Deterring Intellectual Property Thieves: Algorithmic Generation of Adversary-Aware Fake Knowledge Graphs

Snow Kang
Dartmouth College, snow.kang.21@dartmouth.edu

Follow this and additional works at: https://digitalcommons.dartmouth.edu/senior_theses

Part of the Information Security Commons

Recommended Citation
https://digitalcommons.dartmouth.edu/senior_theses/220

This Thesis (Undergraduate) is brought to you for free and open access by the Theses and Dissertations at Dartmouth Digital Commons. It has been accepted for inclusion in Dartmouth College Undergraduate Theses by an authorized administrator of Dartmouth Digital Commons. For more information, please contact dartmouthdigitalcommons@groups.dartmouth.edu.
Deterring Intellectual Property Thieves: Algorithmic Generation of Adversary-Aware Fake Knowledge Graphs

Snow Kang
Undergraduate Thesis presented for the degree of Bachelor of Arts in Computer Science

Advised by
Professor V.S. Subrahmanian

Dartmouth College
Hanover, New Hampshire
June 1, 2021
# Contents

Abstract 2

1 Introduction 3  
1.1 Pretext 3  
1.2 Tackled Problem 4  
1.3 Related Work 5

2 Main Contributions 6

3 Definitions from our AAAI Paper 7  
3.1 Preliminaries 7  
3.2 The FakeKG Problem 8

4 Implementation 10  
4.1 Generation of U 10  
4.2 CliqueComputation 12  
4.3 Clique-FakeKG 14

5 Visualization 16  
5.1 Web-Based Tool 16

6 Experimentation 19  
6.1 Method 19  
6.2 Results 20

7 Conclusion 23

A Visualization Code 26  
A.1 parent.html 26  
A.2 child.html 28

B Algorithm Code 35

C Experimentation Code 41
Deterring Intellectual Property Thieves:  
Algorithmic Generation of Adversary-Aware  
Fake Knowledge Graphs  

Snow Kang  
June 1, 2021  

Abstract  

Publicly available estimates suggest that in the U.S. alone, IP theft costs our economy between $225 billion and $600 billion each year.\(^1\) In our paper, we propose combating IP theft by generating fake versions of technical documents. If an enterprise system has \(n\) fake documents for each real document, any IP thief must sift through an array of documents in an attempt to separate the original from a sea of fakes. This costs the attacker time and money - and inflicts pain and frustration on the part of its technical staff.

Leveraging a graph-theoretic approach, we created the CLIQUE-FAKEKG algorithm to achieve a formalized adversary-aware standard. That is, even an attacker who knows our algorithm as well as every input other than the original Knowledge Graph must not be able to identify the real graph. We create a distance graph between all the KGs in our input universal set \(U\) where vertices are KGs and edges are only drawn between KGs if the distance between them is within a desired interval. Then, if an \((n + 1)\)-clique in our distance graph contains \(K_0\), we have found \(n\) fake KGs that fall within the desired distance interval from one another and from \(K_0\).

In our paper, we first discuss the complexity of this problem and show that it is NP hard. Next, we develop CLIQUE-FAKEKG to solve it using probability inspired by work in cryptography. When testing CLIQUE-FAKEKG on human subjects using 3 diverse real-world datasets, we achieved an 86.8% deception rate with users showing difficulty in distinguishing which KG from a set of KGs was the original \(K_0\) from which all the graphs were generated.

\(^1\)https://legaljobs.io/blog/intellectual-property-statistics/
Chapter 1

Introduction

1.1 Pretext

Design specs for new aeronautical designs, chemical formulae for emerging vaccines, UML diagrams for cutting-edge technologies. A handful of decades ago, these secrets would all be carefully tucked away in manila folders, safeguarded by layers of locks and guards. Now, in an ever more digitized world, businesses and governments are entrusting computer systems with their most sensitive files. The cyber threat landscape is rapidly evolving and particularly dangerous: data breaches in the 21st century can compromise millions of people at once and simultaneously remain undetected for weeks or even months. The same technologies that allow for globalization and for massive loads of information to be mere clicks away render data protection difficult. For better or for worse, access has never been so easy.

Resultantly, the 2019/2020 Global Fraud and Risk Report by Kroll reveals that nearly three quarters of companies deem intellectual property (IP) theft a “high” or “significant” priority\(^1\); this is not surprising as “intellectual property can constitute more than 80 percent of a single company’s value today.”\(^2\) Along with my advisor, Professor V.S. Subrahmanian, and Professors Cristian Molinaro and Andrea Pugliese from the University of Calabria, I have been conducting research to address this ever-widening issue from the angle of deterrence. How can we place obstacles in IP thieves’ way? How can we make it more time-consuming to steal IP? More difficult? More confusing?


1.2 Tackled Problem

Using knowledge graphs, we can intuitively represent the relationships various entities share, enabling us to store information about complex processes. Knowledge graphs have been widely used in domains where intellectual property and/or specialized knowledge must be kept confidential, such as proprietary software designs and the content of technical documents. If every enterprise system contained \( n \) fake documents per each real document, potential IP thieves would need to waste time, energy, and resources to attain valuable information. Thus, we landed on the following questions:

*Given an original knowledge graph that is the representation of the knowledge within a technical document, is there some way to manufacture some \( n \) number of fake knowledge graphs?*

*Can we make these fake knowledge graphs difficult to discern as synthetic yet different enough from the original so that they convey false information?*

If this were possible, one could then extract the text from these fake knowledge graphs (using strategies such as in Chakraborty et al. 2019 or Koncel-Kedziorski et al. 2019) and generate corresponding fake technical documents. This way, any IP thief would be forced to sift through an array of different versions of any real document in an enterprise system to distinguish which is the original.

Further, is there a way to generate such a set of fake knowledge graphs that leaks minimal information pertaining to the original graph? We explored whether the “no security without obscurity” ideal could be achieved. That is, even if the attacker has complete knowledge of the exact algorithm used to generate the fakes plus every input other than the original knowledge graph, is there an algorithm that does not aid the attacker in identifying the real graph? We consequently appended onto our tackled problem:

*Can our generation of fake knowledge graphs be “adversary-aware” and irreversible?*

Our ultimate solution to the tackled problem, the CLIQUE-FAKEKG algorithm, is intended to be used in a 3-step process to generate fake versions of a technical document: (i) Given an original document, we extract a knowledge graph. (ii) We then use the techniques in our paper to generate a set \( K \) of fake KGs. (iii) For each of the fake KGs in \( K \), we then generate a fake document. Our paper focuses on (ii).
1.3 Related Work

Across numerous fields, deception technologies are being employed for defensive purposes. Traditional honeypot schemes, the original deception technologies, have been used for decades to imitate important sites and bait malicious actors. More recently, watermarks and steganography methodologies protect against the plagiarism of media artists’ work, in a similar vein to how recording companies used fake MP3 songs to deter music pirating (Kushner 2003).

