VKG2AG : Generating Automated Knowledge-Enriched Attack Graph (AG) from Vulnerability Knowledge Graph (VKG)

Md Rakibul Hasan Talukder[a], Rakesh Podder[b] and Indrajit Ray[c]
Department of Computer Science, Colorado State University, Fort Collins, Colorado, U.S.A.

Keywords: Attack Graph, Knowledge Graph, Security Vulnerability, Computer Network.

Abstract: Attack Graph (AG) analysis is a well-established technique to asses security threats in networked systems. However, traditional AGs primarily rely on coarse level vulnerability information from the Common Vulnerabilities and Exposures (CVE) repository for identifying attack paths and suggesting patch-based mitigation strategies. This approach presents significant limitations, including unavailability of patches, compatibility constraints, and system downtime, leaving security analysts without viable alternatives for optimized risk mitigation. To address this challenge, we propose two new paradigms: a novel knowledge-enriched AG framework and a Vulnerability Knowledge Graph (VKG). VKG incorporate fine-grained, structured vulnerability information that allows exploration of additional attack mitigation strategies beyond vulnerability patching in the AG analysis. We formally define VKG and AG along with algorithms for automated knowledge build-up, integration, and querying. To ensure seamless interoperability, we develop an interface that facilitates dynamic knowledge transfer between VKG and AG, enabling enhanced security reasoning without introducing interdependencies. We evaluate our methodology on a test network and demonstrate how the knowledge-driven AG can improve security decision-making by providing system administrators with adaptable, scenario-based defense mechanisms with actionable insights.

1 INTRODUCTION

Attack Graph (AG) analysis (Lallie et al., 2017) is one of the most effective ways to evaluate the underlying security posture of a large networked system. The ability to provide meaningful insights in AG analysis depends heavily on the granularity of underlying information from various sources (e.g., hosts, networks, vulnerabilities, attack frameworks), and how the information is synthesized and used. In the vast majority of state-of-the-art AG analysis techniques, vulnerability information as available from the Common Vulnerabilities and Exposure (CVE) repository is a major source of information on potential attacks and recommended patching as defensive measures. However, unavailable patches, compatibility issues of published patches with existing services, and system downtime during patching can leave decision-makers without sufficient alternative mitigation options. On the other hand, in our research, we have observed that CVE descriptions are being underutilized in many AG analysis. As Figure 1 shows, there are many hidden information in a CVE description that can provide critical insights and actionable feedback for optimizing defense strategies without necessarily relying on patching of the vulnerability. In this paper, we develop a technique to better utilize CVE description for AG Analysis.

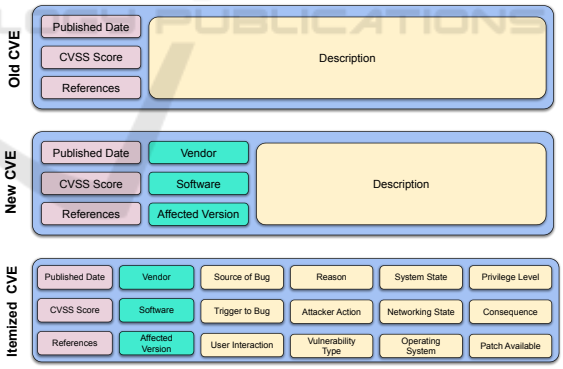

Figure 1: Potential of a CVE record (old & new).

Typically, AGs use the underlying vulnerability information that can be readily extracted from the CVE repository as attack attributes. The effective-

[a] https://orcid.org/0009-0007-7820-0853
[b] https://orcid.org/0009-0008-7394-1369
[c] https://orcid.org/0000-0002-3612-7738

Talukder, M. R. H., Podder, R. and Ray, I.
VKG2AG : Generating Automated Knowledge-Enriched Attack Graph (AG) from Vulnerability Knowledge Graph (VKG).
DOI: 10.5220/0013526700003979
In *Proceedings of the 22nd International Conference on Security and Cryptography (SECRYPT 2025)*, pages 420-428
ISBN: 978-989-758-760-3; ISSN: 2184-7711

ness of AG Analysis for alleviating the patching problem is dependent on the relevancy of the extracted information for mitigation strategies, the granularity level of the vulnerability knowledge, the automation level of organizing, accessing, and transferring knowledge, and the flexibility of interfacing with the system. There have been some works in extracting information from vulnerability descriptions using NLP techniques. Unfortunately, these requirements imposed by AG analysis have not driven the attribute extraction and utilization process.

To mitigate this, we propose a novel technique to create a Vulnerability Knowledge Graph (VKG) that synthesizes more relevant attributes extracted from CVE (shown as 'Itemized CVE' in Figure 1) that is then used to drive better AG generation and analysis. A key advantage of a separate knowledge graph is that it is independent of the AG generation process and the target system's configuration. We also propose an interface to seamlessly integrate information from the VKG into AG. Our key contributions are as follows:

1. We developed an improved, detailed vulnerability dataset, incorporating well-defined, fine-grained attributes extracted from the CVE repository to provide structured vulnerability information.
2. We designed a modular, flexible Vulnerability Knowledge Graph (VKG) that formalizes the data structure and serves as an independent, integrable source for organizing vulnerability attributes.
3. An interface between the VKG and AG has been designed with necessary algorithms to enable seamless dissemination of vulnerability attributes into AG generation automatically.
4. We evaluated the effectiveness of our approach on a test network and discussed how it can be effective for AG analysis.

