Database Deception using Large Language Models

Jason Landsborough
NIWC Pacific
San Diego, USA
0009-0003-5455-7293

Neil C. Rowe
Naval Postgraduate School
Monterey, USA
0000-0003-2612-0062

Thuy D. Nguyen
Naval Postgraduate School
Monterey, USA
0009-0009-1785-7776

***Abstract*—Cyberattackers often try to steal data in a cyberattack. Cyberdeception techniques such as supplying attackers with fake data (honeytokens) instead can interfere with their collection and exfiltration goals. This paper describes a design using a large language model to generate synthetic data from which defenders can create fictitious database records. We evaluated several models for this design using four SQLite databases, and tested the effectiveness of AI systems in identifying new data with two use cases: replacement of real records by fake records and shuffling fake and real records.**

***Index Terms*—cyberdeception, honeytoken, cyber security, artificial intelligence, synthetic data**

I. Introduction

Data breaches are a common result of a cyber attack. A well known incident was the 2020 U.S. government data breach by the Russian advanced persistent threat group APT29 [1]. This attack targeted multiple government agencies using vulnerabilities in commercial software from Microsoft, SolarWinds, and VMware. Detection began when FireEye discovered someone targeting them to steal their tools [2]. In 2019, LinkedIn had 92% of their userbase's information stolen [3]. In 2024, 2.9 billion records including personal information were stolen from the data broker National Public Data [4]. Data theft is a primary impetus for attackers motivated by financial gain or competitive advantages.

Generative artificial intelligence tools, and large language models in particular, can generate realistic data. We explored their use for generating database records, enabling defenders to offer attackers fake data that will make their data thefts worthless. Our threat model focused on data breach since protecting data is critical to many organizations. Data breaches can be damaging when they contain confidential information, details providing a competitive advantage, data requiring careful handling policies such as health records. Deception with false data can disrupt data breaches. An example is the tainting of data obtained in a breach during the 2017 French election, which made it difficult to identify true content [5]. Attackers typically seek various types of data, often from databases. It is therefore useful to create fake data for databases.

Our research makes the following contributions:

- A design for using large language models to generate synthetic database records based on an arbitrary existing database, given sample records and a schema.
- Qualitative and quantitative assessments of synthetic data generated by several large language models.
- An assessment of how well large language models themselves can detect synthetic database records.

II. Background

A. Defensive Cyberdeception.

There is an asymmetry in the favor of attackers over defenders often called the "defender's dilemma" [6]. One way to help defenders is influencing the attacker's decision process with cyberdeception [7]. Deception in cyberspace can exploit the long history of deception in other types of conflict [8]. Deception can either discourage attacks to protect important systems, or encourage attacks to learn information about their methods. An example of discouragement is wasting an attacker's time by providing too much data or deliberately causing long delays [9], [10]. The best known type of encouragement is a honeypot, a system to lure an attacker and observe what they do. Honeytokens are another useful tool for cyberdeception, which are fake objects or data that an attacker may interact with. Honeytokens can either encourage an attack or discourage them based on the defender's goals. Honeytokens for tabular data such as databases were studied by extracting the rules that describe a database constraint and turning them into a specification of a constraint-satisfaction problem [11].

B. Generative AI

Generative artificial intelligence (AI) has become popular. Using AI techniques to generate synthetic data has been done for some time, such as using Hidden Markov Models to generate speech data in the 1950's [12], and statisticians have long generated synthetic data from known distributions. Generative adversarial networks (GANs) can create "deepfakes" of realistic pictures, audio, and video, which can be used for pranks and fraud [13]. In late 2022, ChatGPT was launched publicly and has become the most popular generative artificial-intelligence tool. Since then, ChatGPT and other large language models have been used in many workflows, even taking the role of the cyberattacker to test cyberdefenses [14]. Some attackers use these tools to design attacks [15]. Generative AI based on large language models avoids the traditional statistical requirement of assuming independence of the data properties, by exploiting complex correlations observed in real data using neural networks called "transformers" [16]. However, these models have limitations, including vulnerabilities to bypass guardrails and obtain malicious responses from a model [17]. Another limitation is that models generate incorrect data called "hallucinations." It has been argued that these hallucinations are not lies because AI systems cannot intend to lie. They are a an over-extrapolation from their data [18]. Humans, however can use these systems to lie.

C. Related Work

Several projects have used large language models to create a shell honeypot. One tool used OpenAI's GPT-4o model [19]. Overall, the honeypot was found to be slower, more frustrating, and more difficult to use than the control machine. This experiment cost $2.97, or about $0.05 per user-minute. That sounds cheap, but for a large deployment or a long attack campaign, costs could add up. One participant saw an OpenAI API Python error, which led them to correctly identify that they were interacting with a large language model. Several common shell features were missing at the time of the experiment such as tab auto completion, reverse search, and real-time interactive tools.