When it comes to the protection of technical documents specifically, there are several notable current research efforts which this thesis attempts to build upon. Chakraborty et al. 2019 proposes generating fake documents from authentic documents by swapping text based on meta-centrality metrics; this work is then extended by Xiong et al. 2020 who show that equations within documents can similarly be faked in a highly believable manner. Further, Abdibayev et al. 2021 and Han et al. 2021 continue to build upon ways of producing convincing dupes using target concept replacements and probabilistic logic graphs, respectively.

Our algorithm offers two main distinguishing features from prior work: (i) it considers the knowledge encoded within a document, and (2) it accounts for adversaries having access to the algorithm as well as all the inputs.
Chapter 2

Main Contributions

I conducted my research in collaboration with my Dartmouth thesis advisor and our team lead, Professor V.S. Subrahmanian, in addition to two professors from the University of Calabria (Italy), Professor Cristian Molinaro and Professor Andrea Pugliese. In the “Definitions from our AAAI Paper” section, I will build upon our paper Kang et al. 2021, which was jointly developed. I independently created proofs of some concepts, implemented all of the code, created a UI interface for graphical visualizations, and produced the content for our experimentation. I also helped ideate definition properties and analyze the final results.

This research has resulted in one published paper and one ongoing paper. As a research team, our contributions are as follows:

1. We formally define the FakeKG problem and show that solving it is NP-hard.

2. We formally define a stringent “adversary-aware” standard that goes beyond what previous attempts at generating fake technical documents have done.

3. We propose an algorithm CLIQUE-FAKEKG that uses graph theory to achieve the adversary-aware standard.

4. We consider the knowledge encoded within a document to generate highly believable fakes.
Chapter 3

Definitions from our AAAI Paper

3.1 Preliminaries

Let $E$, entities, and $R$, relations, be two disjoint finite sets. A knowledge graph $K$ is a set of triples from the Cartesian product $E \times R \times E$. Each triple is of the form $(s, r, o)$, semantically representing subject entity $s$ and object entity $o$ being related via relation $r$. A knowledge graph (KG) can thus be represented as a directed graph, where each $(s, r, o)$ triple translates to a directed edge labeled with relation $r$ stemming from subject entity $s$’s node and feeding into object entity $o$’s node.

In addition to representing each individual KG as a directed graph, our algorithm also incorporates a broader undirected graph where nodes are KGs and two KGs are related together via an unlabeled, undirected edge if they are within some distance of one another.

An undirected graph $G$ is a pair $\langle V, E \rangle$, where $V$ is a finite set of vertices, and $E \subseteq V \times V$ is a set of edges, or unordered pairs of vertices. For the purposes of this paper, “graph” will refer to undirected graphs whereas KGs, which can be represented as directed graphs, will always be referred to as KGs.

Given a set $U$ of KG nodes, we define a distance function $d$ for $U$ as the function $d : U \times U \rightarrow [0, 1]$ such that for all $K, K', K'' \in U$,

- $d(K, K') = 0$ iff $K = K'$,
- $d(K, K') = d(K', K)$, and
- $d(K, K') \leq d(K, K'') + d(K'', K')$.

For experimentation, I implemented the Jaccard distance function, which is defined as $d(K, K') = 1 - \frac{|K \cap K'|}{|K \cup K'|}$.

Further, a clique of $G$ is a subset $C$ of $V$ such that for every pair of distinct vertices $v$ and $v'$ in $C$, $(v, v') \in E$. $C$ represents a maximum clique of $G$ iff there
is no clique $C'$ of $G$ such that $|C| < |C'|$. A $k$-clique of $G$ is a clique of $G$ with cardinality $k$.

Given a set of vertices $V' \subseteq V$, the subgraph of $G$ induced by $V'$, denoted $G[V']$, is the graph $(V', E')$ where $E' = \{(v', v'') | v', v'' \in V' \text{ and } (v', v'') \in E\}$. For an arbitrary set $S$, $2^S$ denotes the powerset of $S$, or the set of all subsets of $S$.

### 3.2 The FakeKG Problem

**Definition 1 (FAKEKG problem).** Given a set $\mathcal{U}$ of KGs, a distance function $d$ for $\mathcal{U}$, a knowledge graph $K_0 \in \mathcal{U}$, an integer $n \geq 1$, and an interval $\tau = [\ell, u] \subseteq [0, 1]$, find a set $\mathcal{K} = \{K_0, K_1, \ldots, K_n\} \subseteq \mathcal{U}$ of $n + 1$ distinct KGs s.t. $\ell \leq d(K_i, K_j) \leq u$ for every $0 \leq i \neq j \leq n$.

Thus, an instance $I = \langle \mathcal{U}, d, K_0, n, \tau \rangle$ of FAKEKG includes

- a set $\mathcal{U}$ of KGs, which are all the KGs of interest for the application at hand;
- a function $d$ measuring the distance between KGs in $\mathcal{U}$;
- the original KG $K_0$ for which we want to generate fakes;
- the number $n$ of fake KGs we want to generate;
- an interval $\tau$ that bounds the distance between any two distinct KGs in a solution.

A solution for $I$ is a set $\mathcal{K} = \{K_0, K_1, \ldots, K_n\}$ of $n + 1$ distinct KGs such that their pairwise distance lies in the interval $\tau$. Notice that $\mathcal{K}$ includes the original knowledge graph $K_0$ along with $n$ additional KGs. The requirement $\ell \leq d(K_i, K_j)$ allows users to set a minimum desired distance between every pair of distinct KGs in $\mathcal{K}$, so that every fake KG is “far enough” from the original $K_0$. The requirement $d(K_i, K_j) \leq u$ allows users to set a maximum desired distance between every pair of distinct KGs in $\mathcal{K}$, so that every fake KG is “close enough” to the original $K_0$. (e.g., to keep them believable).

In order to show the NP-hardness of FAKEKG, we demonstrated a reduction from the NP-complete CLIQUE problem, or given an undirected graph $G$ and an integer $k$, decide whether $G$ has a $k$-clique. Specifically, we reduce the CLIQUE problem to the problem of deciding whether an instance of FAKEKG admits a solution, which in turn implies NP-hardness of FAKEKG.

**Theorem 2.** The FAKEKG problem is NP-hard.
Finally, to align with the cryptography ideal of “no security through obscurity,” we impose upon ourselves an adversary-aware standard: our algorithm must still be secure in situations where the adversary knows the algorithm $A$, as well as $\mathcal{K}$, $\mathcal{U}$, $d$, $n$, and $\tau$. That is, when given every input except the original KG $K_0$, an adversary must not be able to use this information to get closer to uncovering $K_0$.

