Graph-Based CPE Matching for Identification of Vulnerable Asset Configurations

Daniel Tovarňák
Institute of Computer Science
Masaryk University
Brno, Czech Republic
tovarnak@ics.muni.cz

Lukáš Sadlek
Institute of Computer Science
Masaryk University
Brno, Czech Republic
sadlek@mail.muni.cz

Pavel Čeleda
Institute of Computer Science
Masaryk University
Brno, Czech Republic
celeda@ics.muni.cz

***Abstract*—In this manuscript, we propose a graph-based approach for identification of vulnerable asset configurations via Common Platform Enumeration matching. The approach consists of a graph model and insertion procedure that is able to represent and store information about CVE vulnerabilities and different configurations of CPE-classified asset components. These building blocks are accompanied with a search query in Gremlin graph traversal language that is able to find all vulnerable pairs of CVEs and asset configurations in a single traversal, as opposed to a conventional brute-force approach.**

***Index Terms*—Common Vulnerabilities and Exposures, Common Platform Enumeration, CVE, CPE, graph model, Gremlin**

I. Introduction

Vulnerability management attempts to discover, analyze, and mitigate vulnerabilities in organization's critical assets to minimize potential business loss to the lowest acceptable level [1]. It is often maintained that majority of attacks is enabled through zero-day vulnerabilities. However, according to Verizon's DBIR report, internet-facing hosts susceptible to new vulnerabilities seem to be more likely to be also defenseless against many older vulnerabilities [2]. Several additional reports, e.g. [3], show that the exploitable vulnerabilities of internet-facing systems can be even 20 years old. Thus, in order to protect critical assets, it is usually necessary to identify vulnerabilities with respect to the whole CVE list, which at this time contains more than one hundred thousand records.

At the same time, the granularity of described and managed asset components, can go as low as to individual software libraries [4], creating complex dependency configurations, much more granular than a typical asset configuration of a *Firefox 43.0, running on top of Windows 10 v2004 en_US, on top of Intel Xeon processor, inside a Dell Laptop*.

When put in contrast with the rate of change in modern infrastructures and the ever-growing number of vulnerabilities, a traditional CPE-based iterative (brute-force) matching of vulnerable assets might become time or cost prohibitive. Especially so, if we would require an assessment of the defended infrastructure with every change in its configuration.

II. Background and Related Work

Common Vulnerabilities and Exposures [5] is a de-facto standard for unambiguous identification and description of vulnerabilities in computer systems. Each CVE is assigned a unique identifier, short description, and at least one publicly available reference. CVE list of publicly disclosed vulnerabilities currently contains approximately 148 thousand records with the participation of a great amount of IT vendors.

Common Platform Enumeration [6] is a standardized method of describing and identifying classes of applications, operating systems, and hardware devices present among an enterprise's computing assets. Each asset (e.g. desktop computer running an application) can be classified via individual CPE names of its respective *components*, forming a particular *asset configuration*. The CPE stack provides means for creating logical expressions about classes of such configurations to support automated decisions regarding the assets.

National Vulnerability Database [7] is a U.S. government repository build upon the records of the CVE list, which it heavily extends with additional metadata and provides in the form of periodically update data feeds. Most importantly, it contains information about impact, severity and other vulnerability scoring information using the Common Vulnerability Scoring System [8], reference to the weakness category using Common Weakness Enumeration [9], and last, but not least, a list of CPE *applicability statements*, which describe classes of vulnerable asset configurations. This valuable CPE information can be typically used to iteratively match against a collection of known asset configurations.

Graph-based methods can be applied to address many issues in domains with highly interconnected data, including cybersecurity. A well-known method for representation of multi-step attacks including vulnerability exploits are attack graphs. This graph-based method depicts attack paths that the attacker can take to reach malicious goals [10]. When looking at opposite side of the equation, graph models can be used to assess impact of vulnerability exploits on organization by incorporating graph-based mission modelling, i.e. mapping of cyber assets to enterprise missions [11].

