Ettu: Analyzing Query Intents in Corporate Databases

Gokhan Kul, Duc Luong, Ting Xie, Patrick Coonan,
Varun Chandola, Oliver Kennedy, Shambhu Upadhyaya
University at Buffalo
{gokhanku, ducthanh, tingxie, pcoonan, chandola, okennedy, shambhu}@buffalo.edu

ABSTRACT

Insider threats to databases in the financial sector have become a very serious and pervasive security problem. This paper proposes a framework to analyze access patterns to databases by clustering SQL queries issued to the database. Our system Ettu works by grouping queries with other similarly structured queries. The small number of intent groups that result can then be efficiently labeled by human operators. We show how our system is designed and how the components of the system work. Our preliminary results show that our system accurately models user intent.

Keywords

Insider Threats; Databases; Clustering

1. INTRODUCTION

It is increasingly important for organizations to be able to detect and respond to cyber attacks. An especially difficult class of cyber attack to detect is the so called *insider attacks* that occur when employees misuse legitimate access to a resource like a database. The difficulty arises because apparently anomalous behavior from a legitimate actor might still have legitimate intent. For example, a bank teller in Buffalo who withdraws a large sum for a client from California may be acting legitimately (e.g., if the client has just moved and is purchasing a house), or may be committing fraud.

The "U.S. State of Cybercrime Survey" [1] states that 37% of organizations have experienced an insider incident and that only 3% of these cases were reported to authorities. Many of the remaining incidents could not be prosecuted due to lack of evidence. A 2015 study [10] identified insider attacks as the most costly attack type among all attack types surveyed. According to the report, financial sectors were the hardest hit with the highest annualized cost over all industry sectors. Worse still, the average response time to an insider attack is 54.5 days, the longest of any attack type surveyed.

The challenge of addressing of insider attacks lies in the difficulty of precisely specifying access policies for shared resources such as databases. Coarse, permissive access policies provide opportunities for exploitation. Conversely, restrictive fine-grained policies are expensive to create and limit a legitimate actor's ability to adapt to new or unexpected tasks. In practice, enterprise database system administrators regularly eschew fine-grained database-level access control. Instead, large companies commonly rely on reactive strategies that monitor external factors like network activity patterns and shared file transfers. In a corporate environment, monitoring user actions requires less preparation and gives users a greater degree of flexibility. However, external factors do not always provide a strong attestation of the legitimacy of a database user's actions. Instead, we propose an approach that attempts to infer a user's intent from their interactions with the database. Actions inconsistent with known acceptable intents can be disallowed, or flagged for inspection by an administrator.

This paper is organized as follows. We discuss the related work that creates a basis for our research in Section 2. We introduce our core contribution, a technique for query intent modeling and describe Ettu[1], a system that uses query intent modeling as a way to flag potential insider attacks in Section 3. In Section 4 we evaluate the feasibility of query intent modeling, and conclude by identifying the steps needed to deploy query intent modeling into practice in Section 5.

2. RELATED WORK

The basic idea behind our system is to profile normal user behavior, detect suspicious behavior using this information, and distinguish malicious behavior from benign intents [5]. Indeed, this idea is not new; there are many anomaly detection systems focusing on suspicious behavior of users. Specific examples focus on file transfers [9], online and social behavior [2], command-line statements [8] and SQL queries issued to a database [7]. As the basic unit of interaction between a database and its users, the sequence of SQL queries that a user issues effectively models the user's behavior. There are different approaches to understand the intents behind SQL queries. One approach relies on the syntax of queries [4] and permits fast query validation. Another method is to use a data–centric approach, which performs better in detecting anomalies [7]. However, when the data contained in the database is not available to the detection system, it is impossible to use such an approach.

[1]Ettu is derived from the last words of the Roman emperor Julius Caesar, "*Et tu, Brute?*" in Latin, meaning "*You, too, Brutus?*" in English to emphasize that this system is meant to detect the unexpected betrayals of trusted people.