Another group created a similar large language model shell honeypot [20]. They used the model *Llama3* with 8 billion parameters. They found larger models were too slow, and models designed for coding were ineffective for honeypot simulation. They tuned this model with a supervised fine-tuning approach on their own dataset of Linux commands. They found that their fine-tuned model's output had a small improvement compared with Cowrie, a popular shell honeypot, results as ground truth data over the *Llama3* model.

One group focused on using large language models to create honeytokens rather than a honeypot. The honeytokens they investigated included the contents of robots.txt, honeywords, services, an invoice file, a configuration file, a log file, and a database [21]. Their work focused mainly on data in the robots.txt and honeywords. The robots.txt file tells Web crawlers which directories it can traverse; the honeywords are username-password pairs. They found the generated robots.txt files were acceptable with some minor issues with GPT4 and Llama; the honeywords produced were acceptable, although Gemini claimed they were "not possible to execute." They did try to use an LLM to produce a database of user information, but found that the large language models struggled to generate acceptable content, with GPT4 and Gemini in the "not possible to execute" category.

III. Choosing Large Language Models

Honeypots are a good place to centralize fake records. Automation can reduce the overhead in generating it. Large language models are efficient for many text generation tasks, so we designed and implemented a way to use them to generate fake database data. Fake data can interfere with an adversary's data collection techniques and impede exfiltration [22].

We used a lightly coupled design where the large language models were treated as interchangeable black boxes. Fine-tuning models was out of scope for this work. This enables defenders to generate a variety of data, and no specific model dependency is required other than a prompt and response mechanism. It also enables comparison of behavior of different models. We used the following criteria when selecting models for this work:

Uses an Application Programming Interface (API). Instead of using a browser-based chat interface, we prompted the large language models with details about the data it should generate. This works for models using the Ollama platform (Table I in Appendix A), which share a common interface.

Output looks reasonable to human reviewers. We chose models that could produce database rows that looked reasonable. We did not do any statistical analysis at this stage.

Runs locally. We chose to run the model locally to maintain privacy and reduce cost. Organizations that choose to create fake data with a large language model may have legal non-disclosure constraints that prevent them from using models run by organizations such as OpenAI, Google, and so on. Running artificial intelligence models locally can also be cheaper because users can avoid API fees.

Runs on our graphics card. We used the Nvidia Geforce RTX 4080 card which has 16 gigabytes of video memory. By choosing smaller models that run on this card, we got acceptable response times. The ":#" convention in model names describes the number of parameters of the model, 'm' for millions and 'b' for billions which factor into model size.

Avoids unnecessary output. Some models such as *phi4:14b* produced commentary in the output despite our request for only database rows. Smaller models such as *llama3.2:1b* would often repeat the creation schema and provide an insertion SQL command, causing only a few good rows of data to be generated if at all.

Inadequate output. We ruled out the *smollm2:135m* model because it rarely produced rows and often had type errors such as text data for an integer column when it did. The larger *smollm2:360m* did not generate new data, often repeating the example data as new rows or generating repeated SQL queries. We ruled out *starcoder2:15b* because it seemed to struggle, despite our assumption that it might do well with databases because they are used in many programs for managing data. Occasionally it would just repeat the input prompt as a response. When it did return new row data, it repeated the example data and did not offer many results, some of which were just filled with placeholder data such as "<NAME>."

IV. Database Deception Design

Traditionally, statisticians generate synthetic data by assigning probability distributions for each column based on sample data, then generating new data randomly and independently from those distributions [16]. If the distribution is of words or phrases, the simplest estimate of probability is proportional to the frequency in a histogram. The weakness of this approach is that in most real data the values in different columns not independent. An advantage of generative AI is that it is based on more complex "transformation" models that include a deeper representation of context affecting a column value. Thus they could in principle do a better job.

The general architecture of our system is shown in Figure 1 (in Appendix A). The goal is to put synthetic (fake) data onto systems in the business operational environment (either real systems or honeypots) to interfere with the attacker if they compromise any of the systems. We tested a synthetic SQL database with all fake records, and a real SQL database with fake records.

We created the Synthetic Data Generator Python program that interacts with a local SQLite database to generate synthetic data for a deception planner. It takes as input the database, model to use, method of operation (of the two use cases), hints to bias the model output, limit of example rows, total number of rows to generate, and limit of rows to generate. Ollama's interface allowed us to easily switch between models.

For both use cases we used SQLite for our target database in our implementation due to its ease of use and compact nature. Changes may be needed for other database features such as authentication connecting to a database server. In both use cases we provided the same *prompt* to request new database data:

```
prompt = "Here are some example rows:\n" +
str(examples) + "\nColumns must be delimeted by: "
+ delimiter + "\nWithout providing any additional
commentary or other explanatory text, provide "
+ str(nRows) + " rows on separate lines (as in
each line is delineated by a newline character)
of new data " + hints + " for the following
SQL table schema:\n" + str(schema) + "\n"
```