The focus of this paper is to follow suit, and apply graph-based methods to the domain of asset vulnerability identification. The goal is to optimize the exhaustive brute-force approach for assessment of all the known CVE vulnerabilities against a collection of known asset configurations by iteratively matching all the related CPE applicability statements.

978-3-903176-32-4

III. Common Platform Enumeration

The CPE stack consists of several modular specifications that are build upon each other. By properly combining them, it enables its users to perform various functions in the area of vulnerability and asset identification [6], [12], [13], [14].

A. CPE Naming

The Naming specification defines the foundational construct of Common Platform Enumeration – a well-formed name (WFN). *Well-formed name* is a logical data construct, rather than a machine-readable representation, used to describe or identify a software application, operating system, or hardware device. It takes the form of an unordered set of *attribute-value pairs*, where an *attribute-value pair* is a tuple $A = V$ in which A (the attribute) is an alphanumeric label (used to represent a property or state of some entity), and V (the value) is the value assigned to the attribute. Note that WFNs are strictly used to describe and identify product classes, e.g. laptops of a particular vendor. Thus, they are not used to identify specific asset instances, e.g. an actual laptop sitting on a desk. [6]

1) Attributes: The specification permits the following attributes to be present in a WFN attribute-value pair. Each attribute can be used at most once in a WFN. If attribute is not used in a WFN, its value defaults to the logical value *ANY*.

- The **part** can contain either: **a** for application, **o** for operating systems, and **h** for hardware devices,
- **vendor** part identifies a person or a company which manufactured or created the product,
- **product** is related to the official product name,
- **version**, **update**, and **sw_edition** determine a version, an update, and an edition of the product,
- **edition** is deprecated and should always contain value *ANY* unless it is necessary backwards compatibility,
- **language** tag for the language of user interface conforming to a related RFC 5646 specification,
- **target_sw** is an operating environment of the product,
- **target_hw** specifies an architecture,
- **other** might refer to the information, which does not fit to any of the previous.

2) Attribute Values: The values for the above-mentioned attributes can be assigned one of the following value types.

- A *logical value ANY*, when there are no restrictions on acceptable values for that attribute.
- A *logical value NA*, when there is no legal or meaningful value for that attribute, or when that attribute is not used as part of the description.
- An exact *string value* in UTF-8, conforming to pre-defined restrictions, e.g. allowed characters, special wildcard characters, or per-attribute value restrictions (for example, as in the case of the **part** attribute).

3) Binding and Unbinding: Since WFN represents only an abstract canonical form of the concept, the CPE specification defines an additional two machine-readable forms of WFNs. The first form is a URI Binding, which is included in the specification for backward compatibility with prior CPE versions. The second form is a formatted string binding that is new to CPE version 2.3. The specification defines the syntax and detailed set of procedures for binding (serialization) and unbinding (de-serialization) these forms to and from WFN. Listing 1 shows an example of three WFN forms: (1) an unbound WFN in an illustrative notation used by the specification, (2) bound URI, and (3) bound formatted string.

```
wfn:[part="o", vendor="microsoft",
↪  product="windows_server_2008", version="r2", update="sp1",
↪  edition=ANY, language=ANY, sw_edition=ANY, target_sw=ANY,
↪  target_hw="itanium", other=ANY]
cpe:/o:microsoft:windows_server_2008:r2:sp1:~~~~itanium~
cpe:2.3:o:microsoft:windows_server_2008:r2:sp1:*:*:*:*:itanium:*
```

Listing 1. CPE name in its unbound (WFN) and bound forms

B. CPE Name Matching

The Name Matching specification defines the procedures for comparing two WFNs to each other (source and target) in order to determine if they describe or identify the same software application, operating system, hardware device. By logically comparing the WFNs as sets of attribute-value (AV) pairs, it can be determined if common set relations hold. The matching process can determine if the source and target CPE names are *EQUAL* ($=$), if the source is a *SUBSET* ($\subset$) or *SUPERSET* ($\supset$) of the target, or if they are *DISJOINT* ($\neq$).