3. SYSTEM OUTLINE

Ettu operates in three stages: (1) An offline **clustering** phase where query logs are aggregated and summarized so that they can be easily examined, (2) A semi-automated **labeling** phase to identify potential signs of insider attacks, and (3) A **pattern-generation** phase that creates pattern matchers that screen queries online as they are processed by the database. These stages are illustrated in Figure 1.

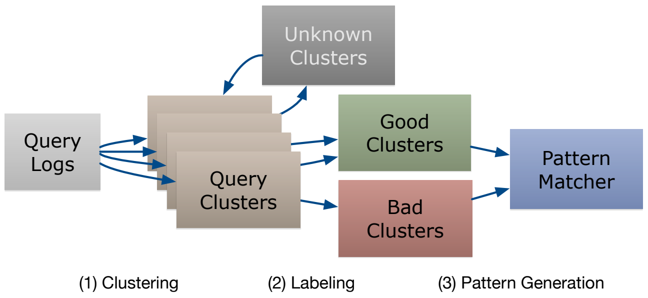

Figure 1: The typical Ettu workflow.

The initial input to Ettu is a log of query activity processed by the target database. The log is annotated with supplemental metadata like usernames and timestamps. The goal of the first phase is to produce a concise, but precise summary of the log. To summarize the log, queries are clustered together into "similar" groups. Each group consists of a set of queries that are issued with similar goals in mind due to the similarity of their structures.

As a declarative language, the abstract syntax tree (AST) of a SQL statement acts as a proxy for the intent of the query author. Intuitively, our approach is based on the assumption that overlap between the ASTs of two queries implies overlapping intents. Thus, naively, we would group a query Q with other queries that have nearly (or completely) the same AST as Q. For the remainder of this paper, we will use queries Q to denote both the query itself and its AST encoding. This structural definition of intent has seen substantial use already, particularly in the translation of natural language queries into SQL [6].

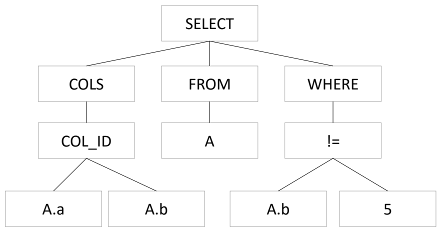

Figure 2: An abstract syntax tree of query `SELECT A.a, A.b FROM A WHERE A.b != 5`.

Queries are grouped by intent using a simple clustering process: (1) A query is first encoded through its AST and then summarized as a vector of *features* that identify meaningful subgraphs of the AST; (2) A distance metric defined over the feature vectors creates a measure of similarity between queries and enables standard vector-based clustering algorithms to organize queries into *intent groups*; (3) A simple user interface uses ground truth from human analysts to validate and refine the intent groups.

3.1 Weisfeiler-Lehman

An ideal distance metric would measure the level of similarity or overlap between ASTs and their substructures. Naively, for two SQL queries Q_1 and Q_2, one satisfactory metric might be to count the number of connected subgraphs of Q_1 that are isomorphic to a subgraph of Q_2. Subgraph isomorphism is NP-complete, but a computationally tractable simplification of this metric can be found in the Weisfeiler-Lehman (WL) Algorithm [11]. Instead of comparing all possible subgraphs of Q_1 against all possible subgraphs of Q_2, the WL algorithm restricts itself to specific subgraphs.

Given a query Q, let $N \in Q$ denote a node in Q. N is labeled with the SQL grammar symbol that N represents. The *i-descendant* tree of N: $desc(N, i)$ is the sub-tree rooted at N, including all descendants of N in Q up to and including a depth of i.

Example 1. *Given the tree in Figure 2, $desc($`COLS`$, 2)$ is the tree containing the nodes* `COLS`*,* `COL_ID`*,* `A.a`*, and* `A.b`*.*

The WL algorithm identifies a query Q by all possible i-descendant trees that can be generated from Q:

$$id(Q) = \{\ desc(N, i) \mid N \in Q \wedge i \in [0, depth(Q)]\ \}$$

Here $depth(Q)$ is the maximum distance from the root of Q to a leaf. As an optimization, subtrees are deterministically assigned a unique integer identifier, and the query is described by the bag of Q's i-descendant tree identifiers. Thus two query trees with an isomorphic subtree will both include the same identifier in their description. The bag of identifiers is encoded as a (sparse) feature vector and allows Euclidean distance to measure the similarity (or rather dis-similarity) of two queries.

