nd International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA)*, pages 1–6, 2023.

[19] Taeyoung Kim, Seonhye Park, and Hyoungshick Kim. Why johnny can't use secure docker images: Investigating the usability challenges in using docker image vulnerability scanners through heuristic evaluation. In *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '23, page 669–685, New York, NY, USA, 2023. Association for Computing Machinery.

[20] Yoonjong Na, Seunghoon Woo, Joomyeong Lee, and Heejo Lee. Cneps: A precise approach for examining dependencies among third-party c/c++ open-source components. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[21] Marcus Nachtigall, Lisa Nguyen Quang Do, and Eric Bodden. Explaining static analysis - a perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 29–32, 2019.

[22] Eric O'Donoghue, Brittany Boles, Clemente Izurieta, and Ann Marie Reinhold. Impacts of software bill of materials (sbom) generation on vulnerability detection. In *Proceedings of the 2024 Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses*, SCORED '24, page 67–76, New York, NY, USA, 2024. Association for Computing Machinery.

[23] Eric O'Donoghue, Ann Marie Reinhold, and Clemente Izurieta. Assessing security risks of software supply chains using software bill of materials. In *2024 IEEE International Conference on Software Analysis, Evolution and Reengineering - Companion (SANER-C)*, pages 134–140, 2024.

[24] Ann Marie Reinhold, Brittany Boles, A. Redempta Manzi Muneza, Thomas McElroy, and Clemente Izurieta. Surmounting challenges in aggregating results from static analysis tools. *Military Cyber Affairs*, 7, 2024.

[25] Ann Marie Reinhold, Travis Weber, Colleen Lemak, Derek Reimanis, and Clemente Izurieta. New version, new answer: Investigating cybersecurity static-analysis tool findings. In *2023 IEEE International Conference on Cyber Security and Resilience (CSR)*, pages 28–35, 2023.

[26] Bonan Ruan, Jiahao Liu, Chuqi Zhang, and Zhenkai Liang. Kernjc: Automated vulnerable environment generation for linux kernel vulnerabilities. In *Proceedings of the 27th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '24, page 384–402, New York, NY, USA, 2024. Association for Computing Machinery.

[27] Joshua Schimel. *Writing Science: How to Write Papers That Get Cited and Proposals That Get Funded*. Oxford University Press, New York, 2012.

[28] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, CODASPY '17, page 269–280, New York, NY, USA, 2017. Association for Computing Machinery.

[29] Mehmet Söylemez, Bedir Tekinerdogan, and Ayça Kolukısa Tarhan. Challenges and solution directions of microservice architectures: A systematic literature review. *Applied Sciences*, 12(11), 2022.

[30] Zach Wadhams, Ann Marie Reinhold, and Clemente Izurieta. Automating static code anlaysis through ci/cd pipeline integration. In *2nd International Workshop on Mining Software Repositories for Privacy and Security*. IEEE International Conference on Software Analysis, Evolution and Reengineering, 2024. [In-press].

[31] Zachary Douglas Wadhams, Clemente Izurieta, and Ann Marie Reinhold. Barriers to using static application security testing (sast) tools: A literature review. ASEW '24, page 161–166, New York, NY, USA, 2024. Association for Computing Machinery.

[32] Haoyu Wang, Yao Guo, Ziang Ma, and Xiangqun Chen. Wukong: a scalable and accurate two-phase approach to android app clone detection. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 71–82, New York, NY, USA, 2015. Association for Computing Machinery.

[33] Ruyi Wang, Ling Gao, Qian Sun, and Deheng Sun. An improved cvss-based vulnerability scoring mechanism. In *2011 Third International Conference on Multimedia Information Networking and Security*, pages 352–355, 2011.

[34] Yongfu Wang, Ying Zhou, Xiaohai Zou, Quanqiang Miao, and Wei Wang. The analysis method of security vulnerability based on the knowledge graph. In *Proceedings of the 2020 10th International Conference on Communication and Network Security*, ICCNS '20, page 135–145, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Susheng Wu, Wenyan Song, Kaifeng Huang, Bihuan Chen, and Xin Peng. Identifying affected libraries and their ecosystems for open source software vulnerabilities. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery.

[36] Julia Wunder, Andreas Kurtz, Christian Eichenmüller, Freya Gassmann, and Zinaida Benenson. Shedding light on cvss scoring inconsistencies: A user-centric study on evaluating widespread security vulnerabilities. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 1102–1121, 2024.