2 RELATED WORK

There have been works on automating the incorporation of knowledge into Attack Graphs (AGs) in the field of automated AG generation AttacKG+ (Zhang et al., 2024) is a framework to generate attack knowledge graph via LLM techniques based on MITRE TTP definitions. But the proposed framework did not pay sufficient attention to the detail of vulnerability weakness information. There were attempts (Inokuchi et al., 2019; Jing et al., 2017) to incorporate automatic extraction of vulnerability descriptions to extend MulVal's (Ou et al., 2005) capability to generate AG. Their definition of attack status (or condition) either is not in granular details (Inokuchi et al., 2019) or has some limited fixed pre-defined values (Jing et al., 2017). In recent years, there have been few works to build knowledge graphs from vulnerability repositories. Anders et al. construct the knowledge graph by performing NER using a pre-trained model and relationship extraction between entities (Anders et al., 2023). VulKG, is constructed from vulnerability descriptions to find a weakness (CWE) chain in a graph (Yin et al., 2024). SecKG2vec provides a combination of structural and semantic embedding models to find the relationship between CVE, CWE, and CAPEC (Liu et al., 2025). However, VulKG or SecKG2vec were not developed to interface with AG, insufficient to meet the required vulnerability knowledge requirements for AG analysis.

The purpose of generating an AG is to perform various analyses, such as analyzing large networks efficiently (Ou et al., 2006), improving cyber-attack modeling (Lallie et al., 2017), enhancing scalability (Jia et al., 2015), risk analyzing (Poolsappasit et al., 2011) or providing any mitigation solutions (Ray et al., 2023; Bashir et al., 2024). However, the main drawback the security researchers or system administrators face is a structural and granular vulnerability knowledge about the system/network. This information is critical for a resource-constrained environment where performing cost-effective analysis for patching or mitigating vulnerabilities is crucial. In this paper, we address these issues and demonstrate how our proposed method can solve these problems.

3 VULNERABILITY KNOWLEDGE GRAPH (VKG)

CVE is the largest and widely used vulnerability repository, where each record represents a vulnerability in a semi-structured form. Our proposed Vulnerability Knowledge Graph (VKG) provides a more structured and automation-compatible form of vulnerabilities that preserves in-detail CVE information.

3.1 CVE Dataset for VKG

Currently, CVE records are semi-structured. The data items we can retrieve directly from CVE ('New CVE') records are: *publishing date, description, CVSS score, references, vendor, software, and affected versions*. Though recent CVEs provide *vendor, software* and *version information* as individual data items, the older ones mention them only in the description (as 'Old CVE' shown in Figure 1).

The most significant but poorly defined data item of the CVE record is *description*. It contains the most

valuable information about the vulnerability. So, we break the *description* item into multiple data items with precise definitions. We also propose a few additional data items to address a vulnerability in a more exhaustive manner. The additional items that can be extracted from CVE description are marked with • in the Table 1. Considering the new definition of a CVE record, we build the most comprehensive and in-detail vulnerability dataset, to the best of our knowledge.

We built fine-detailed dataset based on the defined data types in Table 1. We have performed prompt engineering via OpenAI Batch APIs on the pre-processed data and extracted the desired data items according to defined data types. The detailed mechanism to extract the redefined data from vulnerability description is beyond the scope of this work.

3.2 Structure of VKG

The VKG is defined as a tuple where V is a set of nodes (vertices) and E denotes relationships (edges) between those nodes.

$$VKG = \langle V, E \rangle$$

where,

$$V = \{v_1, v_2, v_3 .... v_n\}, E = \{(v_i, v_j) | v_i, v_j \in V, i \neq j\}$$

Note that throughout this paper, we will use the terms *vertex* and *node* interchangeably, so the words *relationship* and *edge*.

Assumption: The definition of the VKG is based on the dataset described in the previous sub-section 3.1. The incremental data items (as columns) in the dataset can be easily integrated into the VKG.

3.2.1 Nodes, Types, Groups

Depending on the data types (Table 1), each vertex (or node), $v \in V$ is associated with exactly one data type $t \in T$, where T is a set of all vertex types.

$$T = \{VR, S, SV, SR, R, SS, NA, UI, P, OS, AA, TR, C, VT\}$$

Type can be represented as function where each node v corresponds to exactly one type t,

$$Type : V \rightarrow T$$

So, $V_t \subseteq V$ represents the set of vertices of type $t \in T$. For example, V_{tr} is a set of vertices of type *Trigger*, V_{ss} is a set of vertices of type *System State*.

A *group* (G) is a collection of vertices of specific types. We have identified five necessary characteristics that essentially denote five groups of certain data types. A group is expressed as a tuple of vertices of pre-defined types. The elaboration of each group with permitted vertex types is given below:

Identity (I): An *identity* tuple is a group of three types of vertices: v_{vendor}, $v_{software}$ and $v_{software\_version}$ - which provides sufficient identification information about the vulnerability.

Origin (O): The *origin* tuple consists of two types of vertices, v_{source} and v_{reason} - pointing exactly where and why the vulnerability exists.

Exploiting Conditions (EC): This group includes, but not limited to five types of vertices that represent necessary conditions to exploit the vulnerability: $v_{operating\_system}$, v_{system_state}, $v_{network\_access}$, $v_{user\_interaction}$ and $v_{privilege}$

Actions (A): Any tuple from this group contains two vertices: $v_{trigger}$ and $v_{attacker\_action}$ - that denotes the initiation of the exploitation and manifestation of the attack respectively.

Consequence (C): This tuple has only one element $v_{consequence}$ representing consequence of the exploitation.