The WL algorithm assumes zero knowledge about structural features of the trees it compares, limiting itself to i-dependent subtrees. Conversely, the grammar of SQL has a very well defined structure. In Ettu, we exploit this structure to eliminate redundancy and create features that more reliably encode the query's semantics. Concretely, we consider three improvements over WL. First, the number of features created by the WL algorithm is large. Although clustering naturally prunes out features without discriminative power, we can use SQL's semantics to identify structures that are unlikely to be useful. For example, the subtree $desc($`SELECT`$, 1)$ in Figure 2 is common to virtually all queries. Such features can be pruned preemptively.

Second, the precise structure of a feature may not be relevant to the intent of the query. Queries that are procedurally generated often include placeholders or components that are dynamically constructed; At best, such components serve to modify a simpler, more general query intent.

Example 2. *Consider the query* `SELECT * FROM R WHERE R.a = 5`*. Although the subtree* `R.a = 1` *identifies the specific goal of the query, the same goal could also be abstracted as* `R.a = ?` *where* `?` *denotes a placeholder constant.*

Finally, SQL makes frequent use of commutative and associative operators. The semantics of such operators may overlap, even if their i-descendant subtrees do not.

Example 3. *Consider the Boolean expressions* `A AND B` *and* `A AND B AND C`*. The former AST is a* `AND` *node with 2 children while the latter has 3. Although the two ASTs*

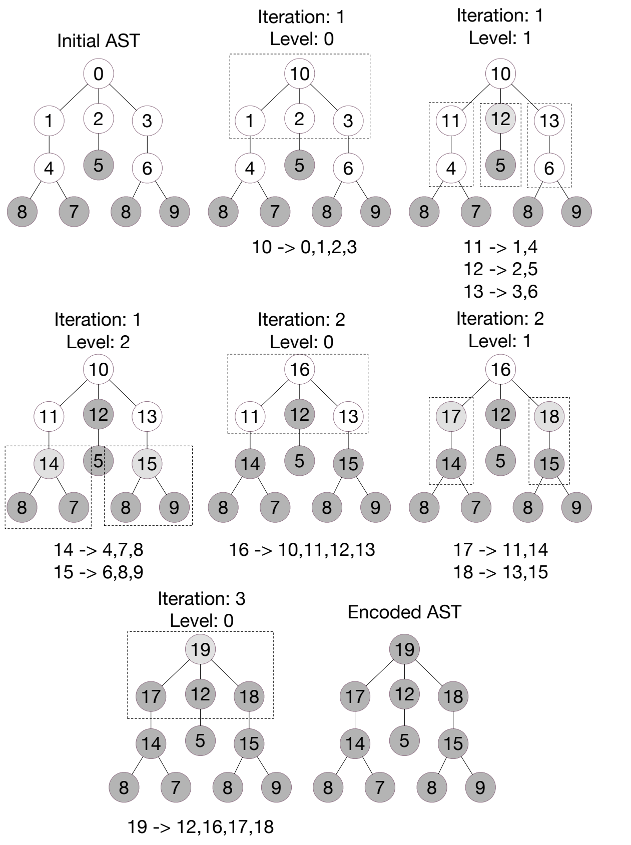

Figure 3: Weisfeiler-Lehman algorithm applied on AST given in Figure 2.

have 3 0-descendant subtrees in common, they share no 1-descendant subtrees; WL ignores the similarity between the two conjunctive expressions.

We address these challenges below in Sections 3.2 and 3.3.