Table I
Possible AV Pair Comparison Results [12]

No.	Source AV pair	Target AV pair	AV-Relation
1	*ANY*	*ANY*	$=$
2	*ANY*	*NA*	$\supset$
3	*ANY*	i	$\supset$
4	*ANY*	m^+	undefined
5	*NA*	*ANY*	$\subset$
6	*NA*	*NA*	$=$
7	*NA*	i	$\neq$
8	*NA*	m^+	undefined
9	i	i	$=$
10	i	k	$\neq$
11	i	m^+	undefined
12	i	*NA*	$\neq$
13	i	*ANY*	$\subset$
14	m^+	i	$\supset$ or $\neq$
15	m^+	*ANY*	$\subset$
16	m^+	*NA*	$\neq$
17	m_1^+	m_2^+	undefined

The source-to-target WFN comparison can be seen as a two-phase process. First, a sequence of pairwise AV comparisons is conducted, yielding a list of results, followed by a holistic evaluation of the results list to arrive at an overall determination of the set-theoretic relationship between source and target [12].

1) Pairwise Attribute-Value Comparison: The first phase compares each AV pair in the source WFN to its corresponding AV pair in the target WFN (their values), yielding one of the four possible set relations. Considering the allowed values listed below, Table I shows all the possible results of this phase.

1) *ANY* and *NA* are the logical values, as already described,
2) i is a string value without wildcards, e.g. `foo`,
3) k is a string value without wildcards ($i \neq k$), e.g. `bar`,
4) m^+ is a string value with wildcards, e.g. `**b??`.

2) CPE Name Comparison: In the second phase, the resulting AV-Relations (AV-Rs) are analysed in order to determine an overall name comparison result. There are four possible situations, which can happen. The *name relation* is undefined otherwise.

1) If any AV-R is *DISJOINT*, then the result is *DISJOINT*.
2) If all AV-Rs are *EQUAL*, then the result is *EQUAL*.
3) If all AV-Rs are *EQUAL* or *SUPERSET*, then the result is *SUPERSET*.
4) If all AV-Rs are *EQUAL* or *SUBSET*, then the result is *SUBSET*.

C. CPE Dictionary

The Dictionary specification defines the concept of a CPE dictionary, i.e. a repository of CPE names and associated metadata. Each CPE name in the dictionary identifies a single class of some IT product in the world. The specification lays groundwork for an official Dictionary, thanks to which the entities within the IT industry are provided with a standardized way to describe and identify IT products. The specification defines the concept itself, and the rules and policies relating to dictionary instantiation and management. [13]

D. CPE Applicability Language

The Applicability Language specification defines a standardized way to describe IT platforms by forming complex logical expressions out of individual CPE names (and external checks, which are, however, not applicable for our purposes). For example, CPE Applicability Language could combine the CPE name for an operating system (such as Microsoft Windows XP) *AND* the CPE name for an application running on that operating system (such as Mozilla Firefox 48.0). These logical expressions are called *applicability statements* and they represent complex Boolean formulas, possibly deeply nested, with allowed negations. Considering some generic *fact reference*, i.e. a reference to a bound source CPE name, taking the form of a propositional variable R_n^{CPE}, Expression 1 is a good example of some applicability statement.

$$(R_1^{\text{CPE}} \wedge R_2^{\text{CPE}} \wedge \neg(R_3^{\text{CPE}} \vee R_4^{\text{CPE}})) \vee \neg R_5^{\text{CPE}} \qquad (1)$$

The spec. defines a simple algorithm that is able to determine, if a specific applicability statement E is $true$ for a set of known target CPE names K. Note that a single fact reference can be evaluated to $true$ only if the corresponding name comparison with some target evaluates to *SUPERSET*. [14]

IV. Graph-Based CPE Configuration Matching

At this point of the manuscript, several things should be apparent. First, the use of Common Platform Enumeration stack is far from trivial, yet, this is balanced out by a very good and detailed specification. Second, in order to determine, if a single CVE vulnerability is present in a collection of known asset configurations, each applicability statement present in the CVE record must be evaluated against every asset configuration in that collection. Also, vice-versa, in order to identify a vulnerability of a single asset configuration, it must be evaluated against every applicability statement of every known CVE vulnerability. For a very large number of vulnerabilities and asset components, this can become prohibitive when executed for every change in the collection of CVEs or assets.