Mathematically, this means that if $A$ is given an input $I = \langle \mathcal{U}, d, K_0, n, \tau \rangle$ and outputs $\mathcal{K} = \{K_0, K_1, \ldots, K_n\}$, all the $K_i$’s in $\mathcal{K}$ must have the same probability of being the original KG. That is, for every $K_i \in \mathcal{K}$, the probability of $\mathcal{K}$ being the output when $K_i$ is the input remains the same. If this is the case, then the knowledge of $A$, $\mathcal{U}$, $d$, $n$, $\tau$, and $\mathcal{K}$ does not allow an adversary to make a better guess than picking one KG (uniformly at random) as the original KG. Thus, we have:

**Property 1** (Adversary-aware algorithm). A (randomized) algorithm $A$ to solve FakeKG is adversary-aware iff it satisfies the following property: For every instance $I = \langle \mathcal{U}, d, K_0, n, \tau \rangle$ of FakeKG, if $\mathcal{K} = \{K_0, K_1, \ldots, K_n\}$ is a possible output of $A(I)$, that is

$$\Pr[A(I) = \mathcal{K}] > 0,$$

then

$$\Pr[A(\langle \mathcal{U}, d, K_i, n, \tau \rangle) = \mathcal{K}] = \Pr[A(\langle \mathcal{U}, d, K_j, n, \tau \rangle) = \mathcal{K}]$$

for every $0 \leq i \neq j \leq n$.

Let us now consider a deterministic algorithm $A$, or an algorithm that always returns the same output when it is called multiple times with the same input. $A$ can be seen as a particular randomized algorithm where, for every instance $I$ of FakeKG, $A(I)$ is such that $\Pr[A(I) = \mathcal{K}] = 1$ for some $\mathcal{K} \in 2^\mathcal{U}$. We can then write $A(I) = \mathcal{K}$ to denote that $\mathcal{K}$ is the (only) output of $A$ when it is called with input $I$.

In order for $A$ to be adversary-aware, then, regardless of which $K_i$ is inputted into $A$, $A$ must output the same $\mathcal{K}$.

**Proposition 3.** A deterministic algorithm $A$ is adversary-aware iff it satisfies the following property: For every instance $I = \langle \mathcal{U}, d, K_0, n, \tau \rangle$ of FakeKG, if

$$A(I) = \mathcal{K} = \{K_0, K_1, \ldots, K_n\},$$

then

$$A(\langle \mathcal{U}, d, K_i, n, \tau \rangle) = \mathcal{K}$$

for every $1 \leq i \leq n$. 


Chapter 4

Implementation

For the purposes of this thesis, I will focus on demonstrating the implementation and visualization of our ultimate algorithm, CLIQUE-FAKEKG. The proof of this algorithm being an adversary-aware solution to FAKEKG can be found in Kang et al. 2021.

I implemented the algorithm in Python on a 2.3 GHz Dual-Core Intel Core i5 with 8GB of LPDDR3 RAM, running MacOS Catalina Version 10.15.6. I used the NetworkX package of Python for graph representation This section will solely contain the main algorithm code; the code in full can be found in Appendix B.

4.1 Generation of U

Overview:
To generate a set $U$ of KGs, a random subgraph $G_0$ of the original dataset was chosen to serve as the “starter” KG. Next, new KGs were built by adding or deleting random vertices/edges/labels from $G_0$ and the subsequent KGs built from $G_0$.

In the code, $generate\_U$ takes in the following parameters:

- $G_0$, the initial starting subgraph from which other KGs are produced, expressed as lines of tab separated triples,
- $D$, a threshold percentage for deletion, expressed as an integer,
- $A$, a threshold percentage for addition, expressed as an integer,
- $L$, the set of possible edge label id’s,
- and $\text{SizeU}$, the desired size of $U$.

It outputs a constructed $U$ of size $\text{SizeU}$. 

10
def generate_U(G0, D, A, L, SizeU):
    
    # To ensure that there are no identical triples
    U_set = set([frozenset(G0)])

    U = {}
    U[0] = G0

    while len(U) < SizeU:
        d = randint(0, D)
        a = randint(0, A)

        # Take a random KG from U, the set of G_0 and built KGs
        lst = list(U_set)
        shuffle(lst)
        G_prime = set(lst[0])

        # Create a directed graph using NetworkX library
        G_prime_graph, _ = construct_DiGraph(G_prime)
        G_prime_size = len(G_prime)
        G_prime_nodes = G_prime_graph.nodes()

        num_edges_to_delete = int(round(d/100*G_prime_size))
        num_edges_to_add = int(round(a/100*G_prime_size))

        # delete d% of edges in G'
        lst = list(G_prime)
        shuffle(lst)
        G_prime = set(lst[:G_prime_size - num_edges_to_delete])

        # add a% of edges to G'
        for _ in range(num_edges_to_add):
            new_edge = sample(G_prime, 1)[0] # initialize to a known edge

            while new_edge in G_prime:
                lst = list(L)
                shuffle(lst)
                new_label = lst[0]
                lst = list(e_dict)
                shuffle(lst)
                new_endpoints = lst[:2]

                if dataset == "Nations" or dataset == "Umls":
                    new_edge = str(new_label) + "\t" + "\t" + \
                    str(new_endpoints[0]) + "\t" + "\t" + \
                    str(new_endpoints[1])
                elif dataset == "FB15K":
                    break

                else:
                    break

                shuffle(lst)
                new_label = lst[0]
                lst = list(e_dict)
                shuffle(lst)
                new_endpoints = lst[:2]
4.2 CliqueComputation

Overview:

CLIQUECOMPUTATION is a helper function that is called within the main algorithm, CLIQUE-FAKEKG. It takes an instance \( I \) as input and outputs a set of cliques each of size at least \( n + 1 \) (\( k \)-cliques where \( k \geq n + 1 \)).

- On line 1 of the algorithm, a distance graph \( G_I \) is defined: an undirected graph where KGs in \( U \) are connected by unlabeled edges if the distance between them is within distance threshold \( \tau \).

- In the algorithm, graph \( G \) is repeatedly pruned and searched for maximum cliques. On line 2, \( G \) is first initialized to the entire distance graph \( G_I \).

- \( C \) is the set of \( k \)-cliques that is being built and that will ultimately be outputted. On line 3, \( C \) is initialized to the empty set.

- Lines 4-10 form a while loop that runs so long as \( G \) has a clique of size at least \( n + 1 \). In the loop, a random maximum clique of the current \( G, C \), is identified and added to \( C \). If \( C \) contains \( K_0 \), then \( C \) is immediately returned. Due to the while loop condition, \( C \) must be a clique of at least \( n + 1 \). If \( C \) does not contain \( K_0 \), we can remove every KG contained in \( C \) from the candidate set; \( C \) is thus deleted from \( G \) and the loop repeats. Eventually, as \( G \) is being continuously pruned, \( C \) will be returned.
Algorithm:

Algorithm 1 CLIQUECOMPUTATION

Input: An instance $I = (U, d, K_0, n, \tau)$ of FakeKG.
Output: A set of $k$-cliques of $G_I$ with $k \geq n + 1$.