The *examples* list contained some rows of existing database data to use as an example. Supplying examples gives the models relevant context and decrease the rate of erroneous row data, which is very useful when database table column names are ambiguous or are not very descriptive. A delimiter other than a comma helps with parsing. We chose example records randomly so the model gets different examples as context each time. The variable *nRows* is the number of rows the model should generate. We optionally insert nudges (recommendations) through *hints* to the model. The variable *schema* is the table specification information such as the table with names, types, constraints, keys, and so on. An example of schema information is:

```
schema = "['CREATE TABLE FinanceAccount (\n
ID INTEGER PRIMARY KEY,\n    customerName TEXT
NOT NULL,\n    customerAddress TEXT NOT NULL,\n
customerPhone TEXT NOT NULL,\n    managedBy
INTEGER,\n    accountID INTEGER NOT NULL,\n
accountBalance REAL NOT NULL,\n    FOREIGN KEY
(managedBy) REFERENCES Employee(ID)\n)']"
```

The prompt is given to the large language model as input. The model should use the example rows and database table information as context so it can generate and respond with a user-defined number of new row data for that table.

In both use cases, fake data can help security teams. Fake records can be tied to real-world indicators such as decoy accounts, which serve as high-confidence tripwires that a database has been breached. Also, these fake databases or records could be re-generated on an interval which if found in an external location such as the Dark Web could give incident responders valuable intelligence on when the breach occurred.

A. Replacement of Real Records by Fake Records

The first use case creates a completely fake version of the database using the same tables and filled with the same number of rows per table; this can be used when defenders want no real data to be stolen. Our method is shown in Algorithm 1. For a given table *tbl*, we save the table schema to *s* to be included in the prompt.

We create a new copy of the database using the real database schema. If the original table is not empty, we start the process for generating new row data by querying the real database for several rows to use as example data in the prompt. We limit the number of examples to *n*, a global variable supplied by user input to the program. We request *iterN* (a user input) new rows of data from the large language model and adjust *nRows* if needed.

Algorithm 1 Fake replacement algorithm

```
function FAKECOPY(tbl, llm, hints, copyDb)
    $s \leftarrow$ getSchema(tbl)
    $pk \leftarrow getPrimaryKey(tbl)$
    $cols \leftarrow$ getTableColumns(tbl)
    $pki \leftarrow cols.index(pk)$
    $tblRowCount \leftarrow$ getTableCount(tbl)
    $createSQLTable(copyDb, s)$
    if $tblRowCount \neq 0$ then
        $rList \leftarrow []$
        $count \leftarrow 0$
        $lim \leftarrow n$
        while count < tblRowCount do
            $data \leftarrow getRandTableData(tbl, lim)$
            $data \leftarrow filterBlobsFromRows(data)$
            $nRows \leftarrow iterN$
            if tblRowCount - count < nRows then
                $nRows = tblCount - count$
                $rowList \leftarrow$
                $genRows(s, data, nRows, hints, llm)$
                for row in rList do
                    $row[pki] \leftarrow count + 1$
                    if insertRow(copyDb, tbl, cols, row then
                        $count \leftarrow count + 1$
```

B. Shuffled Fake and Real Records

The second use case is adding fake records into an existing database. This helps when it is not feasible to offer an entirely fake database for an attacker to steal. The main algorithm is shown in Algorithm 2.

Algorithm 2 Shuffled fake and real records algorithm

```
function FAKECHAFF(dbName, llm, hints)
    $dbDict \leftarrow dbToDictionary(dbName)$
    $map \leftarrow remapRecords(dbDict)$
    $tables \leftarrow getTableNames(dbName)$
    for tbl in tables do
        $shuffleRecords(dbName, tbl, dbDict, map)$
    $saveRealKeys(dbName, map)$
    for tbl in tables do
        $addFakeRows(tbl, llm, nRows, [hints, maxIndex])$
```

The most important function call is that to *remapRecords()* which is shown in Algorithm 3. This function creates a new mapping, saved to *map*, from the old primary key to the new random primary key. We use the mapping in the *shuffleRecords()* function shown in Algorithm 4 for each table *tbl*. This function is much like the *fakeCopy()* function shown in Algorithm 1. The main differences are that we add several fake rows to insert instead of the same number of original rows and we randomize the primary key values before insertion instead of incrementing the value. Insertion also occurs in the same database instead of a separate copy, so we do not create a new SQL table.

Algorithm 3 Remapping records algorithm

```
function REMAPRECORDS(dbDict)
    $map \leftarrow \{\}$
    for tbl in dbDict do
        $pks \leftarrow getPrimaryKeyNames(tbl)$
        if len($pks$) $> 1$ then
            skip
        else if len($pks$) is $1$ then
            $cols \leftarrow getTableColumns(tbl)$
            $pki \leftarrow cols.index(pk[0])$
            $maxInt \leftarrow 2^{31} - 1$
            $newPk \leftarrow randomInt(1, maxInt)$
            while newPk in map[tbl].values() do
                $newPk \leftarrow randomInt(1, maxInt)$
    return map
```