In the pursuit of a more efficient approach, we present a graph-based approach for the problem of CPE configuration matching that can be seen as having three main building blocks.

1) A *graph model* that is able to represent and store (a) information about CVE vulnerabilities and their related applicability statements, and (b) information about tree-like hierarchies of asset components and their related CPE names, i.e. asset configurations.
2) An *insertion procedure* that is able to pre-process CVE and asset data structures, insert them as vertices into a graph, whilst performing AV pair comparison during the insertion time.
3) A *graph search query* that is able to find all vulnerable pairs of CVEs and asset configurations in a single traversal. At the same time, it is able to find a set of assets with a given vulnerability and vice-versa.

A. Graph Model

As with majority of graph-based approaches, an appropriate and elegant graph model can significantly reduce the relative complexity of the subsequent tasks. There are two core principles at play that the created graph model leverages. First, we had to cope with an arbitrary complexity of the applicability statements, which allow for negations (logical complements) to be used. Second, the model had to allow for the vulnerability search to take the form of a straightforward graph traversal.

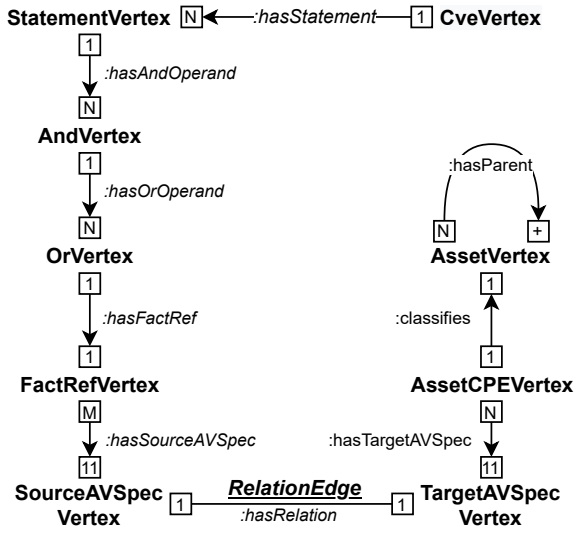

Figure 1. Graph model for CPE configuration matching

We have solved the first problem by always assuming that the applicability statements are always in Conjunctive Normal Form, i.e. it is a *conjunction* ($\wedge$) consisting of one or more *clauses*, each of which is a disjunction ($\vee$) of one or more literals. The logical complement ($\neg$) can be present only for literals. Note that every Boolean formula can be converted into

an equivalent CNF formula. The second problem was solved by evaluating the respective AV pair comparisons at the time of insertion. The resulting model (Figure 1) is described below.

- `CveVertex` represents an individual CVE vulnerability and its metadata. Each vertex can have multiple outbound `StatementVertices`.
- `StatementVertex` represents an applicability statement (Boolean logic formula) in a Conjunctive Normal Form. Each vertex can have multiple outbound `AndVertices`.
- `AndVertex` represents a single clause in a CNF formula conjunction. Each vertex can have multiple outbound `OrVertices`.
- `OrVertex` represents a single positive or negative literal in a CNF clause. Each vertex has exactly one `FactRefVertex`.
- `FactRefVertex` represents a fact reference to a bound CPE name. It can be seen as a related propositional variable for a CNF literal. Each vertex has 11 outbound AV pairs.
- `SourceAVSpecVertex` represents a single AV pair on the source side of the attribute comparison.
- `RelationEdge` represents the result of a pairwise attribute comparison and stores the set relation type.
- `TargetAVSpecVertex` represents a single AV pair on the target side of the attribute comparison.
- `AssetCPEVertex` represents a definition of a bound CPE name classifying a particular `AssetVertex`. Each vertex has 11 outbound AV pairs.
- `AssetVertex` represents an asset component and its metadata. Each vertex can have at most one parent.