3.2.2 Relationships, Edges

To understand the relationships in VKG, it is important to realize that our knowledge graph is generated based on published existing vulnerabilities (CVEs). So, each vertex contains information about the associated CVEs it has extracted from. Another important fact is that multiple CVEs can correspond to a single vertex. For example, a vertex v_{aa} = *Execute remote code* can be attacker action for more than one CVE.

All the edges are directional. The connecting nodes for an edge have $from \rightarrow to$ relationships between them. Observing the nature of relationships between different data types, we find two kinds of relationships (edges): *static* and *dynamic*. There are some types of nodes (e.g., *vendor, software, software version*) that maintain a fixed relationship with their related nodes. For example, for specific software (e.g., Windows), the vendor is always fixed (Microsoft), and this *ownership* relation does not change across multiple CVEs. The edge from v_{vr} = *Microsoft* to v_s = *Windows* is considered as a static edge. Edges from *vendor to software* ($V - S$) and *software to software_version* ($S - SV$) represent static relationships in the VKG:

$$V - S \subseteq \{(v_{vr}, v_s) | v_{vr} \in V_{vr}, v_s \in V_s\}$$
$$S - SV \subseteq \{(v_s, v_{sv}) | v_s \in V_s, v_{sv} \in V_{sv}\}$$

It seems logical to deduce that the elements in *Identity* (I) tuple present static edges between them, which indicates key identification properties for any vulnerability. $V - S$ and $S - SV$ can be combined in a 3-member tuple I:

$$I = \{(v_{vr}, v_s, v_{sv}) | v_{vr} \in V_{vr}, v_s \in V_s, v_{sv} \in V_{sv}\}$$

The rest of the tuple groups contain dynamic relationships between the elements of the tuple. For example, the nodes for *Exploiting Condition* tuples do not

Table 1: Definition of different attributes of a vulnerability.• marked item can be extracted from CVE description

No.	Data Type	Definition	Example
1	cve_id	Unique identifier	CVE-2020-12510
3	published_date	Date-time when published in CVE repository	2020-11-19T00:00:00
4	cvss_score	Vulnerability severity score assigned by NVD. It ranges from 1 to 10	7.3
5	vendor_raw	Name of the vendor that owns the software	Beckhoff
6	software_raw	Vulnerable software name	TwinCat XAR 3.1
7	vendor•	Alt. vendor name	Beckhoff
8	software•	Alt. software name	TwinCAT XAR 3.1
9	software_version•	Affected versions by the specific vulnerability. E.g., *before version x.x* , *upto version y.y*, *from x through y*, *x.x and prior* etc.	all versions
10	operating_system•	If vulnerability requires any OS specification to get exploited. E.g., *Linux, windows,iOS*, etc.	Windows
11	source •	Security bug where the initial weakness originates from. E.g., *endpoint, webpage, API, component, service, process, file, module*, etc. Strongly specific to software and version	Default installation path and permissions
12	reason•	Reason why the bug exists. E.g., *improper validation, improper neutralization of input*, etc. Weakly specific to certain vulnerability. Multiple bugs can have common reason.	Installation path permissions allow local user modification
13	trigger•	Initial action that exploits the security bug. Should be specific to the source of vulnerability	Execution of TcSysUI.exe during user login
14	attacker_action•	Manifestation of attack. Additional details or steps to the trigger. May or may not be specific to the vulnerable software's security bug	Replace TcSysUI.exe with malicious code
15	user_interaction•	If any interaction is required from the user. *Yes/No* value. Detail of required interaction if *Yes*	Yes, requires login by a high-privileged user
16	system_state•	Required system configuration information for exploitation. Can be specific, like *configuration cache enabled in GitHub Actions workflows* and generic, like *target device with physical access*	Installation path at C:\TwinCAT with insufficient permissions
17	network_access•	Type of access required to exploit the vulnerability. E.g., *local access, remote access, network access with HTTP or TLS or IIOP or T3 protocols*, etc. Not strictly specific to the vulnerability	Local access
18	privilege•	Level of privilege required to exploit the vulnerability. Can be generic, like *user-level authenticated, admin-level authenticated, unauthenticated*, etc. Can be specific, like *user must have 'clusters, get' RBAC access* (CVE-2023-40029)	Less privileged local account
19	vulnerability_type•	Type of vulnerability. Can be compared to CWE (Common Weakness Enumeration)	Privilege Escalation
20	consequence•	Consequence of successful exploitation. Can be specific, like *ability to sign in admin web portal*. Can be generic *malicious code execution*	Allows execution of malicious code by higher-privileged users

have any fixed edges between certain values for any data types like *network_access* and *privilege*. Though an edge exists between v_{na} = *local access* and v_p = *with less privileged local account* for CVE-2020-12510, it is not true across all CVEs.

$$O = \{(v_{src}, v_r) | v_{src} \in V_{src}, v_r \in V_r\}$$
$$EC = \{(v_{os}, v_{ss}, ..., v_p) | v_{os} \in V_{os}, v_{ss} \in V_{ss}, ..., v_p \in V_p\}$$
$$A = \{(v_{tr}, v_{aa}) | v_{tr} \in V_{tr}, v_{aa} \in V_{aa}\}$$
$$C = \{(v_c) | v_c \in V_c\}$$

All the elements in the tuple (Except C) are in an ordered relationship. Order between data types in tuples I, O, and A are considered fixed, though *Exploitation Condition* data types can be in different order. In this work, we do not present implementation mechanism to extract order relationships between EC tuples from specific CVEs; rather, we have assumed a common order for the test implementation. The generation of tuples for a group is performed by invoking *cve-id* as input to different functions. Each function takes *cve-id* as input and provides an ordered tuple as output.

