

A Graph-Based National Vulnerability Database Model for Securing and Studying Modern Threat Environments

Jim Jazwiecki

Nathan Kluth

Abstract—Established in 2005, the National Vulnerability Database (NVD) is one of several examples around the world of government-led repositories of information about vulnerabilities in software, firmware, and hardware. It provides a useful resource for collecting and disseminating information about isolated vulnerabilities, but its current format – a collection of standalone indexed documents pertaining to individual vulnerabilities – is sub-optimal for use in environments where multiple hosts, running multiple network-connected software packages, co-exist in a single network, allowing attackers to construct multi-stage attacks by chaining together vulnerabilities. In this paper, we present a graph database model for representing the NVD using Neo4j and the NVD’s public data feeds. We also show that there exist graph queries which can be used in security research which are orders of magnitude more efficient than their relational counterparts for use in evaluating multi-stage threats. We conclude this model is better-adapted for evaluating real-world security risks in a given environment, allowing for the efficient creation of practical attack graphs and facilitating vulnerability research.

I. INTRODUCTION

The Common Vulnerabilities and Exposures (CVE) data stored in the National Vulnerabilities Database (NVD) offers security professionals a resource for navigating and searching for publicly listed vulnerabilities.[MG02] In its current form, the NVD contains information about vulnerability severity, attack vectors, impacts, and web-based supplemental information citations, as well as details about the specific technology affected.[NIS18] This data is made public on a searchable website and in structured data feeds available for download. Prior work in this area has been devoted to gaining insights on the current information exposed on the NVD website.[ZNW10] Existing methods for carrying out this work are out-of-date and brittle.[AT14] Furthermore, they are inefficient for researching and mitigating multi-stage attacks, those which chain vulnerabilities, software packages, and hosts together in order to achieve attacker goals. Although the lack of a comprehensive, structured database of computer attacks (successful or otherwise) makes authoritative statistics about the growth in frequency of multi-stage attacks difficult to source, their relative growth in importance can be imperfectly inferred from the fact that number of research publications mentioning the term in a given year, as listed in Google Scholar, has roughly quadrupled since the NVD was launched (see Appendix I). As research in this area increases, the inflexibility of the existing NVD representations to efficiently complement

this threat environment presents a research bottleneck and information loss point for security vulnerabilities.[ZCO11]

Attack graphs, an efficient data structure for studying multi-stage attacks, have been explored by security researchers since 2000[RA00] in order to identify how combinations of configurations and hosts on the same network contribute to the network’s vulnerabilities. With varying levels of complexity or detail, they typically represent links between hosts, services, preconditions, and postconditions which an attacker could use to reach a goal, such as administrative permissions on a target system.[SO11] Because host and network configurations in production systems may need to be frequently updated, necessitating interactive analysis, Wang et al.[LWJ08] developed a relational model supplanting earlier proprietary, custom graphs, which replaced algorithmic processing of the proprietary graph with standardized relational queries. In practice, this solution was not scalable to online analysis of real-world networks, driving attack graph research towards non-proprietary, interactively queryable graph databases.[BM14] These approaches typically focus on modeling existing or theoretical systems, based on a known or planned environment, but they can only identify critical paths to attacker goals, not provide prescriptive information to inform vulnerability-reducing responses. For example, while it is useful to know that a given version of a package in an environment is vulnerable, it is even more useful to know every minor upgrade of that version is vulnerable to a different CVE with a similar vector or that the probability of future versions from the same vendor having yet-undiscovered but similar vulnerabilities is above average.

The NVD as it exists today can not be easily used to integrate the information required for these enhancements, so we propose the construction of a graph-based version of the NVD which would unlock new analytic capabilities in security research, including a proposed schema and tools for creating and easily working with this database interactively, even in high-security environments. We demonstrate that the graph-based materialization of this data is more efficient for these purposes than a relational database. Section 2 discusses previous approaches to collecting or using this data, and the design of graph databases. The technical choices we made during our research, including data sources, platforms, and our proposed NVD graph schema are covered in Section 3. Several sample queries are demonstrated as answers

to theoretical research questions, including one specifically taking advantage of a graph database’s strengths, and it is compared to a theoretical relational database representation of the NVD in section 4. Finally, we discuss our conclusion and future work in sections 5 and 6, respectively.