B. Insertion Procedure

The proposed graph model, and related search traversal, work thanks to the way the input data are pre-processed and inserted. The insertion of a new `CveVertex` (`AssetVertex`) works in a recursive manner with respect to the vertex insertion operation. Each new vertex triggers a creation of its connected vertices, up until the `SourceAVSpecVertices` (`TargetAVSpecVertices`) are created (or only connected, if they exist), and the AV pair comparison results are stored in the adjacent `RelationEdges`. Note that the deletion of the vertices would work in a similar cascading manner (reversed in sequence). The most important steps of the insertion procedure include the following.

1) For every new CVE vulnerability, `CveVertex` is created and every applicability statement it contains, is converted into CNF. Adjacent `StatementVertices`, `AndVertices`, `OrVertices`, and `FactRefVertices` are recursively inserted.
2) When a new `FactRefVertex` is created, its CPE name is unbound into WFN and exploded into 11 AV pairs (possible future `SourceAVSpecVertices` from Step 3.).
3) If a corresponding `SourceAVSpecVertex` exists in the graph, it is merely connected to the `FactRefVertex`. If it does not exist, it is first created.
4) When a new `SourceAVSpecVertex` is created, all the existing `TargetAVSpecVertices` for the same WFN attribute are listed, and a source-target attribute comparison is performed for each one. The resulting set relation types are then stored in newly created `RelationEdges`.
5) For every new asset component, `AssetVertex` is created, and its (optional) parent is set. Adjacent `AssetCPEVertex` is created for the asset's CPE name.
6) When a new `AssetCPEVertex` is created, its CPE name is unbound into WFN and exploded into 11 AV pairs. Steps equivalent to steps 3. and 4. are followed, but this time for `TargetAVSpecVertices`. (See Listing 2 for illustration.)

```
def AssetVertex(uid, asset):  # uid = asset.uuid
    a = G.createScopedV(uid, asset, "AssetVertex")
    if asset.parentUUID:
        p = G.getV(asset.parentUUID, "AssetVertex")
        if p: a.setLinkOut('hasParent', p)
    aw = getWFN(asset.cpeName)
    a.newLinkIn('classifies', AssetCpeVertex(aw))
    return a

def AssetCpeVertex(aw):
    x = G.createUnscopedV(aw, "AssetCPEVertex")
    avPairs = getAVPairs(aw)
    for avp in avPairs:
        uid = (avp.name, avp.val)
        p = G.getIfExistsV(uid, "TargetAVSpecVertex") \
            .else_(TargetAVSpecVertex(uid, avp))
        x.newLinkOut('hasTargetAVSpec', p)
    return x

def TargetAVSpecVertex(uid, avp): # uid = (avp.name, avp.val)
    t = G.createScopedV(uid, avp, "TargetAVSpecVertex")
    srcs = G.findV("attribute", avp.name, 'SourceAVSpe...')
    for s in srcs:
        e = t.newLinkBoth('hasRelation', s, 'RelationEdge')
        e.setRelationType(compareAVPair(s.avp, t.avp))
    return t
```

Listing 2. Pseudo-code for `AssetVertex` creation

C. Search Query

We have decided to model the search query with the help of the *Gremlin* graph traversal language (domain-specific language). The first reason for this is Gremlin's independence from a particular graph system (database) implementation. This means that the resulting query can be executed over any supporting graph computing system such as an OLTP graph database and/or an OLAP graph processor. Second, Gremlin supports both the imperative traversal queries and the declarative pattern-match queries within the same framework, which allows for powerful search and reasoning [15]. The modelled search query for CPE configuration matching actually benefits from the combination of the two styles.

Gremlin is a graph traversal machine and language, designed, developed, and distributed by the Apache TinkerPop project [16]. As a *graph traversal machine*, it is composed of three components: a graph G (data), a traversal Ψ (instructions), and a set of traversers T (read/write heads). Conceptually, a collection of traversers in T move about G according to the instructions specified in Ψ. The computation is complete when either a) there no longer exists any traversers in T or b) all existing traversers no longer reference an instruction in Ψ (i.e. they have halted). As a *graph traversal language*, it is a functional language with the purpose enabling a human user to easily define Ψ and thus, program a Gremlin machine. The simplicity of Gremlin's grammar enables it to be embedded in

several programming languages. Thus, for a developer, there is no discontinuity between the software code and the graph analysis code. Some common traversal steps of the Gremlin framework include the following [15].