$f_I : ID \rightarrow I, id \in ID, i \in I$
$f_O : ID \rightarrow O, id \in ID, o \in O$
$f_{EC} : ID \rightarrow EC, id \in ID, ec \in EC$
$f_A : ID \rightarrow A, id \in ID, a \in A$
$f_C : ID \rightarrow C, id \in ID, c \in C$
where, $ID = \{cve-id_1, cve-id_2, ...cve-id_n\}$

3.3 Implementation of VKG

The implementation of VKG is similar to any other graph. Iteratively instantiating nodes and edges according to the definition results in the whole VKG. Before executing the VKG generation (algorithm 1), the dataset and configuration file must be ready. Details about the dataset and its attributes are explained in sub-section 3.1.

VKG Config: The configuration file *VKG.config* preserves the definition and rules for generating the VKG from *VKG-Dataset*. The VKG implementation provides the flexibility to incorporate changes or regenerate VKG conveniently. The generation algorithm 1 and other functionality algorithms 2, 3, and 4 need no changes. For example, a new data type *certificate_expired* has been introduced into the dataset. To include this data type into 'EXPLOITING_CONDITIONS' in VKG, only the VKG.config file needs to be updated.

VKG Generation: This procedure takes the dataset and config file as input and returns the VKG as output. The initialization process pulls the configuration of types, groups, and static edge types into the variables. Then, iteratively for each CVE record, a list of vertices is constructed according to the list of vertex types. No new vertex is created if there is already an existing vertex of the same type containing the same semantic meaning. Vertices of the same type having common semantic meaning are consolidated into one representative vertex. As a result, instead of creating vertex for all the CVEs of the same data type, fewer vertices are created to represent data items containing exactly equal or similar meanings. A list of cve_ids and references of neighboring nodes are stored at each vertex. After creating vertices for each CVE record, static and all other edges are created according to CVE record and VKG.config. Vertices and edges instantiated for each CVE are merged to obtain the VKG.

Algorithm 1: VKG Generation.

```
Input: (VKG-Dataset, VKG.config )
Initialization:
v_types ← initialize_vertex_types(VKG.config.types);
groups ← initialize_groups(VKG.config.groups);
static_edges_config, non_static_edges_config ←
  initialize_edge_vertex_types(VKG.config.st_edges
  _types, VKG.config.non_st_edges_types);
all_vertices ← [ ];
all_edgess ← [ ];
Output: VKG ← (all_vertices, all_edgess);
for each cve in VKG-Dataset do
    cve_vertices ← [ ] ;
    for each type in v_types do
        v_data ← get_data(cve[type]);
        vertex ← ∅
        if get_vertex(cve[cve-id],type) is ∅ then
            rep_vertex ← get_representative_vertex(type, v_data);
            if rep_vertex is ∅ then
                vertex ← create_vertex(type, type.group,
                  v_data, cve[cve − id]);
            else
                vertex ← add_cve_id_to_rep_vertex(rep_
                  vertex, cve[cve − id]);
            end
        end
        add_to_cve_vertices(vertex);
    end
    all_edges ← create_edges(static_edges_config,
      non_static_edges_config, cve_vertices);
    all_vertices ← merge(cve_vertices);
end
```

VKG Functionality: The structure of VKG serves part of the compatibility, and the rest is served by providing *query-in-VKG* capability. Defining and implementing VKG as a graph provides the built-in analysis capability as any generic graph, like traversal, shortest path, node analysis, etc. To extract vulnerability knowledge and interface with AG, 3 query functionalities have been provided and defined below:

- **get-group(cve_id,group-name):** This procedure (Algorithm 2) extracts the tuple of vertices specified by a cve_id and a group name. This procedure is used in mapping attributes of pre-conditions and post-conditions to generate the AG (Algorithm 5).

Algorithm 2: Get Specific Group by CVE ID.

```
Input: (cve-id,VKG.config.groups['group_name'])
Initialization:
group ← {group_name : group_name, tuple : ( )};
Output: group;
for each type in VKG.config.groups[group_name] do
    vertex ← get_vertex(cve − id, type);
    add_to_group_tuple(vertex, group.tuple);
end
reorder_tuple_elements_if_required(group);
```

- **get-all-groups(cve_id, VKG.config):** This procedure (Algorithm 3) provides more automation while extracting the vertices for a specific CVE. All the vertices wrapped in tuples are returned in a list. The list can be iteratively used without knowing the actual *group_name*.
- **find-path(cve_id):** The purpose of this method (Algorithm 4) is to get all the vertices for a CVE in the order of progression from identification to consequence.

Algorithm 3: Get All Groups by CVE ID.

```
Input: (cve-id,VKG.config.groups)
Initialization:
groups ← [ ]
Output: groups
for each group_name in VKG.CONFIG.groups do
    group ←
      VKG.get_group(cve − id, VKG.config.groups[group_name]);
    add_to_groups(group);
end
```

Algorithm 4: Find Path for CVE ID.

```
Input: (cve-id,VKG.config)
Output: path
Initialization:
groups ← VKG.get_all_groups(cve − id, VKG.config.groups);
path ← ( );
for each group in groups do
    for each vertex in group do
        add_to_path(vertex, path);
    end
end
```

4 AG GENERATION FROM VKG

An Attack Graph (AG) is a hierarchical representation used in network vulnerability management to showcase the potential sequences and combinations of attacks that could compromise a system. To perform any type of "*what-if analysis*", the most crucial step is generating a detailed and structured AG. In this section, we define an AG and explain the architecture behind automatically generating AG from VKG.