II. PREVIOUS WORK

Previous work on vulnerability research have been involved with data gathering and reporting, as well as representation of data. For data gathering, web crawler and mining approaches have been used to try and extract useful information and actionable insights from text. Once data is extracted, storing it in a format that is easy to use becomes the primary goal.

A. Crawler

A web crawler (sometimes called a spider or “web scraping”) is a program that systematically browses the target website, pulling text it finds out as data. [CGMP98] Data displayed by most websites can only be viewed using a web browser or program capable of making HTTP requests. Data export options are limited, and users must resort to manually copying and pasting the information they are interested in. [NIS18] This become a very tedious task, so many seek to automate this process through the use of web crawlers.

While this does offer some relief to any manual work, it is merely treating the symptoms and not the underlying cause of inaccessible data. Coupled with the fact that setting up a web crawler is a difficult task, the costs do not outweigh the benefits.

B. Mining

Data mining is a similar technique where the patterns are discovered by a learning algorithm, which then attempts to intelligently gather data. As a result, the data extracted is often more valuable however the process to set this up is even more technical than that of a web crawler. [Han07] The data obtained in each of these approaches does not offer connections between the data that is pulled out.

In [Mul+11], Mulwad et al. describe a process for extracting information based on a lexicon from an outside source. While this yields interesting insights, there are potential areas for missed data as the system first finds potential targets before moving on to extract data. While this technique may find patterns that the researcher may not be aware of, the data is still not in a state to easily query and find information about patterns that the researcher may in fact be aware of. In [ZCO11] the techniques focus on the analysis, rather than the data preparation; insights are shown into time to next vulnerability measurements. The bottleneck of data preparation and processing is where our focus lies, and insights like these will still be obtainable with easier data collection.

Neither of these methods offer a holistic approach to querying and analyzing the data that is present on the NVD database. Additionally, both these methods required

technical expertise for each query, while our solution offers ease of access to any user with knowledge of the Cypher query language. [Fra+18] By moving the data to a graph database, we rid the system issues that arise due to the inadequate medium that currently houses the CVE data. [Rod+18] Additionally, storing the data in a graph database can still benefit from artificial intelligence. We will show how this unlocks not only robust queries and patterns but also increases productivity and performance across everyday data-gathering.

C. Representation

Representing the data becomes difficult when using proprietary information and specialized algorithms. [LWJ08] Instead, a model should be used to representing the vulnerabilities.

Graph databases are an up-and-coming technology that stores data and relationships in a graph structure with nodes and edges. A node is an entry in the database, and can be anything. Figure 2 shows how a basic node might connect to another node in the graph. A graph is defined as:

$$G = (V, E)$$

where V is a finite, nonempty set of vertices and E is a set of edges (pairs of vertices).

An attack graph depicts ways in which an adversary can exploit vulnerabilities to break into a system. Attack graph generation has been approached many different ways. [SO11] [OBM06] In, [SO11], the authors showed that to accurately assess the security of enterprise systems, vulnerabilities need to be viewed in how they can be combined. This combination can be time consuming, so work was done to generate these graphs with the use of specialized tools. [OBM06] However, this generation was still time-consuming and did not generalize to other systems. In [LWJ08], these normally proprietary graphs were turned into relational models, which alleviated the dependency on proprietary implementations of specialized algorithms. This removed the delay that comes from writing these specific queries. We build on top of this concept, taking advantage of the graph model for data storage along with attack graph representation.

Neo4j is a popular graph database that takes a connections-first approach to data. It is open source and offers expressive querying using the Cypher query language. [Neo18] By moving the CVEs from HTML storage to Neo4j we unlock the power of relationships between various CVEs.