- `out("label")`: Move to the outgoing adjacent vertices given the edge labels.
- `in("label")`: Move to the incoming adjacent vertices given the edge labels.
- `bothE("label")`: Move to both the incoming and outgoing incident edges given the edge labels.
- `hasLabel("label")`: Remove the traverser if its element does not have any of the labels.
- `hasId(<predicate P>)`: Remove the traverser if the element id does not satisfy the given predicate.

The main search traversal query is shown in Listing 3, and, when needed, it references related helper traversals and predicates, which can be seen in Listing 4.

```
QUERY = g.V()
.hasLabel("type::CveVertex").hasId(P1).as_("x_Cve") ①
.out("hasStatement").as_("x_Statement")
.out("hasAndOperand").out("hasOrOperand").as_("x_Or")
.choose(has_("negate", False), T1, T2).as_("x_AssetCPE") ②
.dedup("x_FactRef", "x_AssetCPE")
.out("classifies").optional(Z).hasId(P2).as_("x_RootAsset") ③
.group() ④
    .by(select("x_Cve", "x_Statement", "x_RootAsset"))
    .by(select("x_Or", "x_FactRef", "x_AssetCPE").fold())
.unfold()
.project("match", "path", "expectedCount", "actualCount") ⑤
    .by(select(Column.keys)).by(select(Column.values))
    .by(C1).by(C2)
.where("expectedCount", P.eq("actualCount"))
```

Listing 3. Search Traversal in the Gremlin DSL

1) Starting the Traversal: The traversal starts in all the `CveVertices`, which can be possibly filtered by a predicate `P1` in order to limit the search to a sub-set of CVE IDs. By default, all CVE IDs are considered, since an empty `P.without([])` predicate is used. The traversal continues through the outbound edges to `StatementVertices`, then `AndVertices`, and finally to the `OrVertices`, where the traversal branches.

2) Branching of the Traversal: Since the applicability statements were converted into CNF when inserted, the only place where negation can occur, is the `OrVertex`, which represents a positive or a negative literal. The goal of this crucial step is (a) for positive literals, find such (`FactRefVertex`, `AssetCPEVertex`) pairs that evaluate to $true$, i.e. they match, or (b) for negative literals, find such (`FactRefVertex`, `AssetCPEVertex`) pairs that evaluate to $false$, i.e. don't match. As per the CPE specification, the result of such evaluation can be $true$ only if the name comparison of source `FactRefVertex` and target `AssetCPEVertex` evaluates to *SUPERSET*.

In order for such a name comparison to evaluate to *SUPERSET*, all the 11 corresponding AV-Relations (`RelationEdges`) must be either of type *SUPERSET*, or *EQUAL*, as per the CPE specification. The helper traversal `T1` does this by considering only such (`FactRefVertex`, `AssetCPEVertex`) paths, that have all the 11 AV-Relations `P.within([SUPERSET, EQUAL])`. Note that the used pattern-matching `match()` step continues in the traversal only if all of its inner traversals are able to continue.

In the case of a negative literal, the name comparison must NOT be evaluated to *SUPERSET*, which means that at least one corresponding AV-Relation (`RelationEdge`) must be `P.without([SUPERSET, EQUAL])` on the path from `FactRefVertex` to `AssetCPEVertex`. This is the job of the helper traversal `T2`.