Algorithm 4 Shuffling records algorithm

```
function SHUFFLERECORDS(dbName, tbl, dbDict, map)
    $pks \leftarrow getPrimaryKeysNames(tbl)$
    $pki \leftarrow None$
    $cols \leftarrow getTableColumns(tbl)$
    if len(pks) is 1 then
        $pk \leftarrow pks[0]$
        $pki \leftarrow cols.index(pk)$
    $deleteAllRows(tbl)$
    for row in dbDict[tbl] do
        if len(pks) is 1 then
            $row[pki] \leftarrow map[tbl][row[pki]]$
        $fks \leftarrow getForeignKeyTables(tbl)$
        for tblName in fks do
            $fki \leftarrow cols.index(fks[tblName]$
            if row[fki is not None] then
                $row[fki] \leftarrow map[tblName][row[fki]$
        $insertRow(dbName, tbl, cols, row)$
```

V. Test Database Schema

We tested our approach with two open-source databases and one custom database.

Northwind Database. A small business schema that was introduced as an example by Microsoft for Access 95-2003 that represents a company that ships grocery items [23]. We use a SQLite version [24]. A database like this would be interesting for attackers to disrupt or use a supply chain in a future attack. It has tables for suppliers, categories of products, products, orders, order details, employees, employee territories, territories, regions, shippers, customers, and customer demographics. There are 625,896 rows in all tables.

Chinook Database. A music industry schema that is available for multiple database engines and represents a digital media store [25]. This could be appealing to an attacker looking to steal music before it is officially released [26]. The schema contains the tables for artist, album, track, mediaType, playlist, playlistTrack, genre, invoiceLine, invoice, customer, and employee. There are 15,607 rows in all Chinook tables.

Financial Company Database. We created a small database, that we call *Breeze*, representing an asset management company shown in Figure 2 (in Appendix A). This company manages accounts for clients. There are 24 rows across all tables. An attacker could use this information to target either the customer of an account or the employee that manages a lucrative account in a future attack campaign.

Real-world Utility Database. To further validate our approach, we used a subset of the Public Utility Data Liberation (PUDL) project data, which contains real-world data from utility reporting in the United States [27]. The entire database currently has 89,437,360 rows across all tables. A database of utility data would help an attacker planning to cause a critical-infrastructure disruption in a specific region.

VI. Results

A. Model Metrics

We compared the performance metrics for several models generating data using each database. The Northwind database had the most complicated tables; Chinook had some complicated tables; and Breeze had fairly simple tables. For a consistent comparison, we generated 100 rows for each table in each database with each model using at most five example rows in each prompt. Tables II, III, and IV (in Appendix A) show statistics on the results. Success is measured in percentage of rows that satisfy the schema and can therefore be inserted.

We did not disallow using example content for generated data; we merely asked for new data. Models sometimes used portions of the original data mixed with some new data to create a new row. To check how often successfully inserted LLM database data differed from the source material, we evaluated each model's performance using a "uniqueness" percentage also shown in Tables II, III, and IV (in Appendix A). We compare the new data to the original source data. We focus on the string values because integer values can often be repeated such as key id numbers for each table. Phone numbers and addresses are treated as strings for our purposes because they have non-numeric characters. We compare the number of unique strings in both databases to get the number of strings that are unique to the new database. Dividing this number by the total unique strings in the new database gives us the uniqueness percentage.

The gemma3 models did the best, although it is surprising that *gemma3:4b* generated valid data as well as the *12b* variant. The *llama3:8b and llama3.1:8b* models did very well. It is also interesting that the *llama3.2:3b* model did well despite being less than half the size of the other two Llama models and the smallest model we used. The *llava:13b* model and mistral:7b models did poorly in our tests.

B. Exploration of the Data Generated

Effectiveness of the generated data for deception will be tested more carefully in future work. But for now, we can make some observations. Most models performed well.

With the smallest model, llama3.2:3b, we observed some odd results where a small number of rows in the Chinook database provide number instead of a date for InvoiceDate and give the date instead under BillingAddress.

Smaller models also struggled more with generating realistic numbers. They often repeated digits such as 7777 or used a sequence of numbers such as in "456 Oak St."

The most common error that led to failed inserts was a row that looked reasonable but did not match the required number of columns required for a full record.

The hints ("nudge") mechanism attempts to influence the model response. Cases exist where defenders may want to use obvious fake data, such as in the French presidential election [5], to instill doubt in the attacker as to whether target data is real. To test this, we used the Chinook database and created a "very fake" database copy using as a hint "cats are funny" to skew the large language model results to generate less realistic, but not gibberish, data. This sort of data could be used in a discouraging honeypot that gets adversaries to leave because it appears to be obviously fake. Some of the artist names that were inserted into the database were:

```
1 |Cats are funny
2 |Garfield
3 |The Pussycat Dolls
4 |Kitty Purry
5 |Furry Felines
6 |Whiskers and Wits
7 |Purrfect Storm
8 |Meowzart
```

Of these, only two were real artists: Garfield, and The Pussycat Dolls. This also demonstrates that simple hint nudges can steer the large language model response without needing to customize the system and assistant roles or go through greater effort to fine-tune a model.