3.2 Query Skeletons

Given a set of queries, instead of considering all of them, we only consider the differences in their structures assuming that differences in constant values have a minor effect on the intent of queries. A query with its constant values replaced by a placeholder grammar term is called a *query skeleton*. Hence, the two queries "`SELECT A.a FROM A WHERE A.b = 5`" and "`SELECT A.a FROM A WHERE A.b = 2`" share the same skeleton because they have exactly the same query structure except for constant values.

Analysis using the set of query skeletons instead of original SQL queries will reduce the number of distinct queries processed and in general will produce similar clusters. However, there are some cases where the constants do play a role. For example, consider the earlier example of a bank teller in Buffalo withdrawing money for a customer from California. To handle these cases, we can pass the constants themselves as additional features for the clustering algorithm.

3.3 Dynamic Features

Second, we generalize the WL algorithm to enable more flexible, structure-aware query tree labeling. Suppose we are given a query Q. Each node $N \in Q$ has a set of *labels* $\texttt{labels}(N)$, initialized to the singleton set containing the SQL grammar atom of N. A *labeling rule* is applied to the labels of a node N and those of its children and generates new labels to be added to $\texttt{labels}(N)$. Given a set of rules, we apply them bottom-up to the nodes of Q to compute a full set of labels for Q's nodes. Finally, the feature vector of Q is defined by the bag $\biguplus_{N \in Q} \texttt{labels}(N)$ of all labels on all nodes of Q.

Example 4. *The AST for the Boolean formula* `A=a OR B=b`*, would begin with the feature set* $\{|A, =, a, OR, B, =, b|\}$*. We could define a rule to be applied to = nodes that constructs a skeleton label: Applied to the subtree* `B=b`*, this rule would add a new label denoting the tree* `B=?`*, where* `?` *is a placeholder value.*

3.4 Clustering

Finally, vector-based clustering is performed on the feature vectors obtained from query skeletons. These clusters are manually classified into three categories by the user: (1) Safe Clusters that correspond to normal activities, (2) Unsafe Clusters that could potentially be insider attacks, (3) Unknown Clusters that represent too broad a group of queries to classify as safe or unsafe. Clusters of this third type are subdivided further until a set of clusters is obtained that are purely safe or unsafe.

To help administrators to reliably label intent clusters, we need a way to compactly present the set of user queries in each cluster. We adopt Frequent Pattern Trees (FP Trees) [3] for this purpose. Normally, FP Trees help to mine frequent patterns of item-sets, for example bags of items commonly bought together. Individual items in each item-bag are sorted by their global *frequency*, or the total number of times the item occurs in any bag. An FP Tree is a prefix-trie built over these sorted item-bags. We build an FP Tree over the queries, using query features as items and query trees as item-bags. Recall that each feature corresponds to a subtree of the AST and there is a unique shared feature behind every node in the FP Tree. To describe a cluster, Ettu presents the user with an expandable view of the FP Tree. As the user selectively traverses the tree, Ettu merges the feature ASTs of nodes from the root to each expanded node to create a syntactically correct partial AST for visualization. FP trees provide three beneficial features for human inspection: (1) A high compression rate (ratio of total number of items consumed v.s. number of nodes generated in the tree), compactly summarizing millions of incoming queries per day; (2) A customizable visualization level, allowing users to settle on an appropriate level of detail without being overwhelmed.

If the items are sorted in the descending order of *frequency*, there are better chances that more prefix strings can be shared. This assumption is validated by experimental results in [3]. But this assumption fails when an item occurs very frequently but with no others. Thus, as an optimization we replace *frequency* by *total popularity.* The *popularity* of a feature in its bag is the number of distinct features that coexist with it. *Total popularity* is the sum of single bag *popularities.*

We compare the compression rates for FP Trees generated using both frequency and total popularity in Figure 4, which shows how compression rate varies with the number of queries incorporated into the tree. As the number of queries increases, the compression rate for both approaches increases super-linearly, suggesting that FP Trees are a good way to visualize large query clusters.