Thanks to the `choose()` step, the traversal stream now contains only such `OrVertices`, that evaluate to $true$. This is because for positive literals, only matching ($true$) paths were taken, and for negative literals, only the non-matching ($false$) paths were taken.

```
P1 = P.without([]) # or P.eq(<someCveId>)
P2 = P.without([]) # or P.eq(<someAssetUUID>)
RL = [SUPERSET, EQUALS] # desired AV pair match

M[] = CPE_ATTR_NAMES.forEachElement(_X -> # 11 WFN attributes
__.as_("x_FactRef").out("hasSourceAVSpec").has_("attrName", _X)
.bothE("hasRelation").has_("relation", P.within(RL))
.outV().in_("hasTargetAVSpec").as_("x_Same_AssetCPE")).toArr()

T1 = __.out("hasFactRef") # negate = False
.match([traverse(p) for p in M]).select("x_Same_AssetCPE")

T2 = __.out("hasFactRef").as_("x_FactRef") # negate = True
.out("hasSourceAVSpec").bothE("hasRelation")
.has_("relation", P.without(RL)).outV().in_("hasTargetAVSpec")

Z = __.repeat(out("hasParent"))
.until(outE("hasParent").count().is_(0))

C1 = __.select(Column.keys).select("x_Statement")
.out("hasAndOperand").count()

C2 = __.select(Column.values).unfold().select("x_Or")
.dedup().in_("hasOrOperand").dedup().count()
```

Listing 4. Helper Traversals in the Gremlin DSL

3) Finding the Asset Roots: The traversal continues by de-duplicating its path history with respect to the unique (`FactRefVertex`, `AssetCPEVertex`) pairs it encountered, ending up with the corresponding `AssetCPEVertices` at the traversal head. From here, the traversal follows through the `AssetCPEVertices` and their outbound edges to `AssetVertices` and does not stop until there are no further parents to continue to (helper traversal `Z`). Parent-less `AssetVertex`, serving as a tree root, induces a set of all the related `AssetCPEVertices`, i.e. it represents an asset configuration. `AssetVertices` can be filtered by the predicate `P2` much like in the the case of `CveVertices` and the predicate `P1`.

4) Grouping of the Results: At this point, the primary graph traversal essentially stops, and the whole path history is passed through a `group().by()` step. In this step, the encountered traversal history is keyed by the n-tuple of (`CveVertex`, `StatementVertex`, `AssetVertex`). Each key represents a candidate match for a particular `StatementVertex` (of some `CveVertex`) and a particular `AssetVertex` root. The value for each key is an aggregate of the encountered traversal paths projected to the `OrVertices`, `FactRefVertices`, and `AssetCPEVertices`.

5) Boolean Evaluation: The rest of the search traversal represents an evaluation of a Boolean logic CNF formula. For each grouped pair, two counts (`C1`, `C2`) are calculated. The first helper traversal `C1` represents the (expected) number of `AndVertices` (operands) for each `StatementVertex` (conjunction) that all must hold $true$, for the whole conjunction to also hold $true$. The second helper traversal `C2` represents the (actual) number of `AndVertices` that evaluate to $true$, when at least one of its `OrVertices` also evaluates to $true$. The grouped pairs are filtered, and only such results are returned, where the number of actual `AndVertices` is equal to the number of all expected `AndVertices` for a given `StatementVertex`.

By executing the traversal `QUERY`, it is possible to find not only all the vulnerable (`CveVertex`, `StatementVertex`, `AssetVertex`) tuples in a *single pass*, but at the same time, also the corresponding full-paths that are responsible for this fact. This is in contrast with the iterative approach, where all the possible combinations of applicability statements and asset configurations must be evaluated to achieve the same result.

V. Experimental Implementation

As a part of our research we have created an open-source experimental implementation of the proposed graph-based CPE configuration matching approach in Java language [17]. The implementation validates the proposed concepts and demonstrates the feasibility of the presented approach.

It leverages a Java binding of the Gremlin framework for graph manipulation and traversal. Note that Gremlin-Java is considered the canonical reference implementation of Gremlin, and serves as the foundation by which all other Gremlin language variants should emulate [16]. Our implementation takes advantage of the Object Graph Mapping approach in order to reduce the amount of boiler-plate code associated with creation and manipulation of many vertex classes. An important dependency is a *CPE 2.3 Reference Implementation*[1], which provides a Java API for creating, using, and matching CPE Names. The implementation of graph-based matching works with CVE vulnerability records in JSON, and it is fully compatible with the *JSON Schema for NVD Vulnerability Data Feed version 1.1*[2]. It also supports the NVD extension of CPE names that allows for the values of AV pair **version** to be defined as a *version range*. Individual asset components are also represented as custom JSON records.