4.1 Attack Graph (AG)

Traditionally, AG is defined as a *Directed Acyclic Graph (DAG)*, where the nodes represent specific states or conditions of interest to an attacker, while the edges indicate the cause-consequence relationships. The AG simplifies the understanding of complex attack scenarios by using explicit conjunctive and disjunctive branch decompositions, offering a clear picture of the hierarchies present between attacker sub-goals. We are following the state-of-the-art definitions of attribute, attack, and AG (Dewri et al., 2007).

Definition 1. *The **attribute-template** (S_T) refers to the underlying categories or characteristics that make up attributes of hosts, vulnerabilities, connections, etc., such as:*

- *host-level attribute-template (S_H): associated with host name, software, versions, etc.*
- *vulnerability-level attribute-template (S_V): associated with CVEs, exploits, triggers, sources, etc.*
- *network-level attribute-template (S_C): associates with connectivity, port, IP address, access, etc.*

For a network with multiple hosts, an attribute template is defined as $S_T = S_H \bigcup S_V \bigcup S_C$. This attribute

template helps in systematically structuring and organizing the critical information of a network.

Definition 2. *An **attribute** (s) is a specific instance derived from a broader category of properties (attribute-template) concerning the network's hardware or software configuration, host-id, port, IP, access, etc., which can be either true or false. We denoted S as the set of attributes where $s \in S$.*

Each attribute-template (S_T) points to a set of attributes S that consists of particular types of attributes.

Definition 3. *Given a set of attributes S, **attack** ($\mathcal{A}$) is defined as a mapping $\mathcal{A} : S \times S \rightarrow \{true, false\}$. $a \in \mathcal{A}$ is defined as a tuple of $\langle pre(a), post(a) \rangle$ and is valid if $pre(a) \neq post(a)$ and $post(a)$ is true iff $pre(a)$ is true. $pre() \subseteq S$ and $post() \subseteq S$ are the pre-conditions and post-conditions of the attack, respectively.*

Each attack is a result of exploring some vulnerabilities in the network. For an attack to be successfully executed, an attacker has to satisfy the pre-conditions that lead to exploitation and thus resulting post-conditions, which an attacker leverages to launch further attacks in the network. Once we sum up all the attacks and their conditions, we can construct an AG.

Definition 4. *Given a set of all attacks $\mathcal{A}$, the **Attack Graph** $\mathcal{G}$ is defined as a tuple $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$, where,*

- *$\mathcal{N}$ is the set of nodes such that each node $n \in \mathcal{N}$ represents a state or condition of a host in the network, captured by a function $\phi(n) = s_n$ where $s_n \subseteq S$.*
- *$\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is the set of directed edges. An edge $e = (n_i, n_j) \in \mathcal{E}$ exists if and only if there exists an attack $a \in \mathcal{A}$ such that: $\langle pre(a), post(a) \rangle = true$, $\phi(n_i) \subseteq pre(a)$, and $\phi(n_j) \subseteq post(a)$ and $pre(a), post(a) \subseteq S$.*

The relationship between nodes and edges can be expressed as:

1. *For every edge $e = (n_i, n_j)$, n_i and n_j are distinct nodes where:*

$$\exists a \in \mathcal{A} : [\phi(n_i) \subseteq pre(a)] \wedge [\phi(n_j) \subseteq post(a)].$$

2. *Each edge e may also capture logical dependencies between attacks:*
   - *Conjunctive (AND) dependencies: for $e = (n_i, n_j)$ is active if $pre(a) = \bigcup_{n_i \in \mathcal{N}} \phi(n_i) \subseteq S$.*
   - *Disjunctive (OR) dependencies: $e = (n_i, n_j)$ is active if $pre(a) \subseteq \phi(n_i)$.*

As we define AG as DAG, we assume that attacks have no cyclic dependencies. However, for real-world scenarios with feedback or loops, $\mathcal{G}$ may contain cycles. Now to automatically construct the attacks and AG from the VKG, we define a mapping function called λ which maps the vertices value of VKG to an attribute for the AG.

Definition 5. *We define a **mapping function** λ, that maps the vertices of VKG to the attributes of AG, such as $\lambda_V : V \rightarrow S$ and groups (G) of VKG to attribute-template of AG, $\lambda_G : G \rightarrow S_T$. For each vertex $v \in V$ in VKG, there exists an attributes $s \in S$ in AG, captured by the function λ_V,*

$$\forall v \in V \, | \lambda_V : v \rightarrow s, \text{ where } s \in S$$

Similarly, λ_G maps groups (G) in VKG to attribute-templates in AG as,

$$\begin{aligned} \lambda_G &: I \rightarrow S_H \\ \lambda_V &: \{V_{vr}, V_s, V_{sv}\} \rightarrow S_h \subseteq S; \\ \lambda_G &: \{O, EC, A, C\} \rightarrow S_V \\ \lambda_V &: \{\{V_{src}, V_r\}, \{V_{os}, V_{ss}, ..., V_p\}, \\ &\quad \{V_{tr}, V_{aa}, \}, V_c\} \rightarrow S_v \subseteq S; \end{aligned}$$