CVEs are stored as a text document in HTML on the NVD website. Figure 1 shows some basic fields that are present for a given CVE. Having a standard to work from helps to define properties of each node. [HH05]

III. METHODOLOGY

Although it would be possible to write a web crawler that would scrape all of the data from the NVD’s public site, in light of the fact that the NVD currently exposes views of their complete data set in commonly supported, well-structured formats, as well as the implicit necessity

Name: CVE-2018-19568

Description: A floating point exception in `kodak_radc_load_raw` in `dcraw` through 9.28 could be used by attackers able to supply malicious files to crash an application that bundles the `dcraw` code.

Source: MITRE

Figure 1: A sample CVE entry. Source: <https://nvd.nist.gov/vuln/detail/CVE-2018-19568>

Figure 2: An example of a graph with two nodes connected by a directed edge.



of frequently (or continually) re-crawling the site to ensure the mirrored CVE data is up-to-date, with its concomitant bandwidth overhead, we have decided to use the public data feeds as the canonical data source for our model. For reasons unknown, not all of the data exposed on the HTML version of the NVD is included in the two different XML versions of the database, so we selected the JSON feeds, which are more complete. The National Institute of Standards and Technology (NIST), which maintains the NVD, provides a supplemental feed of CVE updates made in the preceding eight days, distinct from their primary data sets, which can be used to update local copies of the database without requiring either the frequent re-crawling alluded to above, or completely rebuilding the graph database from scratch, making this the most efficient and reliable way to work with a local copy of the NVD. Although these update feeds could be used in concert with a crawled, parsed version of the HTML-encoded data, the speed with which it is possible to download and process the complete NVD, relative to a crawler's speed, without risking the kinds of client IP address bans to which anonymous crawlers are sometimes subjected, further supports our choice. If our tools are adopted, we would prefer they be considered conscientious consumers of NIST's preferred methodology for mirroring the NVD.

As to our choice of graph database, because our goal was producing a non-commercial tool for academic security research, we prioritized open-source platforms with non-commercial licenses, ease of use, and documentation quality. According to DB-Engines.com[DE18], Neo4j is the market leader, twice as popular as the next graph database system. Because we would like to make the product of our research as widely available as possible, we chose a platform-independent, self-contained "Community" edition of Neo4j built into a Docker image. Docker is an open-source platform, designed to provide a complete, isolated environment with bundled dependencies, ameliorating some of the complications which might accompany setting up a database system in different environments. The "Community" edition is free for non-commercial use, and the fact that it is built into

a Docker image maintained by Neo4j's makers should make it easy for other researchers to use on any operating system while trusting that any executable code is safely sandboxed in a virtual machine. The Docker image exposes three ports through which other systems may interact with Neo4j: HTTP, HTTPS, and Bolt, Neo4j's proprietary binary protocol, with an optional authentication requirement available for all three. In all other ways, it is run in complete isolation. Finally, in order to import the data, it was necessary to install an open-source supplementary Neo4j plugin, APOC, a step which the Docker build automatically executes.

A. Schema

Considering possible uses for a graph representation of the NVD, our organizing principle was designing a schema that would privilege the strengths of graph databases over relational databases. Specifically, in order to facilitate research of multi-stage attacks, we needed to produce a connected graph. We reviewed the data exposed by NIST in the NVD JSON feeds to determine which fields would be best served as nodes, and which as properties on those nodes.

Each entry in the NVD consists of a unique ID, a human-readable description, reference URLs, and structured data relating to the nature of the vulnerability, including its attack vector, complexity, scope, impact, affected products, and exploitability. Most of the structured data is defined by the Common Vulnerability Scoring System (CVSS), and the NVD includes both versions 2.0 and 3.0 of the CVSS. Our work focused on version 3.0 of the CVSS, as this is the latest version in use, but we have included the 2.0 data as well. This was necessary because more than half of the existing vulnerabilities predate the introduction of CVSS 3.0 in June of 2015, and the NVD does not include CVSS 3.0 data for those, although it does continue to include 2.0 data for more recent CVEs. Because there are differences in the definitions of properties and their values, to synthesize CVSS 3.0 data from CVSS 2.0 data would compromise the integrity of this project as a "copy" of the public NVD. The structured data is of three types, Boolean flags, floating points, and string. Floating point values between 0 and 10.0 are used to gauge the relative exploitability, impact, and "base" (a combination of exploitability and impact) "scores". NVD CVEs do not include the "temporal" or "environmental" scores defined by the CVSS specification, on the grounds that they are, respectively, dependent on events external to the vulnerability or specific to an organization. All of the CVSS data is also available compressed into a single string of abbreviations known as the "vector string".