C. Shuffling

We also examined the results of shuffling (permuting) the small finance company database *Breeze* and inserting fake records among the real. We started with five employees seen in the "select all" statement. After shuffling the records and inserting new fake records, we got 15 employees (real employee records are marked with double asterisks):

```
80999381 |109 |Julian White ||Manager
294098349 |107 |Ryan Brooks |408-901-2345 |Analyst
922878050 |103 |Ava Taylor |949-765-4321 |Intern
1066670999 |101 |Keanu Reeves |650-416-7899 |Manager**
1438145888 |103 |Annie Robertson ||Analyst**
1498633672 |111 |Michael Kim |310-765-4321 |Manager
1700628308 |108 |Emily Chen |818-456-1234 |Manager
1713268347 |108 |Sophia Brown |415-234-5678 |Analyst
1961761195 |107 |Emily Chen |310-456-1234 |Intern
1983755203 |106 |Sofia Patel ||Intern
1993677995 |102 |Lawrence Meyers |424-654-3210 |Analyst**
2033814233 |109 |Olivia Hall |916-890-1234 |Analyst
2058148069 |105 |John Kang |424-309-5467 |Analyst**
2068128765 |104 |David Lee ||Manager
2107692453 |104 |Alice Johnson |424-435-6914 |Analyst**
```

When we joined the FinanceAccount and Employee tables, we saw that Lawrence Meyers managed five accounts. In the deceptive version, we found that he managed the original five as well as ten more. These fake accounts could be used as tripwires to let the company know that their database is being accessed by an attacker.

D. Detecting Model Deception

Attackers can use artificial intelligence of their own. We already see this happening with tools like pentestGPT [14]. However, most models would not do well at detecting AI-generated content, as specialized AI-generated text detection tools are not very reliable [28]. Still, we can test our output.

We tested each of the candidate models against 100 rows each from three variants of the Chinook database's artist table as well as from each of the three variants of the Northwind database's Orders table. The first variant was the original database. The second variant was the version of the database with generated records using the approach from Section IV-A. The third variant was created using the same approach as the second but with the hint "indoor house plants" for the Chinook and "Emotions, feelings, and human experiences" for the Northwind to generate less realistic results. We hoped that plants would seem strange for artist names and emotions were not businesses creating orders. The second and third databases were generated using the *llama3.1:8b* model. The schema is also supplied to give additional context since an attacker could obtain it.

We used the template below within the prompt as well as a row to test:

```
template = "Without providing any additional
commentary or other explanatory text, output
FAKE if you think it is AI generated, and REAL
otherwise.\n Here is the schema of the table:\n"
+ str(schema) + "\nHere is the row to check:\n"
```

We repeated this for 100 rows from all databases, with no prior chat history or results included to ensure test independency.

Tables V and VI (in Appendix A) show statistics on the results of this test. The *llama3:8b*, *llama3.2:3b*, and *gemma3:4b* models were very gullible, guessing that just about every row was real. The *gemma3:12b* model identified the "very fake" data more often than many of the other models in the Nortwind database case, but did not do as well in the Chinook database case. The *llava:13b* model may have guessed randomly, because it was close to 50% for all cases.

This suggests that models like the Llama line would be very easy to fool with deception. Models like *phi4:14b* may be harder to fool, but one could take advantage of their accuracy to deter an attacker by including obviously fake data in a database to make them think an operational system is a honeypot. The *llava:13b* model should not be used by either attackers or defenders.

Members of our team also assessed some of the data. They were instructed to determine if the data was original (real) or AI-generated (fake) and were given the database schema with an example row much like we did with the large language model tests. We chose five rows from each of the three variants of both the Chinook and Northwind databases that were part of the large language model tests for a total of 30 questions. The 12 participants were multi-disciplinary members of our cyberdeception team, most of which were familiar with this work and most had a strong background in cybersecurity. Many members only had some experience with databases, as it is not their normal area of expertise.

Table VII (in Appendix A) shows the results from our team. The selection options for real or fake were randomized to avoid skewing the results through a default choice bias if someone was just selecting the first choice if they were unsure. We did not do well identifying the real records. We did do well identifying the "very fake" records.

This was a difficult task. For the Chinook database, many artists the model produced were real music artists. Because musicians can sometimes have strange names, it was rather difficult to identify some of the "very fake" artists. This also applies to human names because some people have rare names or unusual variations of names. For example, it is reasonable that an artist could name themselves after a plant such as "Amaryllis" which was generated with the "indoor house plant" hint. We had greater success for the Northwind database using the "Emotions, feelings, and human experiences" hint. The model generated realistic results for the fake database, but with the hint we see several oddities such as for the company "Feeling Euphoric." The timestamps were all on the hour and produced a fake-sounding city and zip code.

```
 (134, JOY, 2, 2023-04-01 18:00:00, 2023-04-02
19:00:00, 2023-04-03 20:00:00, 5, 110.25,
Feeling Euphoric, 654 Pine St, Towntown, Asia,
12345, Japan)
```

This seems promising to make data that looks fake as a deceptive deterrence strategy.