We assume that the connectivity template S_C can be obtained by using any network scanner. The information about vulnerability can be obtained from VKG. So, we have the complete attribute-template $S_T = S_H \bigcup S_V \bigcup S_C$. An attack $a \in \mathcal{A}$ on a node $n_i \in \mathcal{N}$ is defined by a set of tuples $\langle pre(a), post(a) \rangle$. From VKG we can obtain all the Identity (I) vertices as the host-level attributes:

$$\lambda_V^1 : \{v_{vr}, v_s, v_{sv}\} \rightarrow \phi(n_i)$$

Also, the $pre(a) \subseteq S$ is related to the vulnerability and host-level attributes, which can be obtained from the VKG's Identity (I), Origin (O), Exploiting Conditions (EC), Actions (A) group of vertices.

$$\begin{aligned} \lambda_V^2 : \{\{v_{vr}, v_s, v_{sv}\}, \{v_{src}, v_r\}, \{v_{or}, \\ v_{ss}, ..., v_p\}, \{v_{tr}, v_{aa}\}\} \rightarrow pre(a) \end{aligned}$$

And the $post(a) \subseteq S$ of the node n_i is nothing but the Consequences (C) described in VKG.

$$\lambda_V^3 : \{v_c\} \rightarrow post(a)$$

As previously defined, the pre- and post-conditions are a subset of attributes for AG. We automatically identify which group of vertices in VKG can be used for pre- and post-conditions, and by using λ, we populate the values for AG. The architecture and algorithm to generate AG from VKG are discussed in the next section.

4.2 Architecture for AG Generation

Figure 2 illustrates the overall architecture for automatically generating AG from VKG. How the VKG is constructed is described in Section 3 along with algorithms for VKG Generation (algorithm 1), finding groups (algorithms 2, 3) and paths (algorithm 4).

The algorithm for generating AG from VKG (algorithm 5) systematically constructs an AG based

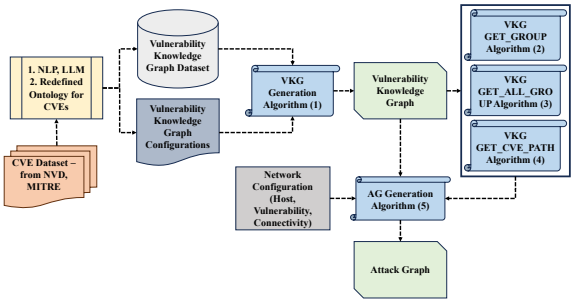

Figure 2: Overall architecture for VKG to AG generation.

Algorithm 5: Attack Graph Generation from VKG.

Input: (VKG, Network Scan Data)
Output: (Attack Graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$
Initialization:
$\mathcal{N} \leftarrow [\,]$, $\mathcal{E} \leftarrow [\,]$, $cve\_list \leftarrow [\,]$;
for ***each*** n *in* $\mathcal{N}$ **do**
  $n \leftarrow$ each host from network scan data;
  $cve\_list \leftarrow$ cve_ids for each node n;
end
$pre(a) \leftarrow [\,]$, $post(a) \leftarrow [\,]$;
for ***each*** n *in* $\mathcal{N}$ **do**
  for ***each*** *cve_id in cve_list* **do**
    $groups \leftarrow VKG.get\_all\_groups(cve-id, VKG.config.groups)$
    for ***each*** *group in groups* **do**
      if $group.group\_name == '$ *Consequence*$'$ **then**
        $G \leftarrow VKG.get\_group(cve\_id,\ group.group\_name)$
        for ***each*** v *in* $G.tuple$ **do**
          if $v.v\_data == \varnothing$ **then**
            skip;
          end
          $post(a) \leftarrow \{v.type : v.v\_data\}$;
        end
      end
      else
        $G \leftarrow VKG.get\_group(cve\_id,\ group.group\_name)$
        for ***each*** v *in* $G.tuple$ **do**
          if $v.v\_data == \varnothing$ **then**
            skip;
          end
          $pre(a) \leftarrow \{v.type : v.v\_data\}$;
        end
      end
    end
  end
end
for ***each*** e *in* $\mathcal{E}$ **do**
  for $i = 0$ *to* $i = n+1$, $i{+}{+}$ **do**
    $e \leftarrow$ connections between $(n_i, n_{i+1}) \in \mathcal{N}$;
    if $e == \varnothing$ **then**
      if $post(a_i).value$ *match any* $pre(a_{i+1}).value$ **then**
        $e \leftarrow (post(a_i))$;
      end
    end
  end
end

on vulnerability information derived from both network scan data and the VKG. For each CVE, relevant groups from the Vulnerability Knowledge Graph (VKG) are retrieved: *"Consequence"* data is stored as post-conditions ($post(a)$), while others are stored as pre-conditions ($pre(a)$). Edges are created between nodes if any $post(a)$ of one matches a $pre(a)$ of the next, forming a valid attack path. The algorithm outputs a graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$, where $\mathcal{N}$ is the set of nodes representing hosts in the network and $\mathcal{E}$ is the set of edges representing potential attack paths.

5 EVALUATION

In this section, we demonstrate the effectiveness of our approach by applying it to a test network.

5.1 Test Network & Host Information

Figure 3 shows the test network we used in this study. We employ the topology of a server-based network. The network is divided into two sub-networks: Sub-Network 1 (demilitarized zone), and Sub-Network 2 (trusted zone). A DNS Server, Mail Server, and Web Server are located in the DMZ (demilitarized zone). W-1 & W-2 are services or applications (on a machine) in the Web Server, M-1 & M-2 for the Mail Server, and D-1 & D-2 for DNS Server. There are three servers in the TZ (trusted zone): the FTP Server with F-1, F-2, & F-3, Database Server with S-1, S-2 & S-3, and Admin Server with A-1 & A-2. Table 2 lists the vulnerabilities present in this network.