Creating individual CVE nodes was an obvious starting point for building an NVD graph. The hierarchical relation-

ship between product versions, products, and the vendors who control them suggested three more connected node types, which would, in turn, be linked back to the CVE by the affected product version, allowing users to query the database for all CVEs connected to a specific product version, product, or vendor. It was necessary to concatenate vendor and product names with the version strings given in the NVD in order to prevent unrelated CVEs from linking to, for example, a single ProductVersion node named "16.1" which would, in turn, link back to two different products and vendors. Instead, they would link to different nodes with names such as "vendor-1-product-a-16.1" and "vendor-2-product-b-16.1". These strings should not be difficult to synthesize as part of query for CVEs affecting very specific product versions. We chose the attack vector as our next node type, on the grounds that this would also serve as a very useful starting point for a query. As the CVSS 3.0 specification defines only four attack vectors — Network, Adjacent (meaning the exploit can not be performed across an OSI layer 3 boundary), Local, or Physical — these would serve as "supernodes" which could reflect paths along which attackers could chain exploits. The remaining CVE data, including the expanded versions of the data encapsulated in the "vector string" was assigned to properties of the CVE nodes, including attack complexity, availability impact, base score, base severity, confidentiality impact, exploitability score, impact score, integrity impact, privileges required, score, and user interaction requirements. All of these properties can be used to query CVEs, but did not apply to establishing a series of links between CVEs to model multi-stage attacks. Figure 3 shows a version of our schema, excluding these properties, with five node types (AttackVector, CVE, ProductVersion, Product, and Vendor) and four edge types (ATTACKABLE_THROUGH, AFFECTS, VERSION_OF, and MADE_BY).

With regard to including the CVSS 2.0 data in the schema, we wanted to strike a balance between query usability, utility in broad searches over many years of data, and fidelity to both CVSS specifications. Prioritizing usability meant that we did not want to introduce multiple copies of a CVE with synthesized IDs (e.g. "CVE-2017-9317-v2"), or "CVSS version" nodes that would have to be specified as additional relationships in almost any query, without necessarily adding value. As to utility and fidelity, we wanted to make it possible to use broad or narrow definitions for the semantic underpinnings of a node without making either definition more conceptually cumbersome. For example, one user might want to explicitly consider only "Local" attack vectors as defined in 3.0, while another may want to aggregate all vulnerabilities with local attack vectors over a decade for a specific software package, and we wanted both queries, with regard to identifying the right nodes and edges for the purpose at hand, to be equally legible where possible. Our schema therefore combines the "access vector" of CVSS 2.0 with the "attack vector" of CVSS 3.0 and creates two relationships between the CVE and its attack vector, each relationship given a property with the relevant

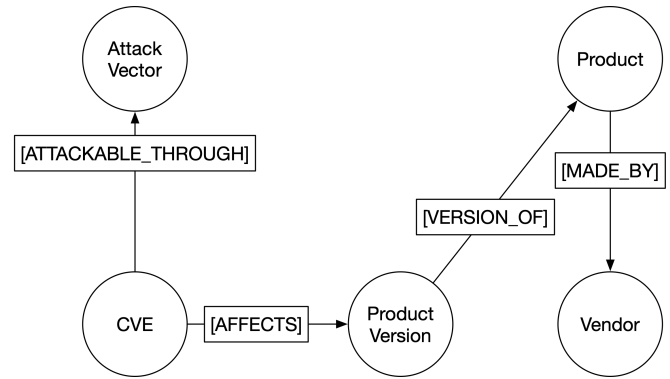


Figure 3: Schema Diagram

version number. In practice, this means that the denotation of a node can be changed at query time without introducing superfluous signifiers. For all other CVSS 3.0 and 2.0 fields, they have been attached as properties to the CVE node itself, prefixed with 'v2' or 'v3' as appropriate.