E. PUDL Results

Due to its large size, we only used one table in the PUDL database to limit the runtime of our test . We chose the 'core_eia923__monthly_generation_fuel_nuclear' table because an attack on a nuclear facility would have temporary regional outage impacts but potentially also long-lasting environmental impacts like the disaster at Chernobyl near Pripyat, Ukraine. This table has 28,437 rows containing data from the EIA 923 form, a power plant operations report that details energy generation and fuel consumption.

For this validation, we chose 'gemma3:12b' which we found performed the best for insertion success rate as well as execution time in the three other databases. Using the replacement algorithm to create entirely fake records, it had a 99.80% successful insertion percentage and took eight hours and 29 minutes to complete. Using the shuffled fake and real records algorithm, it had a 99.99% successful insertion percentage and took 18 hours and 26 minutes to add twice as many records as there originally were leading to a total of 85,311 rows.

The data generated looked reasonable, with some outliers. For example, a date was generated for the year 2049 which has not happened yet. For some databases, a future date may make sense, but this table contains record of submitted reports. LLMs sometimes struggle to generate random looking numbers, so some of the values for energy generation or fuel consumption were obviously fake such as 1234567.8. However, to convince an attacker that this database is fake, results like these may be a benefit.

F. Limitations

Because most of our models are text-based, they struggled with binary data (such as pictures for product categories or employee photos in the Northwind database) so we removed that content when giving example rows to the LLM. Our algorithms could be changed to use a model that can describe pictures, such as *llava*, to generate descriptions of the example, which are then given to the text-based LLM to create a relevant description of a new picture for the rows it is creating. The generated description can then be passed to an image-generating model.

We only support shuffling records for keys with integer values. The PUDL table we used referenced some other tables that have alphabetic codes, such as NUC for Nuclear, so they were not changed. creating new random codes for these rows would have made the database much harder to use. It may be desirable in other databases to randomize alphabetic or alphanumeric keys. Similarly, it may be desirable to not randomize some integer keys if they correspond to some entity external to the database such as employee ID numbers.

Improvements can be made to handling the responses from the models. The models that generate SQL code could be either filtered or parsed to ensure these statements are not included as row data. Similarly, simple phrases that models tend to use as a response instead of data could be filtered out.

Although our trial with humans indicate that the fake data generated was hard to detect, it remains to be seen what attackers would do if they encountered fake but realistic or obviously fake data.

There is a risk to using data deception approaches like ours. Because deception interferes with an attacker's thought process, there may be unexpected second or third order effects to the deception. For example, in the asset management database example, an attacker could get fooled by a fake account and try to access it. However, if the fake account shows considerable money, the attacker could try to kidnap the manager of that account for ransom in the real world.

VII. Conclusion

Our design shows promise with using any compatible large language model to generate fake database records. Compared to randomly selecting values from statistical models, generative large language models can model complex correlations between data columns. Model performance varied, and the *gemma3* models offered the best performance in terms of both how likely the output could be used as well as the time it took to generate data. We also found that large language models did not do well at detecting if database records were generated by a large language model, which is something defenders can exploit.

Future work will try to measure the effectiveness of the generated data by comparing it to known distributions of numeric data and known histograms of symbolic data or through embedding comparisons. If generative AI is using a too-small sample of the real world, it should show weaknesses like those mentioned in Section VI-B, which we can detect using statistical hypothesis testing.

References

[1] Mandiant. (2022) Assembling the russian nesting doll: Unc2452 merged into apt29. [Online]. Available: https://www.mandiant.com/resources/blog/unc2452-merged-into-apt29url

[2] L. H. Newman. (2020) Russia's fireeye hack is a statement—but not a catastrophe. [Online]. Available: https://www.wired.com/story/russia-fireeye-hack-statement-not-catastrophe/

[3] Securiti. (2025) A comprehensive analysis of the biggest data breaches in history and what to learn from them. [Online]. Available: https://securiti.ai/analysis-of-the-biggest-data-breaches-in-history-and-what-to-learn/

[4] Microsoft. (2025) National public data breach: What you need to know. [Online]. Available: https://support.microsoft.com/en-us/topic/national-public-data-breach-what-you-need-to-know-843686f7-06e2-4e91-8a3f-ae30b7213535

[5] CSIS. (2018) Successfully countering russian electoral interference. [Online]. Available: https://www.csis.org/analysis/successfully-countering-russian-electoral-interference

[6] Google, "Secure, empower, advance: How ai can reverse the defender's dilemma," Google, Tech. Rep., 2024.

[7] K. Ferguson-Walter, S. Fugate, J. Mauger, and M. Major, "Game theory for adaptive defensive cyber deception," in *Proceedings of the 6th Annual Symposium on Hot Topics in the Science of Security*, 2019, pp. 1–8.

[8] N. C. Rowe and J. Rrushi, *Introduction to cyberdeception*. Springer, 2016.