1: Let $G_I = (V_I, E_I)$.
2: $G = (V, E) = (V_I, E_I)$.
3: $C = \emptyset$.
4: while $G$ has a clique of size at least $n + 1$ do
5: Let $C$ be a maximum clique of $G$.
6: Add $C$ to $C$.
7: if $C$ contains $K_0$ then
8: return $C$.
9: $G' = G[V \setminus C]$.
10: $G = G'$.
11: return $C$.

Code:

```python
import networkx as nx

def clique_computation(G_I, K_0, n):
    G = G_I
    C = []
    C_nodes = set()
    C_graph = nx.DiGraph()

    while nx.graph_clique_number(G) > n:
        maximal_cliques = list(nx.find_cliques(G))
        max_size_clique = n+1
        maximum_cliques = []

        for clique in maximal_cliques:
            clique_num_nodes = nx.number_of_nodes(clique)
            if clique_num_nodes == max_size_clique:
                maximum_cliques.append(clique)
            elif clique_num_nodes > max_size_clique:
                max_size_clique = clique_num_nodes
                maximum_cliques = [clique]

        shuffle(maximum_cliques)
        max_clique = set(maximum_cliques[0])

        # Construct max_clique_graph by finding the subgraph of
        # G with max_clique's
        # nodes
        max_clique_graph = G.subgraph(max_clique)
```
4.3 Clique-FakeKG

Overview:

CLIQUE-FakeKG takes as input $I$ and outputs a solution for this instance. It first performs the clique computation in line 1 using CLIQUECOMPUTATION. If CLIQUECOMPUTATION finds a clique containing $K_0$, a clique which is guaranteed to have at least $n + 1$ elements, then CLIQUE-FakeKG outputs a set of exactly $n$ elements chosen uniformly at random from $C \setminus \{K_0\}$. Otherwise, it outputs the $\emptyset$ to indicate that no solution has been found for the instance $I$.

Algorithm:

**Algorithm 2 CLIQUE-FakeKG**

**Input:** An instance $I = \langle U, d, K_0, n, \tau \rangle$ of FakeKG.

**Output:** A solution for $I$ or $\emptyset$.

1. $C = \text{CLIQUECOMPUTATION}(I)$.
2. if $C$ includes a clique $C$ containing $K_0$ then
3.   Pick a set $S$ of $n$ elements from $C \setminus \{K_0\}$ uniformly at random.
4.   return $S \cup \{K_0\}$.
5. else
6.   return $\emptyset$.

Code:

```python
def clique_fakeKG(G_I, K_0, n):
    C = clique_computation(G_I, K_0, n)

    shuffle(C)
    for clique in C:
        if K_0 in clique:
            clique.remove(K_0)
            lst = list(clique)
            shuffle(lst)
```
K = set(lst[:n])
K.add(K_0)
return K

return set()
Chapter 5

Visualization

5.1 Web-Based Tool

I employed the d3-force module of the JavaScript library D3.js to develop a web-based tool for graph visualization. Given a KG, the tool displays it as a directed graph with labeled vertices and edges. This visualization was used during experimentation to present KGs to evaluators and the code can be found in Appendix A.

I split the scripts among two files: parent.html and child.html. The parent is the larger container with a UI that allows users to select which test number to generate. Upon selecting a test number, the ten corresponding KGs are displayed as force-directed graphs. The child.html file stores the code for an individual force-directed graph.

Click generate to see a graph

Note: you can drag/click on a node to lock it in place

Figure 5.1: Parent.html user interface prompt
Deterring Intellectual Property Thieves

Test13

Note: you can drag/click on a node to lock it in place

Figure 5.2: Parent.html populated display

Figure 5.3: Child.html example: a KG as a force-directed graph

Each test should be present as a subfolder titled "Test#" (Test1-Test22) of the same directory as parent.html and child.html. Within each test folder, each of the 10 KGs should be present as JSONs of the following form:

```json
{
    ...
}
```
Chapter 6

Experimentation

6.1 Method

For experimentation, I found three real world datasets: Nation\(^1\) (Kim, Xie, and Ong 2016), which represents international relations among countries, UMLS\(^2\) (Kim, Xie, and Ong 2016), which represents biomedical relations, and the Microsoft FB15K-237\(^3\) (Toutanova et al. 2015), which stores triples and textual mentions of Freebase entity pairs. FB15K-237 will be referred to as FB for brevity. The code for experimentation can be found in Appendix C.

For each dataset, I extracted 22 original KGs. For each original KG, I then used the Clique-FakeKG code I wrote to generate 9 fakes, 3 for each of the following ranges of \(\tau\): [0, 1/3], [1/3, 2/3], and [2/3, 1]. In sum, the set \(T\) consisted of 66 tests, each consisting of 9 fake KGs plus the original \(K_0\).

I used the generate\(_U\) code to create a \(U\) consisting of reasonably sized KGs. The average number of vertices and edges of the original KGs was 15.79 and 9.98, respectively (16.18 and 9.95 for the fake KGs). The size of \(U\) was 50. We used the Jaccard distance function.

Our collaborators from Calabria ran experiments with 10 human evaluators, all possessing a Masters or a Ph.D. degree in Computer Engineering. Each evaluator was given the full 66 tests in \(T\) for a total of 660 tests overall. For each test, I included a clear description of the domain and the labels and asked 5 questions to gauge competence. For each question, the evaluator was presented two vertices and had to identify which label(s), out of 5, applied if used on an edge between the two vertices. Each question was generated by (i) randomly picking a triple \(\langle s, r, o \rangle\) from the dataset, (ii) making \(s\) and \(o\) the basis of the question, (iii) presenting \(r\) as a possible answer, together with 4 other randomly picked labels from the dataset, and

\(^1\)https://github.com/dongwookim-ml/kg-data/tree/master/nation
\(^2\)https://github.com/dongwookim-ml/kg-data/tree/master/umls
(iv) considering as right all of the chosen labels \( r' \) for which there was actually a triple of the form \( \langle s, r', o \rangle \) in the dataset.

From these questions, a competency score was computed by rewarding 1 point for choosing a correct label and 1 point for not choosing an incorrect label. As each question had 5 available answer choices, an evaluator could get a score of up to 5 points per question, which was then normalized in the [0, 1] interval. The competence score was 0.61 on the Nation dataset, 0.74 on the UMLS dataset, and 0.81 on the FB dataset. The overall average competence score was 0.72. The average time to generate one test was 10.57 seconds for the Nation dataset, 12.44 for the UMLS dataset, and 50.55 seconds for the FB dataset.