The experimental implementation will be further developed in order to support standalone distributed graph systems. Also, our plan is to optimize the insertion procedure so that the relatively costly source-target attribute comparison is executed in a single pass, similarly to the search traversal.

VI. Conclusion and Future Work

In this paper, we have proposed a graph-based approach for identification of vulnerable asset configurations via CPE matching. The approach consists of a graph model and insertion procedure that is able to represent and store information about CVE vulnerabilities and also about hierarchies of asset components classified by CPE names, i.e. asset configurations. This is accompanied with a graph search query in Gremlin graph traversal language that is able to find all vulnerable pairs of CVEs and asset configurations in a single traversal.

In the future, we plan to perform a thorough performance evaluation of the optimized experimental implementation. Also, since the amount of information about known CVE vulnerabilities obtainable from public sources can be described as adequate, we plan to focus our work on the other side of the equation. Our plan is to research approaches for automated asset (component) discovery and CPE classification with the highest granularity possible. The goal is to be able to fully assess the state of the defended assets in an automated manner.

[1] https://pages.nist.gov/cpe-reference-implementation/

[2] https://nvd.nist.gov/General/News/JSON-1-1-Vulnerability-Feed-Release

Acknowledgements

This research was supported by the Security Research Programme of the Czech Republic 2015–2022 (BV III/1-VS) granted by the Ministry of the Interior of the Czech Republic under No. VI20202022164 Advanced Security Orchestration and Intelligent Threat Management.

References

[1] J. Muniz, G. McIntyre, and N. AlFardan, *Security Operations Center*. Cisco Press.

[2] Verizon, "2020 Data Breach Investigations Report," Tech. Rep., 2020. [Online]. Available: https://enterprise.verizon.com/resources/reports/dbir/

[3] "The Year 2020 Vulnerability statistics report," Edgescan, Dublin, Ireland, Tech. Rep., 2020. [Online]. Available: https://info.edgescan.com/vulnerability-stats

[4] J. Williams and A. Dabirsiaghi, "The unfortunate reality of insecure libraries," Contrast Security, Inc., White Paper, 2014.

[5] CVE – Common Vulnerabilities and Exposures (CVE). [Online]. Available: https://cve.mitre.org/

[6] B. A. Cheikes, D. Waltermire, and K. Scarfone, "Common Platform Enumeration: Naming Specification Version 2.3," National Institute of Standards and Technology, NIST IR 7695, 2011.

[7] NVD – General. [Online]. Available: https://nvd.nist.gov/general

[8] Common Vulnerability Scoring System v3.1. [Online]. Available: https://www.first.org/cvss/v3-1/cvss-v31-specification_r1.pdf

[9] CWE – Common Weakness Enumeration. [Online]. Available: https://cwe.mitre.org/

[10] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, ser. CCS '06. New York, NY, USA: ACM, 2006, pp. 336–345.

[11] M. Javorník, J. Komárková, and M. Husák, "Decision support for mission-centric cyber defence," in *Proceedings of the 14th International Conference on Availability, Reliability and Security*, ser. ARES '19. New York, NY, USA: ACM, 2019, pp. 34:1–34:8.

[12] M. C. Parmelee, H. Booth, D. Waltermire, and K. Scarfone, "Common Platform Enumeration: Name Matching Specification Version 2.3," National Institute of Standards and Technology, NIST IR 7696, 2011.

[13] P. Cichonski, D. Waltermire, and K. Scarfone, "Common Platform Enumeration: Dictionary Specification Version 2.3," National Institute of Standards and Technology, NIST IR 7697, 2011.

[14] D. Waltermire, P. Cichonski, and K. Scarfone, "Common Platform Enumeration: Applicability Language Specification Version 2.3," National Institute of Standards and Technology, NIST IR 7698, 2011.

[15] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proceedings of the 15th Symposium on Database Programming Languages*, ser. DBPL 2015. New York, NY, USA: Association for Computing Machinery, 2015, p. 1–10.

[16] TinkerPop Compendium. [Online]. Available: https://tinkerpop.apache.org/docs/3.4.6/

[17] D. Tovarňák, "Graph-Based CPE Matcher," Mar. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4569393