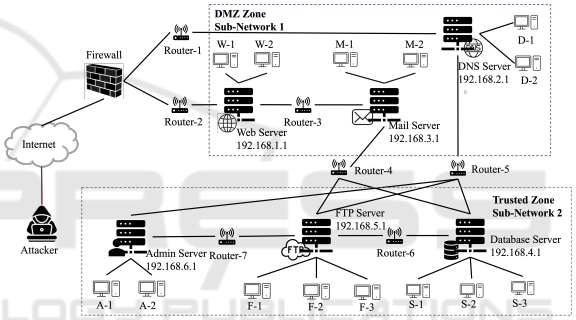

Figure 3: Topology of Test Network.

Table 2: Vulnerabilities (CVE ID) on each server.

Host	System Config: Vulnerabilities
Web Server (192.168.1.1)	W-1: Script execution. (CVE-2020-11026) W-2: Remote code execution. (CVE-2020-5902)
DNS Server (192.168.2.1)	D-1: Remote Code Execution. (CVE-2021-26897) D-2: Remote Code Execution. (CVE-2020-0718)
Mail Server (192.168.3.1)	M-1: SQL Injection (CVE-2023-3087) M-2: Unauthorized access and modification due to type juggling issue. (CVE-2023-6875)
Database Server (192.168.4.1)	S-1: SQL Injection. (CVE-2024-26026) S-2: Denial of Service. (CVE-2022-3931) S-3: Improper validation of cstrings. (CVE-2021-20329)
Admin Server (192.168.5.1)	A-1: Incorrect privilege assignment. (CVE-2020-1989) A-2: Privilege Escalation. (CVE-2020-15264)
FTP Server (192.168.6.1)	F-1: Privilege Escalation. (CVE-2021-1572) F-2: Backdoor. (CVE-2021-33540) F-3: Directory Traversal. (CVE-2020-29026)

5.2 VKG for Test Network

To incorporate the vulnerability knowledge into the AG for the test network, we extract the vulnerability subgraphs from the generated global VKG. Depending on each host configuration in the test network, the ex-

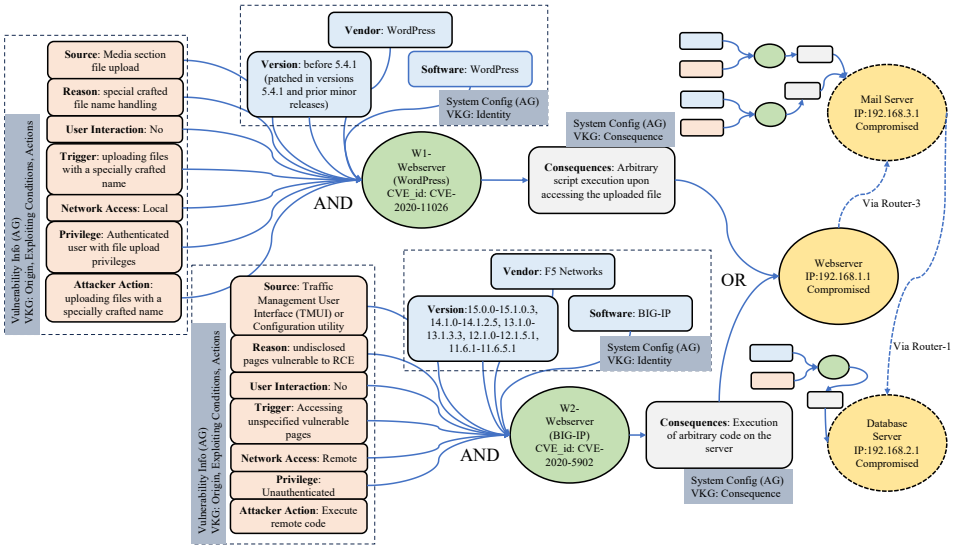

Figure 4: Automatically generated knowledge-enriched Attack Graph (for Web Server).

tracted sub-graphs are different. First, we run a vulnerability scanner on each host that provides the CVE identifiers (CVE IDs) for potential vulnerabilities (Table 2). Using the CVE IDs, we search for the sub-graphs on VKG using algorithms 2 and 3. The extracted sub-graph for each CVE contains all the fine-detailed vulnerability information in the form of nodes and edges. All the vulnerability-related information can be extracted from VKG by querying through cve_id or system information.

5.3 AG for Case Study

Once we gather all the information about the network, hosts, and vulnerabilities, we construct AG using algorithm 5. We extract the related vertices from VKG and automatically generate the attack vector for CVE-2020-11026 and CVE-2020-5902 associated with Web Server. The post-conditions, derived from the VKG's "*Consequence*" vertices, indicate the final state of exploitation, leading to Web Server compromise. Since the vulnerabilities in WordPress and BIG-IP are independent of each other, the compromise of the Web Server is represented with an 'OR' logic.

Once the Web Server is compromised, the attacker can leverage its connectivity to other systems. Information from the network scan reveals that the Web Server connects to the Mail Server via Router 3, and the Mail Server is connected to Database Server via Router 4. This allows the attacker to penetrate further into the network, expanding the AG as additional systems are compromised. The iterative nature of this process illustrates how vulnerabilities propagate through a network, enabling gradual and systematic construction of the AG illustrated in Figure 4.