Finally, we created indexes on almost all of the properties, excluding only the text description of the CVE. Our reasoning for this was that this graph will be almost exclusively read-oriented after the initial data load, so the increased load time to create indexes on so many fields will pay off in practice, when those indexes can be used to very quickly identify all CVEs which grant admin privileges, or do not require user interaction, rather than requiring the Neo4j engine to scan all CVE nodes in its first pass.

B. Loading Data

A simple script, written in Python, is used to load the contents of NVD's data feeds page ¹, identify the URLs for the complete, compressed data, grouped by year, download it to the running Docker container, decompress it, and load the resulting JSON to Neo4j. The data load process takes advantage of the open-source APOC package to load the NVD JSON file, iterating over the individual CVEs and creating nodes or links between nodes as necessary. Only one transformation is applied to the data, translating the CVE's publication date string into a Unix epoch date integer to simplify querying over dates on the version of Neo4j (3.3.3) in use. APOC also provides facilities for transforming epoch dates back into human-readable strings, so this transformation is not inimical to our ease-of-use goal for this project. See the GitHub repository ² for all source used in this project, setup instructions, and sample queries.

IV. PERFORMANCE

A. Example Queries

The following queries were run using version 3.3.3 of Neo4j, on the official 3.3.3 Docker image, with version 3.3.0.4 of the APOC package for Neo4j. First, we present three relatively simple queries that may be useful in security

¹<https://nvd.nist.gov/vuln/data-feeds>

²<https://github.com/jazwiecki/neo4j-cve>

research, and then a more complicated query which takes advantage of the Neo4j's native graph support.

1) *Calculate the number of vulnerabilities found in the same software with the same impact score:* We present two versions of this query, one which is product-specific (in this case, Microsoft Edge), and one which will return all products in the NVD. For both, we not only include the product name, but also the vendor listed in the NVD, as there exist different products with identical names made by different organizations in the NVD. For both, we use the CVSS 2.0 impact score, which is available for more CVEs. To use version 3.0, the only difference would be changing the string "v2" to "v3" in the "RETURN" statements below.

```
1 MATCH
2   (vendor:Vendor {name: 'microsoft'})
3   -[:MADE_BY]-
4   (product:Product {name: 'edge'})
5   -[:VERSION_OF]-
6   (product_version:ProductVersion)
7   -[:AFFECTS]-
8   (cve:CVE)
9 RETURN cve.'v2.impact_score', count(cve)
```

The results of the above query will group the total number of CVEs for each distinct impact score. To return the above data for all CVEs, grouped by vendor and product, it is necessary only to remove the "name" property specifications on the Vendor and Product nodes, and include their respective names in the return values:

```
1 MATCH
2   (vendor:Vendor)
3   -[:MADE_BY]-
4   (product:Product)
5   -[:VERSION_OF]-
6   (product_version:ProductVersion)
7   -[:AFFECTS]-
8   (cve:CVE)
9 RETURN
10  vendor.name,
11  product.name,
12  cve.'v2.impact_score' AS impact_score,
13  count(cve) AS vulnerabilities
14 ORDER BY vendor.name, product.name, impact_score
```

2) *Calculate the number of vulnerabilities found in the same software with the same impact score across two consecutive years in all the possible years reported:* This query is slightly more complicated, because we have to produce a sliding window covering two years, which advances one year at a time, then group our results by each of those windows. As above, we first present query results for a single product, and subsequently for all CVEs. It uses APOC's date functions, as well as the UNWIND command, which transforms the list returned by RANGE into a series of rows. Date windows are created by taking the specified year, creating an ISO 8601 date string from it, and then converting it to a UNIX epoch timestamp, the same format as used for the CVE publish dates in our schema.

```
1 UNWIND RANGE(1988, 2018, 1) AS t
2 WITH
3   apoc.date.fromISO8601(t + '-01-01T00:00:00.000Z')
4   AS start_window,
5   apoc.date.fromISO8601((t + 1) + '-12-31T23
6   :59:59.999Z')
```

```
6   AS end_window
7 MATCH
8   (vendor:Vendor {name: 'microsoft'})
9   -[:MADE_BY]-
10  (product:Product {name: 'edge'})
11  -[:VERSION_OF]-
12  (product_version:ProductVersion)
13  -[:AFFECTS]-
14  (cve:CVE)
15 WHERE
16   cve.published >= start_window
17   AND
18   cve.published <= end_window
19 RETURN
20   apoc.date.format(start_window)
21   AS start_window,
22   apoc.date.format(end_window)
23   AS end_window,
24   cve.'v2.impact_score'
25   AS impact_score,
26   count(cve)
27   AS vulnerabilities
28 ORDER BY start_window
```