[9] K. Ferguson-Walter, T. Shade, A. Rogers, M. C. S. Trumbo, K. S. Nauer, K. M. Divis, A. Jones, A. Combs, and R. G. Abbott, "The Tularosa study: An experimental design and implementation to quantify the effectiveness of cyber deception." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), Tech. Rep., 2018.

[10] J. Landsborough, L. Carpenter, B. Coronado, S. Fugate, K. Ferguson-Walter, and D. Van Bruggen, "Towards self-adaptive cyber deception for defense." in *HICSS*, 2021, pp. 1–10.

[11] A. Shabtai, M. Bercovitch, L. Rokach, Y. Gal, Y. Elovici, and E. Shmueli, "Behavioral study of users when interacting with active honeytokens," *ACM Transactions on Information and System Security (TISSEC)*, vol. 18, no. 3, pp. 1–21, 2016.

[12] Y. Cao, S. Li, Y. Liu, Z. Yan, Y. Dai, P. S. Yu, and L. Sun, "A comprehensive survey of ai-generated content (aigc): A history of generative ai from gan to chatgpt," 2023. [Online]. Available: https://arxiv.org/abs/2303.04226

[13] M. S. Rana, M. N. Nobi, B. Murali, and A. H. Sung, "Deepfake detection: A systematic literature review," *IEEE access*, vol. 10, pp. 25 494–25 513, 2022.

[14] G. Deng, Y. Liu, V. Mayoral-Vilches, P. Liu, Y. Li, Y. Xu, T. Zhang, Y. Liu, M. Pinzger, and S. Rass, "Pentestgpt: An llm-empowered automatic penetration testing tool," *arXiv preprint arXiv:2308.06782*, 2023.

[15] G. T. Intelligence. (2025) Adversarial misuse of generative ai. [Online]. Available: https://cloud.google.com/blog/topics/threat-intelligence/adversarial-misuse-generative-ai

[16] S. Nikolenko, *Synthetic data for deep learning*. Springer, 2021.

[17] Z. Xu, Y. Liu, G. Deng, Y. Li, and S. Picek, "Llm jailbreak attack versus defense techniques–a comprehensive study," *arXiv e-prints*, pp. arXiv–2402, 2024.

[18] P. Mishra. (2023) Chatgpt3 is bulls*** artist. [Online]. Available: https://punyamishra.com/2023/03/02/chatgpt3-is-bulls-artist/

[19] S. Johnson, R. Hassing, J. Pijpker, and R. Loves, "A modular generative honeypot shell," in *2024 IEEE International Conference on Cyber Security and Resilience (CSR)*. IEEE, 2024, pp. 387–394.

[20] H. T. Otal and M. A. Canbaz, "Llm honeypot: Leveraging large language models as advanced interactive honeypot systems," in *2024 IEEE Conference on Communications and Network Security (CNS)*. IEEE, 2024, pp. 1–6.

[21] D. Reti, N. Becker, T. Angeli, A. Chattopadhyay, D. Schneider, S. Vollmer, and H. D. Schotten, "Act as a honeytoken generator! an investigation into honeytoken generation with large language models," in *Proceedings of the 11th ACM Workshop on Adaptive and Autonomous Cyber Defense*, 2024, pp. 1–12.

[22] J. Landsborough, T. Nguyen, and N. Rowe, "Retrospectively using multilayer deception in depth against advanced persistent threats," 2024.

[23] Arrow of Time. Test datasets. [Online]. Available: https://arrow-of-time.com/comm$_testdata.aspx$

[24] jpwhite3. (2023) northwind-sqlite3. [Online]. Available: https://github.com/jpwhite3/northwind-SQLite3

[25] lerocha. (2024) chinook. [Online]. Available: https://github.com/lerocha/chinook-database

[26] J. Rhysider. (2024) Darknet diaries episode 148: Dubsnatch. [Online]. Available: https://darknetdiaries.com/episode/148/

[27] C. Cooperative. title. [Online]. Available: The Public Utility Data Liberation (PUDL) Project

[28] D. Weber-Wulff, A. Anohina-Naumeca, S. Bjelobaba, T. Foltýnek, J. Guerrero-Dib, O. Popoola, P. Šigut, and L. Waddington, "Testing of detection tools for ai-generated text," *International Journal for Educational Integrity*, vol. 19, no. 1, Dec. 2023. [Online]. Available: http://dx.doi.org/10.1007/s40979-023-00146-z

Appendix A: Tables and Figures

Model Name	<16 GB?	Reasonable output?
gemma3:4b	Y	Y
gemma3:12b	Y	Y
gemma3:27b	N	-
llava:13b	Y	Y
llava:34b	N	-
llama3:8b	Y	Y
llama3.1:8b	Y	Y
llama3.1:70b	N	-
llama3.1:405b	N	-
llama3.2:3b	Y	Y
llama3.3:70b	N	-
mistral:7b	Y	Y
phi4:14b	Y	Y
smollm2:135m	Y	N
smollm2:360m	Y	N
smollm2:1.7b	Y	N
starcoder2:15b	Y	N