Next, evaluators were shown the graphical visualizations of the KGs and told to select their top-three best guesses as to which KG they believed to be the original \( K_0 \) for each test.

A Deception Rate (DR) metric was calculated from their responses. For each subject \( h \), each original KG \( K_0 \), and \( r \in \{ 1, 2, 3, \text{top-3} \} \), if \( h \) selected a fake KG as their \( r \)-th choice, \( w(h, K_0, r) = 1 \), and \( w(h, K_0, r) = 0 \) otherwise; for the top-three case, evaluators were recorded to be correct when any of their top-3 choices were right. Then, for each \( r \)-th choice, the average value of \( w \) was computed for each subject over all KGs.

\[
DR(r, h) = \frac{\sum_{K_0} w(h, K_0, r)}{|T|}
\]

\[
DR(r, K_0) = \frac{\sum_{h} w(h, K_0, r)}{10}.
\]

6.2 Results

The following graphs illustrate the obtained results:

![Figure 6.1: Average deception rate for the different choices.](image)
Table 6.1: Standard deviation of $DR(r, h)$ (resp. $DR(r, K_0)$) across the human subjects (resp. across $T$).

<table>
<thead>
<tr>
<th></th>
<th>St.dev. $DR(r, h)$</th>
<th>St.dev. $DR(r, K_0)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st choice</td>
<td>0.06</td>
<td>0.11</td>
</tr>
<tr>
<td>2nd choice</td>
<td>0.04</td>
<td>0.09</td>
</tr>
<tr>
<td>3rd choice</td>
<td>0.04</td>
<td>0.10</td>
</tr>
<tr>
<td>Top-3 choice</td>
<td>0.11</td>
<td>0.16</td>
</tr>
</tbody>
</table>

High deception levels are shown across the data, which is summarized in Kang et al. 2021:

In 86.8% of the cases (Figure 6.1) the KG that users selected as their top choice was in fact fake. Even in the top-3 case, our approach was able to deceive users in 62.7% of the cases. Moreover, the standard deviation across the human subjects (Table 6.1) was lower than that across the tests, which suggests that our approach achieves similar performance in achieving deception on different subjects. Finally (Figure 6.2), the deception rate for the first choice was above
90% on 41 out of 66 tests (14 out of 22 for Nation, 16 for UMLS, and 11 for FB) and above 80% on 56 out of 66 tests—for each of the datasets, in just 1 out of 22 test the deception rate was lower than 70%. For the second and third choices, the results were similar. Even in the top-3 case, the deception rate was above 60% in 49 out of 66 tests.
Chapter 7

Conclusion

In this thesis, the FakeKG problem is formally defined, where $n$ fake knowledge graphs are generated from an original $K_0$ in a way that is irreversible and cryptographically secure. This is formalized into an “adversary-aware” standard, which the CLIQUE-FakeKG algorithm is presented as a solution for.

Next, the CLIQUE-FakeKG algorithm is implemented using Python, and a D3.js data visualization tool is generated for depicting the generated fake KGs. Experimental tests are produced from 3 diverse real-world datasets for distribution to 10 human subjects. A Deception Rate metric is defined, and the responses of subjects instructed to identify an original $K_0$ among 9 fake KGs reveal high deception rates. From the data, CLIQUE-FakeKG is ultimately shown to have succeeded in generating adversary-aware fake knowledge graphs capable of deceiving human subjects.

As future steps, we are looking into refining the generation of $U$, formalizing graph transformations, and analyzing different transformation costs.
Acknowledgements

A major thank-you to Professors Andrea Pugliese, Cristian Molinaro, and V.S. Subrahmanian for providing me with such a wonderful first exposure to research. Their guidance and teachings have been tremendous, and I remain inspired by their intellect and work ethic; I hope to follow in all your footsteps.

I would also like to thank every Dartmouth professor I’ve had. There truly has not been a single Computer Science class that I have not enjoyed, and I am sincerely grateful for my time spent in the Life Sciences Center, where I met some of my greatest mentors and my greatest friends. I will miss Dartmouth dearly.
Bibliography


Appendix A

Visualization Code

A.1 parent.html

```
<html>
<!DOCTYPE html>
<html lang="en">
<head>
<br />
<meta charset="utf-8">
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta name="viewport" content="width=device-width, initial-scale=1">
<style type="text/css">
.node {}
.link { stroke: #999; stroke-opacity: .6; stroke-width: 1px; }
</style>
</head>
<body>
<h1 id="GraphTitle">Click generate to see a graph</h1>
<h3> Note: you can drag/click on a node to lock it in place</h3>
<form name="generate_params_form" id="GenerateParamsForm" onsubmit="return generateGraph(event)"
<select name="TestFold" id="TestFolder"/>
<input type="submit" value="Generate"/>
<br />
<p id="iFrames"></p>
<script>
</script>
```
Deterring Intellectual Property Thieves

```
numGraphs = 10

let iframeHTML = ""
for (let i = 1; i < numGraphs + 1; i++) {
  iframeHTML += `<iframe id="iframe${i}" width="500" height="500"></iframe>`
}
document.getElementById("iFramess").innerHTML = iframeHTML;

let testFolderHTML = ""
for (let i = 1; i < 23; i++) {
  testFolderHTML += `<option value="Test${i}">Test${i}</option>`
}
document.getElementById("TestFolder").innerHTML = testFolderHTML;

function generateGraph(e) {
  e.preventDefault();

  var folder = document.getElementById("TestFolder").value;
  console.log(folder);
  document.getElementById("GraphTitle").innerHTML = `${folder}`;
  for (let i = 1; i < numGraphs + 1; i++) {
    document.getElementById(`iframe${i}`).src = `./child.html?folder=${folder}&graph=${i};
  }
  return false;
}
```