As above, in order to run this query across all CVEs, we have only to remove the specific names in the MATCH statement, and return those same properties in our RETURN statement. Note that this query demonstrates the kind of broad all-time queries we could not run if the CVSS v2 properties were not included for all CVEs.

```
1 UNWIND RANGE(1988, 2018, 1) AS t
2 WITH
3   apoc.date.fromISO8601(t + '-01-01T00:00:00.000Z')
4   AS start_window,
5   apoc.date.fromISO8601((t + 1) + '-12-31T23
6   :59:59.999Z')
7   AS end_window
8 MATCH
9   (vendor:Vendor)
10  -[:MADE_BY]-
11  (product:Product)
12  -[:VERSION_OF]-
13  (product_version:ProductVersion)
14  -[:AFFECTS]-
15  (cve:CVE)
16 WHERE
17   cve.published >= start_window
18   AND
19   cve.published <= end_window
20 RETURN
21   apoc.date.format(start_window)
22   AS start_window,
23   apoc.date.format(end_window)
24   AS end_window,
25   vendor.name,
26   product.name,
27   cve.'v2.impact_score'
28   AS impact_score,
29   count(cve)
30   AS vulnerabilities
31 ORDER BY start_window
```

3) *Find vulnerabilities that require user interaction through the network:* Although it may seem like an abstract distinction, this query demonstrates the utility of using two relationships between attack vectors and CVEs for researchers who may be focused on the precise differences between CVSS 2.0's "access vector" and the renamed, slightly different 3.0 "attack vector" definition.

This query will use either definition for the ATTACKABLE_THROUGH relationship, and returns 37,698

results:

```

1 MATCH
2   (attack_vector:AttackVector {name: 'NETWORK'})
3   -[:ATTACKABLE_THROUGH]-
4   (cve:CVE {'v2.user_interaction_required': TRUE
5   })
6 RETURN count(cve)

```

This query, on the other hand, will only return CVE nodes considered network-attackable under CVSS 3.0, and it returns 9,104 results.

```

1 MATCH
2   (attack_vector:AttackVector {name: 'NETWORK'})
3   -[:ATTACKABLE_THROUGH {cvss_version: 3}]-
4   (cve:CVE {'v3.user_interaction': 'REQUIRED'})
5 RETURN count(cve)

```

4) *Find easy exploits allowing scope changes that simplify access to high-impact exploits which are otherwise high-complexity:* All of the preceding queries would be relatively uncomplicated to implement in a relational database with a suitable schema. Where a graph database truly outperforms a relational database is by rapidly traversing a graph of connected nodes to find different nodes of the same type without requiring multiple expensive self-referential joins. In one classic example, a relational database with one table of people, and a one-to-many table listing the relationships between those people, retrieving information about all individuals two network hops from a given starting individual would require two joins for each hop, with the query expense scaling exponentially for each hop. Given that modern security environments must defend against the multi-stage attacks discussed earlier, improving the execution time (and conceptual simplicity) of queries which could uncover these paths is very valuable.

Below, we demonstrate a trivial query which could be used to identify such multi-stage attack vectors. We begin with CVEs vulnerable to network attacks, requiring no user interaction, low complexity, and a scope change. Scope changes represent vulnerabilities which can impact resources beyond the means or privileges of the vulnerable system. CVSS v3 introduced the scope change property specifically to enable analysis of escalating chains of vulnerabilities, but in its current form, the NVD does not support exploring the consequences of these escalations.