TABLE I
Initial model evaluation

Model Name	Size (GB)	Rows Generated	Failed Inserts	% Success	Time (min)	% Unique
llama3:8b	4.7	1,350	250	81.48	8	91.97
llama3.1:8b	4.9	1,675	575	65.67	13	90.39
llama3.2:3b	2.0	1,804	704	60.98	7	77.99
mistral:7b	4.1	2,010	910	54.73	13	95.01
gemma3:4b	3.3	1,101	1	99.91	6	94.84
gemma3:12b	8.1	1,160	60	94.83	13	95.00
phi4:14b	9.1	1,333	233	82.52	14	87.71
llava:13b	8.0	2,393	1,293	45.97	35	78.87

TABLE II
Performance results for Northwind database generation

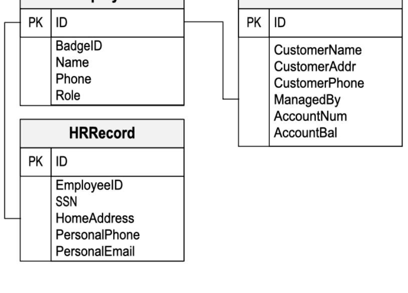

Fig. 2. Asset management company tables

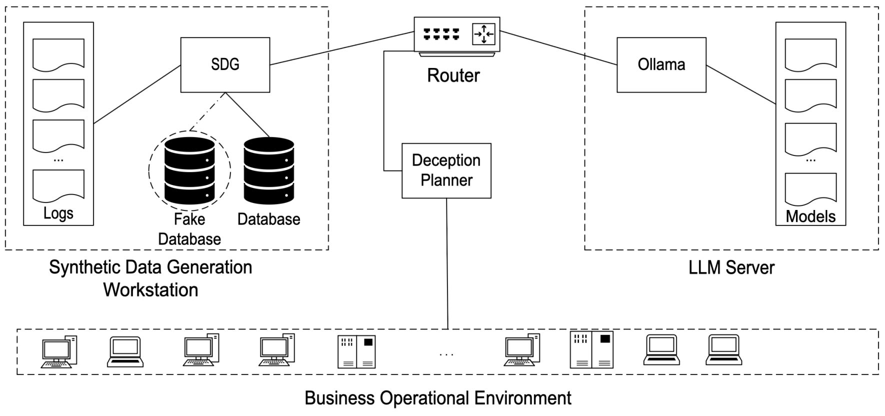

Fig. 1. Example deception system architecture

Model Name	Size (GB)	Rows Generated	Failed Inserts	% Success	Time (min)	% Unique
llama3:8b	4.7	1,261	161	87.23	7	89.86
llama3.1:8b	4.9	1,198	98	91.82	5	90.73
llama3.2:3b	2.0	1,201	101	91.59	3	81.25
mistral:7b	4.1	1,933	833	56.91	12	89.66
gemma3:4b	3.3	1,100	0	100	4	86.99
gemma3:12b	8.1	1,100	0	100	9	92.57
phi4:14b	9.1	1,189	89	92.51	8	91.91
llava:13b	8.0	2,486	1,386	44.25	31	82.79

TABLE III
PERFORMANCE RESULTS FOR CHINOOK DATABASE GENERATION

Model Name	Size (GB)	Rows Generated	Failed Inserts	% Success	Time (min)	% Unique
llama3:8b	4.7	311	11	96.46	1	98.88
llama3.1:8b	4.9	307	7	97.72	1	99.27
llama3.2:3b	2.0	310	10	96.77	1	88.37
mistral:7b	4.1	694	394	43.23	4	98.00
gemma3:4b	3.3	301	1	99.67	1	98.39
gemma3:12b	8.1	301	1	99.67	3	99.14
phi4:14b	9.1	317	17	94.64	2	99.03
llava:13b	8.0	344	44	87.21	3	89.71

TABLE IV
PERFORMANCE RESULTS FOR BREEZE DATABASE ROW GENERATION

Model Name	% Real	% Fake	% Very fake
llama3:8b	100	0	0
llama3.1:8b	81	16	17
llama3.2:3b	100	0	0
mistral:7b	64	10	39
gemma3:4b	100	0	0
gemma3:12b	82	1	11
phi4:14b	70	33	56
llava:13b	60	46	37

TABLE V
PERCENT AI CORRECT GUESSES FOR CHINOOK DATABASE

Model Name	% Real	% Fake	% Very fake
llama3:8b	100	1	7
llama3.1:8b	94	8	14
llama3.2:3b	99	0	1
mistral:7b	100	1	7
gemma3:4b	100	3	18
gemma3:12b	100	2	46
phi4:14b	86	19	62
llava:13b	65	41	36

TABLE VI
PERCENT AI CORRECT GUESSES FOR NORTHWIND DATABASE

Database	% Real	% Fake	% Very fake
Chinook	78.18	7.27	61.81
Northwind	68.09	40.0	83.63

TABLE VII
PERCENT HUMAN CORRECT GUESSES FOR EACH DATABASE