Chapter A  Snow Kang  27
A.2  child.html

```html
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <style type="text/css">
    .node {}
    .link { stroke: #999; stroke-opacity: .6; stroke-width: 1px; }
  </style>
</head>
<body>
<svg width="500" height="500"></svg>

<script src="https://d3js.org/d3.v4.min.js" type="text/javascript"></script>
<script src="https://d3js.org/d3-selection-multi.v1.js"></script>

<script type="text/javascript">
  var folder = GetURLParameter("folder");
  var filename = "Graph"+GetURLParameter("graph");
  var colors = d3.scaleOrdinal(d3.schemeCategory10);
  var svg = d3.select("svg"),
            width = +svg.attr("width"),
            height = +svg.attr("height"),
            node, 
            link;

  svg.append('defs').append('marker')
      .attrs({'id': 'arrowhead',
              'viewBox': '-0 -5 10 10',
              'refX': 13,
```
Deterring Intellectual Property Thieves

```javascript
var simulation = d3.forceSimulation()
  .force("link", d3.forceLink().id(function (d) {
    return d.id;}).distance(100).strength(1))
  .force("charge", d3.forceManyBody())
  .force("center", d3.forceCenter(width / 2, height / 2));

function GetURLParameter(sParam)
{
  var sPageURL = window.location.search.substring(1);
  var sURLVariables = sPageURL.split('&');
  for (var i = 0; i < sURLVariables.length; i++)
  {
    var sParameterName = sURLVariables[i].split('=');
    if (sParameterName[0] == sParam)
    {
      return sParameterName[1];
    }
  }
}

filepath = folder+'/'+filename+'.json'
```
Deterring Intellectual Property Thieves

```javascript
    d3.json(filepath, function (error, graph) {
      if (error) throw error;
      update(graph.links, graph.nodes);
    })

    function update(links, nodes) {
      link = svg.selectAll(".link")
        .data(links)
        .enter()
        .append("line")
        .attr("class", "link")
        .attr('marker-end', 'url(#arrowhead)')

      link.append("title")
        .text(function (d) { return d.type;});

      edgepaths = svg.selectAll(".edgepath")
        .data(links)
        .enter()
        .append("path")
        .attrs({
          'class': 'edgepath',
          'fill-opacity': 0,
          'stroke-opacity': 0,
          'id': function (d, i) { return 'edgepath' + i}
        })
        .style("pointer-events", "none");

      edgelabels = svg.selectAll(".edgelabel")
        .data(links)
        .enter()
        .append("text")
        .style("pointer-events", "none")
        .attrs({
          'class': 'edgelabel',
          'id': function (d, i) { return 'edgelabel' + i},
          'font-size': 15,
          'fill': '#000'
        })
```

30 Chapter A Snow Kang
Deterring Intellectual Property Thieves

```javascript
node = svg.selectAll(".node")
    .data(nodes)
    .enter()
    .append("g")
    .attr("class", "node")
    .call(d3.drag()
        .on("start", dragstarted)
        .on("drag", dragged)
        //.on("end", dragended)
    );

node.append("circle")
    .attr("r", 8)
    .style("fill", function (d, i) { return colors(i);});

node.append("title")
    .text(function (d) { return d.id;});

node.append("text")
    .attr("dy", -3)
    .text(function (d) { return d.name;});

simulation
    .nodes(nodes)
    .on("tick", ticked);

simulation.force("link")
    .links(links);
```

Chapter A  

Snow Kang  

31
function ticked() {
    let radius = 6;
    node.attr("cx", function(d) { return d.x =
        Math.max(radius+20, Math.min(width - radius-50, d.x)); })
        .attr("cy", function(d) { return d.y =
        Math.max(radius+20, Math.min(height - radius-50, d.y)); });

    link
        .attr("x1", function (d) {return d.source.x;})
        .attr("y1", function (d) {return d.source.y;})
        .attr("x2", function (d) {return d.target.x;})
        .attr("y2", function (d) {return d.target.y;});

    node
        .attr("transform", function (d) { return "translate(" + d.x + ", " + d.y + ")";});

    edgpaths.attr(('d'), function (d) {
        var x1 = d.source.x,
            y1 = d.source.y,
            x2 = d.target.x,
            y2 = d.target.y,
            dx = x2 - x1,
            dy = y2 - y1,
            dr = Math.sqrt(dx * dx + dy * dy),

            // Defaults for normal edge.
            drx = dr,
            dry = dr,
            xRotation = 0, // degrees
            largeArc = 0, // 1 or 0
            sweep = 0; // 1 or 0

        /* Lines 184-209 handle the visualization for
        self-linking nodes and are from
        https://stackoverflow.com/questions
        /16358905/d3-force-layout-graph-self-linking-node
    })
Deterring Intellectual Property Thieves

```javascript
183 */
184 // Self edge.
185 if ( x1 === x2 && y1 === y2 ) {
186   // Fiddle with this angle to get loop oriented.
187   xRotation = -45;
188   // Needs to be 1.
189   largeArc = 1;
190   // Change sweep to change orientation of loop.
191   // sweep = 0;
192   // Make drx and dry different to get an ellipse
193   // instead of a circle.
194   drx = 30;
195   dry = 20;
196   // For whatever reason the arc collapses to a point if the beginning
197   // and ending points of the arc are the same, so kludge it.
198   x2 = x2 + 1;
199   y2 = y2 + 1;
200    } else {
201      drx = 0;
202      dry = 0;
203    }
204
205  return "M" + x1 + "," + y1 + "A" + drx + "," + dry + " " + xRotation + "," + largeArc + "," + sweep + " " + x2 + "," + y2; });
```

```javascript
212 edgelabels.attr('transform', function (d) {
213   if (d.target.x < d.source.x) {
214     var bbox = this.getBBox();
215     rx = bbox.x + bbox.width / 2;
216   }
```
Deterring Intellectual Property Thieves

```javascript
ry = bbox.y + bbox.height / 2;
return 'rotate(180 ' + rx + ' ' + ry + ')';
}
else {
    return 'rotate(0)';
}
});

function dragstarted(d) {
    if (!d3.event.active) simulation.alphaTarget(0.3).
        restart()
    d.fx = d.x;
    d.fy = d.y;
}

function dragged(d) {
    d.fx = d3.event.x;
    d.fy = d3.event.y;
}
</script>
</body>
</html>
```
Appendix B

Algorithm Code

```python
# In Jupyter Notebook's kernel.json,
# set "env": {"PYTHONHASHSEED":"0"} for reproducability

import networkx as nx
import json
import csv
import math
import time
from random import seed, shuffle, randint, sample
from itertools import chain

# # Load dataset

# SOURCE: https://github.com/villmow/datasets_knowledge_embedding/blob/master/FB15k-237/entity2wikidata.json
with open('entity2wikidata.json', 'r') as f:
    fb_dict = json.load(f)

get_ipython().run_line_magic('store', '-r fb_dict

def get_dict(txt):
    lines = txt.read().split("\n")
    new_dict = {}
    for line in lines:
        [ID, val] = line.split("\t")
        new_dict[ID] = val
    return new_dict

def load_data():
    if dataset == "Umls" or dataset == "Nations":
        entities = open(dataset + "/entities.txt","r")
```
relations = open(dataset + "relations.txt", "r")
triples = open(dataset + "/triples.txt", "r")
e_dict = get_dict(entities)
r_dict = get_dict(relations)
triples = triples.read().split("\n")

elif dataset == "FB15K":
    triples_temp = open("fb_train.txt", "r").read().split("\n")
    e_dict = fb_dict
    r_dict = []
    triples = []

    for t in triples_temp:
        [ent1, rel, ent2] = t.split("\t")
        if ent1 in e_dict and ent2 in e_dict:
            r_dict.append(rel)
            triples.append(t)

    return e_dict, r_dict, triples

# # Algorithm Functions

# Takes in a KG, returns its NetworkX graph and a dictionary where the key is an edge pair (obj1, obj2) and the value is a list of the edge's label(s)
def construct_DiGraph(KG):
    G = nx.DiGraph()
    G_labels = {}
    for t in KG:
        triple = t.split("\t")
        if dataset == "Nations" or dataset == "Umls":
            rel = triple[0]
            ent_1 = triple[1]
            ent_2 = triple[2]
        elif dataset == "FB15K":
            ent_1 = triple[0]
            rel = triple[1].split("/")[-1]
            ent_2 = triple[2]
        if (ent_1, ent_2) in G_labels:
            G_labels[(ent_1, ent_2)].append(rel)
        else:
            G_labels[(ent_1, ent_2)] = [rel]
        G.add_edge(ent_1, ent_2)
return G, G_labels