The "easy" CVEs with which we begin are then used to query their affected product version, and those are in turn connected back to our target CVEs, those with high impacts, requiring high privileges, no user interaction, and local access. For the purposes of this example, we assume a scope change made through the network effectively grants LOCAL access with high privileges. As of this writing, our trivial query returns 2,478 CVEs. A relational database representation of the NVD would have required twelve joins, assuming that, as the underlying data demonstrates, the `AttackVector` and `ProductVersion` nodes would be represented as one-to-many relationships of three tables each, one for each node, and one for the relationship. Considering that a more sophisticated multi-stage attack might involve many times as many hops, it is easy to see that the chain

of relational table joins would quickly become unworkably slow, and far more complicated.

```

1 MATCH
2   (attack_vector:AttackVector {
3     name: 'NETWORK'
4   })
5   -[:ATTACKABLE_THROUGH]-
6   (cve:CVE {
7     'v3.scope': 'CHANGED',
8     'v3.user_interaction': 'NONE',
9     'v3.attack_complexity': 'LOW'
10    })
11   )
12   -[:AFFECTS]-
13   (product_version:ProductVersion)
14   -[:AFFECTS]-
15   (escalated_cve:CVE {
16     'v3.privileges_required': 'HIGH',
17     'v3.user_interaction': 'NONE',
18     'v3.integrity_impact': 'HIGH'
19   })
20   )
21   -[:ATTACKABLE_THROUGH]-
22   (escalated_vector:AttackVector {name: 'LOCAL'})
23 RETURN cve.name,
24        product_version.name,
25        escalated_cve.name

```

V. CONCLUSION

We have shown a graph database model for representing the NVD, and that there exists graph queries which can be used in security research which are orders of magnitude more efficient than their relational counterparts for use in evaluating multi-stage threats. This model is better-adapted for evaluating real-world security risks in a given environment, allowing for the efficient creation of practical attack graphs and facilitating vulnerability research. As shown in the example queries, this relationship graph allows for easy queries along with complex multi-stage queries. In moving the NVD data from the NIST website, we unlock a greater potential for research and analysis, and by delivering working code that runs in an isolated environment, we do so in a way that can be easily used in high-security environments.

VI. FUTURE WORK

In the next version of our proposed schema, we suggest changing `MADE_BY` to `MAKER_OF` in `Vendor` and `Product` nodes. This is a minor change to the schema to allow more consistency with the semantics of other relationships, and make queries easier to construct and understand. Additionally, we would add `Common Weakness Enumeration (CWE)` nodes and replace the existing `ProductVersion-Product-Vendor` chains with nodes based on the `Common Platform Enumeration (CPE)` URIs in the NVD. `CWE` information attached to CVEs represents a link to well-defined higher-level vulnerability patterns (e.g. "Information Exposure Through Log Files") which could be useful for identifying common patterns in particular software packages delivered by particular vendors. `CPE` URIs are structured naming scheme for information technology systems, software, and packages. Based off URIs, they offer more complete vendor information than that found in the NVD's JSON

representation of its affected product data. Some existing CVEs include detailed, accurate CPE URIs but no affected product data. Incorporating CPE URIs will require carefully planned schema changes, as they have Boolean operators to cover combinations of configurations between software, firmware and hardware vulnerabilities. A final change to the schema we suggest is to include the various CVSS score fields, divided by 10 to ensure they are in the range 0-1. This would convert them to probabilities, allowing for the graph to be used as part of a dynamic Bayesian network for calculating final probabilities of multi-stage attack exploits. Finally, the current data loader has no facilities for processing NIST's "update" feed or adding new vulnerabilities for the current year, rather than forcing users to re-run a complete refresh of the entire NVD, a change which would improve update speeds.

Continued efforts towards a comprehensive solution of vulnerability analysis could include hosting a centralized, online database to allow researches to run queries without having to clone the NVD contents locally. Additionally, allowing new CVEs to be reported directly to this hosted database would allow for quicker turnaround in time-to-report metrics. This change would mean this database would be the canonical source of truth for vulnerabilities. If created in this manner, any improvements to reporting, schema, or analysis would benefit all users.