5.4 Discussion

Our knowledge-based AG is enriched with more comprehensive content, providing deeper insights for network and system administrators. For instance, a conventional AG might represent an attack vector for Web Server as follows: '*Exploit CVE-2020-11026 → Script execution in WordPress → Web Server (W-1) compromised*' or '*Exploit CVE-2020-5902 → Remote code execution in F5 BIG-IP → Web Server (W-2) compromised*'. The rest of the graph in traditional AG follows: [CVEs] → [exploitations] → [compromise].

Our AG addresses previously mentioned limitations by offering actionable alternatives to patch vulnerabilities. For example, instead of simply recommending '*patching CVE-2020-5902*', from our knowledge-based AG, sys-admin can obtain multiple alternatives, such as modifying the TMUI, altering the configuration utility, upgrading the BIG-IP version, restricting network access from remote to local, or changing the authentication privileges, etc. Any of this information can provide a suggestion to mitigate the vulnerability according to the on-premise scenario. These options provide sysadmins with greater flexibility to formulate optimal mitigation strategies.

Our dataset encompasses over 1,500+ categorized CVEs, which, while not exhaustive, continues to expand as part of an ongoing effort.

6 CONCLUSION

In this work, we have addressed the challenges of enhancing Attack Graph (AG) analysis by integrating more in-depth and meaningful vulnerability informa-

tion to support improved security planning. To achieve this, we developed an architecture that leverages a separate Vulnerability Knowledge Graph (VKG), which provides fine-grained, structured insights into system weaknesses. Unlike traditional approaches, our VKG is built on a comprehensive, in-depth vulnerability dataset and designed to function independently from the AG while enabling seamless knowledge transfer. We formally defined the structure of both graphs along with the algorithms for generation and querying, ensuring a granular-level integration of vulnerability attributes into the AG without introducing inter-dependencies. The future goal is to develop automated mechanisms for updating the VKG with emerging vulnerabilities, ensuring its continued relevance and accuracy. Additionally, incorporating a feedback loop from AG to VKG could enable dynamic refinement of vulnerability insights, facilitating more advanced reasoning and analysis.

REFERENCES

Anders, Lison, P., and Moonen, L. (2023). Constructing a knowledge graph from textual descriptions of software vulnerabilities in the national vulnerability database. In Alumäe, T. and Fishel, M., editors, *Proceedings of the 24th Nordic Conference on Computational Linguistics (NoDaLiDa)*, pages 386–391, Tórshavn, Faroe Islands. University of Tartu Library.

Bashir, S. K., Podder, R., Sreedharan, S., Ray, I., and Ray, I. (2024). Resiliency graphs: Modelling the interplay between cyber attacks and system failures through ai planning. In *2024 IEEE 6th International Conference on Trust, Privacy and Security in Intelligent Systems, and Applications (TPS-ISA)*, pages 292–302. IEEE.

Dewri, R., Poolsappasit, N., Ray, I., and Whitley, D. (2007). Optimal security hardening using multi-objective optimization on attack tree models of networks. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 204–213.

Inokuchi, M., Ohta, Y., Kinoshita, S., Yagyu, T., Stan, O., Bitton, R., Elovici, Y., and Shabtai, A. (2019). Design procedure of knowledge base for practical attack graph generation. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, Asia CCS '19, page 594–601, New York, NY, USA. Association for Computing Machinery.

Jia, F., Hong, J. B., and Kim, D. S. (2015). Towards automated generation and visualization of hierarchical attack representation models. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 1689–1696. IEEE.

Jing, J. T. W., Yong, L. W., Divakaran, D. M., and Thing, V. L. L. (2017). Augmenting mulval with automated extraction of vulnerabilities descriptions. In *TENCON 2017 - 2017 IEEE Region 10 Conference*, pages 476–481.

Lallie, H. S., Debattista, K., and Bal, J. (2017). An empirical evaluation of the effectiveness of attack graphs and fault trees in cyber-attack perception. *IEEE Transactions on Information Forensics and Security*, 13(5):1110–1122.

Liu, X., Guo, X., and Gu, W. (2025). Seckg2vec: A novel security knowledge graph relational reasoning method based on semantic and structural fusion embedding. *Computers & Security*, 149:104192.

Ou, X., Boyer, W. F., and McQueen, M. A. (2006). A scalable approach to attack graph generation. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 336–345.

Ou, X., Govindavajhala, S., and Appel, A. W. (2005). MulVAL: A logic-based network security analyzer. In *14th USENIX Security Symposium (USENIX Security 05)*, Baltimore, MD. USENIX Association.

Poolsappasit, N., Dewri, R., and Ray, I. (2011). Dynamic security risk management using bayesian attack graphs. *IEEE Transactions on Dependable and Secure Computing*, 9(1):61–74.

Ray, I., Sreedharan, S., Podder, R., Bashir, S. K., and Ray, I. (2023). Explainable ai for prioritizing and deploying defenses for cyber-physical system resiliency. In *2023 5th IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)*, pages 184–192. IEEE.

Yin, J., Hong, W., Wang, H., Cao, J., Miao, Y., and Zhang, Y. (2024). A compact vulnerability knowledge graph for risk assessment. *ACM Trans. Knowl. Discov. Data*, 18(8).

Zhang, Y., Du, T., Ma, Y., Wang, X., Xie, Y., Yang, G., Lu, Y., and Chang, E.-C. (2024). Attackg+:boosting attack knowledge graph construction with large language models. arXiv 2405.04753, Available from https://arxiv.org/abs/2405.04753.