# G_0 is the original graph represented as a set of the original
# tab separated string lines
# D and A are some threshold percentages (expressed as integers)
# L is the set of possible edge label id's
# SizeU is the desired size of U
def generate_U(G0, D, A, L, SizeU):
    # To ensure that there are no identical triples
    U_set = set([frozenset(G0)])

    U = {}
    U[0] = G0

    while len(U) < SizeU:
        d = randint(0, D)
        a = randint(0, A)

        # Take a random KG from U, the set of G_0 and built KGs
        lst = list(U_set)
        shuffle(lst)
        G_prime = set(lst[:G_prime_size])

        # Create a directed graph using NetworkX library
        G_prime_graph, _ = construct_DiGraph(G_prime)
        G_prime_size = len(G_prime)
        G_prime_nodes = G_prime_graph.nodes()

        num_edges_to_delete = int(round(d/100 * G_prime_size))
        num_edges_to_add = int(round(a/100 * G_prime_size))

        # delete d% of edges in G'
        lst = list(G_prime)
        shuffle(lst)
        G_prime = set(lst[:G_prime_size-num_edges_to_delete])

        # add a% of edges to G'
        for _ in range(num_edges_to_add):
            new_edge = sample(G_prime, 1)[0] # initialize to a
            while new_edge in G_prime:
                lst = list(L)
                shuffle(lst)
                new_label = lst[0]
                lst = list(e_dict)
shuffle(lst)
new_endpoints = lst[:2]
if dataset == "Nations" or dataset == "Umls":
    new_edge = str(new_label) + "\t" + \
    str(new_endpoints[0]) + "\t" + \
    str(new_endpoints[1])
elif dataset == "FB15K":
    new_edge = str(new_endpoints[0]) + "\t" + \
    "addedEdge/" + new_label + "\t" + \
    str(new_endpoints[1])
G_prime.add(new_edge)
G_prime_size = len(G_prime)
if frozenset(G_prime) not in U_set and G_prime_size >= 8
   and G_prime_size <= 12 :
    U[len(U)] = G_prime
    U_set.add(frozenset(G_prime))
return U

def generate_GI(U, d, t):
    G_I = nx.Graph()
    # Add edge if distance between KGs falls within range t
    for i in range(len(U)):
        for j in range(i+1, len(U)):
            dist = d(U[i], U[j])
            if dist <= t[1] and dist >= t[0]:
                G_I.add_edge(i, j)
    return G_I

def jaccard_dist(KG_a, KG_b):
    KG_intersection = len(KG_a.intersection(KG_b))
    KG_union = len(KG_a.union(KG_b))
    if (KG_union == 0):
        return 1
    else:
        return 1-(KG_intersection / KG_union)

def clique_computation(G_I, K_0, n):
    G = G_I
    C = []
    C_nodes = set()
    C_graph = nx.DiGraph()
while nx.graph_clique_number(G) > n:
    maximal_cliques = list(nx.find_cliques(G))
    max_size_clique = n+1
    maximum_cliques = []

    for clique in maximal_cliques:
        clique_num_nodes = nx.number_of_nodes(clique)
        if clique_num_nodes == max_size_clique:
            maximum_cliques.append(clique)
        elif clique_num_nodes > max_size_clique:
            max_size_clique = clique_num_nodes
            maximum_cliques = [clique]

    shuffle(maximum_cliques)
    max_clique = set(maximum_cliques[0])

    # Construct max_clique_graph by finding the subgraph of G with max_clique's nodes
    max_clique_graph = G.subgraph(max_clique)

    # Add this max_clique to C
    C.append(max_clique)

    # Adds the nodes of max_clique to C_nodes
    C_nodes.update(max_clique)

    if K_0 in max_clique:
        return C

    G.remove_nodes_from(C_nodes)

return C

def clique_fakeKG(G_I, K_0, n):
    C = clique_computation(G_I, K_0, n)

    shuffle(C)
    for clique in C:
        if K_0 in clique:
            clique.remove(K_0)
        lst = list(clique)
        shuffle(lst)
        K = set(lst[:n])
        K.add(K_0)
        return K
212
213  \text{return set()}
Appendix C

Experimentation Code

```python
# # Generating Test Fakes

# ## Filter misleading data for Nations

def get_nation_data(triples):
    # ids of triples that are not misleading
    nation_rels = list(chain(range(0,12), range(162, 242), range(255, 283),
                             range(590, 613), range(677, 691), range(703, 735)))

    nation_triples = list(map(lambda x: triples[x], nation_rels))

    # ids of relationships that are not misleading
    nation_rels = [0, 5, 6, 8, 27, 33, 35]

    return nation_triples, nation_rels


def generate_fakes(i, D, A, k, U_size):
    global e_dict, r_dict, triples, time_taken
    seed(i)
    e_dict, r_dict, triples = load_data()

    if dataset == "Nations":
        nation_triples, nation_rels = get_nation_data(triples)
        G_0 = set()
        shuffle(nation_triples)
        G_0.update(nation_triples[:k])
        U = generate_U(G_0, D, A, nation_rels, U_size)
    elif dataset == "Umls":
        G_0 = set()
        shuffle(triples)
        G_0.update(triples[:k])
```

41
Deterring Intellectual Property Thieves

```
U = generate_U(G_0, D, A, set(range(len(r_dict))),
               U_size)

elif dataset == "FB15K":
    G_0 = set()
    shuffle(triples)
    G_0.update(triples[:k])
    U = generate_U(G_0, D, A, r_dict, U_size)

d = jaccard_dist
K_0 = 0
n = 3

for t in t_intervals:
    G_I = generate_GI(U, d, t)
    result = clique_fakeKG(G_I, K_0, n)
    if len(result) == 0:
        fails.append("fail")
        break
    else:
        KGs.append(result)
        intervals_dict[str(t)] = result

flat_KGs = list(set([item for KG in KGs for item in KG]))
shuffle(flat_KGs)

return U, flat_KGs, intervals_dict, fails
```