APPENDIX I

FREQUENCY OF USES OF "MULTI-STAGE ATTACKS"

Table I lists the total number of papers indexed by Google Scholar [Goo18] which mention the term "multi-stage attacks", collected on December 2nd, 2018. The year 2018 was not included as it is not complete, but there are 78 citations as of December 2nd, 2018.

TABLE I: Mentions per Year

Year	Papers
2017	71
2016	92
2015	75
2014	49
2013	55
2012	51
2011	41
2010	32
2009	29
2008	36
2007	35
2006	38
2005	15

REFERENCES

[CGMP98] Junghoo Cho, Hector Garcia-Molina, and Lawrence Page. "Efficient crawling through URL ordering". In: *Computer Networks and ISDN Systems* 30.1-7 (1998), pp. 161–172.

[RA00] R. W. Ritchey and P. Ammann. "Using model checking to analyze network vulnerabilities". In: *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium*. IEEE. 2000, pp. 156–165.

[MG02] Peter Mell and Tim Grance. "Use of the common vulnerabilities and exposures (cve) vulnerability naming scheme". In: *NATIONAL INST OF STANDARDS AND TECHNOLOGY GAITHERSBURG MD COMPUTER SECURITY DIV* (2002).

[HH05] Simon Hansman and Ray Hunt. "A taxonomy of network and computer attacks". In: *Computers & Security* 24.1 (2005), pp. 31–43.

[OBM06] Xinming Ou, Wayne F Boyer, and Miles A McQueen. "A scalable approach to attack graph generation". In: *Proceedings of the 13th ACM conference on Computer and communications security*. ACM. 2006, pp. 336–345.

[Han07] David J Hand. "Principles of data mining". In: *Drug safety* 30.7 (2007), pp. 621–622.

[LWJ08] A. Singhal L. Wang C. Yao and S. Jajodia. "Implementing interactive analysis of attack graphs using relational databases". In: *Journal of Computer Security* 16.4 (2008), pp. 419–437.

[ZNW10] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. "Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista". In: *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. IEEE. 2010, pp. 421–428.

[Mul+11] Varish Mulwad et al. "Extracting information about security vulnerabilities from web text". In: *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology-Volume 03*. IEEE Computer Society. 2011, pp. 257–260.

[SO11] A. Singhal and X. Ou. *Security Risk Analysis of Enterprise Networks Using Probabilistic Attack Graphs*. NIST Interagency Report 7788. 2011. URL: https://www.nist.gov/publication/get_pdf.cfm?pub_id=907935.

[ZCO11] Su Zhang, Doina Caragea, and Xinming Ou. "An empirical study on using the national vulnerability database to predict software vulnerabilities". In: *International Conference on Database and Expert Systems Applications*. Springer. 2011, pp. 217–231.

[AT14] Sayyed Muzammil Ali and Ms RR Tuteja. "Data Mining Techniques". In: (2014).

[BM14] M.S. Barik and C. Mazumdar. "A graph data model for attack graph generation and analysis". In: 2014.

- [DE18] DB-Engines.com. *DBEngines Ranking of Graph DBMS*. 2018. URL: <https://db-engines.com/en/ranking/graph+dbms>.
- [Fra+18] Nadime Francis et al. “Cypher: An evolving query language for property graphs”. In: *Proceedings of the 2018 International Conference on Management of Data*. ACM. 2018, pp. 1433–1445.
- [Goo18] Google. *Google Scholar*. 2018. URL: <https://scholar.google.com/>.
- [Neo18] Neo4j. *The Neo4j Developer Manual v3.4*. 2018. URL: <https://neo4j.com/docs/developer-manual/current/>.
- [NIS18] NIS. *National Vulnerabilities Database*. 2018. URL: <https://nvd.nist.gov/>.
- [Rod+18] Luis Gustavo Araujo Rodriguez et al. “Analysis of Vulnerability Disclosure Delays from the National Vulnerability Database”. In: *Workshop de Segurança Cibernética em Dispositivos Conectados (WSCDC_SBRC)*. Vol. 1. 2018.