## Finding tests that contain 3 fake graphs per interval

```
def generate_tests(dataset):
    global results_txt
    U_size = 50

    all_tests = []
    for i in range(0, 22):
        results_txt += "\nSeed: {}
".format(i)
        print("i:", i)
        valid_tests = []
        for D in range(20, 60, 5):
            for A in range(20, 60, 5):
                for k in range(8, 13):
                    U, flat_KGs, intervals_dict, fails =
generate_fakes(i, D, A, k, U_size)
    if len(fails) == 0:
        valid_tests.append([i, D, A, k, U_size])
        break

results_txt += str(valid_tests) + "\n"

# pick one valid test per seed
shuffle(valid_tests)
all_tests.append(valid_tests[0])

return all_tests

# ## Run single test
def generate_summary(U, flat_KGs, intervals_dict, testnumber):
    summary_filename = "FakeKGTests/" + dataset + "/Test" + str(testnumber) + "/summary.txt"

    txt = "Dataset: {}
            Test number: {}\n".format(dataset, testnumber)
    txt += "\nOrder of graphs in test:\n";
    for idx in range(len(flat_KGs)):
        txt += "Graph {}: KG {}
             .format(idx+1, flat_KGs[idx])

    summary_lines = [txt,
                     "\nParameters:\n" , i is {},
                     D is {}, A is {}, k is {},
                     U_size is {}\n"
                     .format(i, D, A, k, U_size),
                     "\nIntervals:\n"]
    for idx in intervals_dict:
        summary_lines.append(idx + ": " + str(intervals_dict[idx]) + "\n")

    for idx1 in range(len(flat_KGs)):
        txt = "\nPairwise distances for Graph {} (KG {}):
           .format(idx1+1, flat_KGs[idx1])

    for idx2 in range(len(flat_KGs)):
        txt += "from Graph {} (KG {}): {}
             .format(idx2+1, flat_KGs[idx2],
                     jaccard_dist(U[ flat_KGs[idx1]], U
                     [flat_KGs[idx2]]))"
Deterring Intellectual Property Thieves

```python
summary_lines.append(txt)

summary_file = open(summary_filename, "w")
summary_file.writelines(summary_lines)
summary_file.close()

def generate_JSON(U, flat_KGs, testnumber):
    for a in range(1, len(flat_KGs) + 1):
        KG_id = flat_KGs[a - 1]
        json_links = []
        json_nodes = []

        G, G_labels = construct_DiGraph(U[KG_id])
        for node in G.nodes():

            if dataset == "Nations" or dataset == "Umls":
                new_node = {
                    "name": e_dict[node],
                    "id": node
                }
            elif dataset == "FB15K":
                new_node = {
                    "name": e_dict[node]["label"],
                    "id": node
                }
            json_nodes.append(new_node)

        for node_pair in G_labels:
            for relationship in node_pair:
                if dataset == "Nations" or dataset == "Umls":
                    new_link = {
                        "source": node_pair[0],
                        "target": node_pair[1],
                        "type": r_dict[relationship]
                    }
                elif dataset == "FB15K":
                    new_link = {
                        "source": node_pair[0],
                        "target": node_pair[1],
                        "type": r_dict[relationship].split("/")[1]
                    }

                json_links.append(new_link)

        json_dict = {
            "nodes": json_nodes,
            "links": json_links
        }
```

Chapter C

Snow Kang
Deterring Intellectual Property Thieves

```python

graph_filename = "FakeKGTests/" + dataset + "/Test" + str(testnumber) + "/Graph" + str(a) + ".json"

with open(graph_filename, 'w') as json_file:
    json.dump(json_dict, json_file)

def generate_questions():
    questions_filename = "FakeKGTests/" + dataset + "/questions.txt"

twentytwo_tests = all_tests
shuffle(twentytwo_tests)
five_tests = twentytwo_tests[:5]

questions_lines = ""
for test in five_tests:
    [i, D, A, k, U_size] = test
    U, flat_KGs, intervals_dict, fails = generate_fakes(i, D, A, k, U_size)

    original_graph = list(U[0])
    shuffle(original_graph)

    if dataset == "FB15K":
        [obj1, rel, obj2] = original_graph[0].split("\t")
        rel = rel.split("/")[1]
        obj1 = e_dict[obj1]["label"]
        obj2 = e_dict[obj2]["label"]
    else:
        [rel, obj1, obj2] = original_graph[0].split("\t")
        rel = r_dict[rel]
        obj1 = e_dict[obj1]
        obj2 = e_dict[obj2]

    questions_lines += obj1 + " and " + obj2 + " are related by " + rel + "\n"

    if dataset == "Nations":
        nation_triples, nation_rels = get_nation_data(nation_rels)
        lst = list(nation_rels)
    else:
        lst = list(r_dict)
```

Chapter C Snow Kang 45
Deterring Intellectual Property Thieves

```python
shuffle(lst)
answer_choices = set([rel])

while len(answer_choices) < 5:
    new_choice = lst.pop()
    if dataset == "FB15K":
        answer_choices.add(new_choice.split("/")[1])
    elif dataset == "Nations":
        answer_choices.add(r_dict[str(new_choice)])
    else:
        answer_choices.add(r_dict[new_choice])

answer_choices = list(answer_choices)
shuffle(answer_choices)
for choice in answer_choices:
    questions_lines += choice + "\n"

questions_lines += "\n"

questions_file = open(questions_filename,"w")
questions_file.writelines(questions_lines)
questions_file.close()

def generate_csv():
csv_filename = "FakeKGTests/" + dataset + "summary.csv"
csv_lines = ""

for test in all_tests:
    [i, D, A, k, U_size] = test
    U, flat_KGs, intervals_dict, fails = generate_fakes(i, D, A, k, U_size)

    for KG in flat_KGs:
        csv_lines += str(round(jaccard_dist(U[KG], U[0]), 2)) + "\n"

    csv_lines += "\n"

csv_file = open(csv_filename,"w")
csv_file.writelines(csv_lines)
csv_file.close()

dataset = "FB15K"
results_filename = "AllResults/" + dataset + "Results.txt"
results_file = open(results_filename,"w")
results_txt = "Dataset: {}
all_tests = generate_tests(dataset)
```
Deterring Intellectual Property Thieves

```python
244 testnumber = 1
245 for test in all_tests:
246     [i, D, A, k, U_size] = test
247     U, flat_KGs, intervals_dict, fails = generate_fakes(i, D, A, k, U_size)
248     generate_summary(U, flat_KGs, intervals_dict, testnumber)
249     generate_JSON(U, flat_KGs, testnumber)
250     testnumber += 1
251
252 results_file.writelines(results_txt)
253 results_file.close()
254
255
256 generate_questions()
257 generate_csv()
```