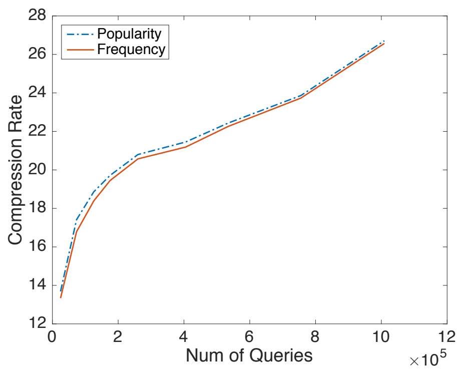

Figure 4: Popularity vs Frequency when determining the compression rate.

4. FEASIBILITY

Our evaluation is based on anonymized SQL query logs that capture all query activity on the central databases of a major US bank over a period of approximately 19 hours. Although there are 1.35 million parsable SELECT queries in our dataset, there are only 1614 different query skeletons; Most queries are differ only in constants. Even this simple optimization significantly reduces the load on Ettu's users.

In order to demonstrate the feasibility of our proposed system, we run Ettu with a sample set of 140 different SQL query skeletons, allowing us to more reliably visualize the clustering process. We perform hierarchical clustering with this query skeleton set and compare the clustering result with our manual clustering. Figure 5 shows correspondences between 2 trees: the dendrogram of hierarchical clustering on the left hand side and manual grouping on the right hand side. The high correlation between two trees shows that Ettu is actually able to group SQL queries based on their intents and it is possible to flag queries that deviate from normal user behavior as abnormal so that database administrator can easily inspect and prevent possibly harmful activities.

5. FUTURE WORK

This paper represents the first steps for Ettu. We plan several extensions as future work. First, the dataset we used to perform our experiments included only 19 hours of data. We will explore ways to make the clustering process scalable, allowing Ettu to validate much larger query logs. Second, we will explore new feature weighting strategies and new labeling rules that better capture the intent behind logged queries. Third, we will explore user interfaces that better present clusters of queries — Different weighting strategies for sorting features in an FP Tree, for example. Lastly, we will investigate the effect of temporality on query clustering.

6. ACKNOWLEDGMENTS

This material is based in part upon work supported by the National Science Foundation under award number CNS - 1409551. Usual disclaimers apply. We also would like to thank Manish Gupta and Rick Jesse for their efforts in preparation of the dataset we used in our experiments.

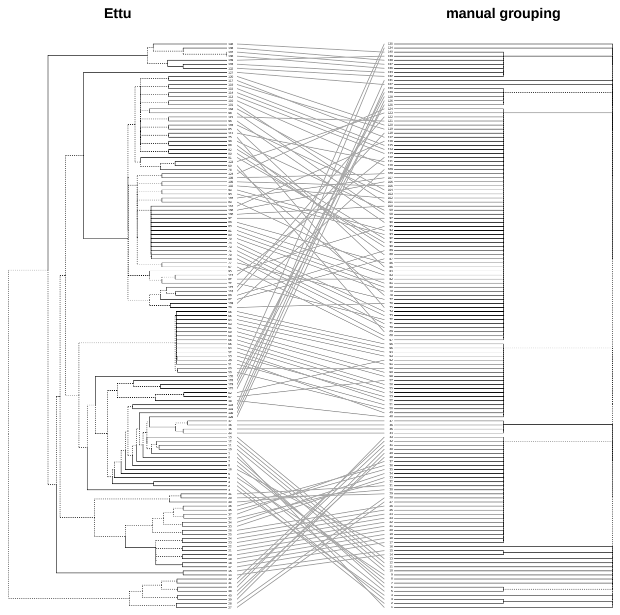

Figure 5: Correspondences between hierarchical clustering obtained from Ettu and from manual grouping.

7. REFERENCES

[1] CERT Insider Threat Center. 2014 U.S. State of Cybercrime Survey. July 2014.

[2] G. Gavai, K. Sricharan, D. Gunning, J. Hanley, M. Singhal, and R. Rolleston. Supervised and unsupervised methods to detect insider threat from enterprise social and online activity data. *JOWUA*, 2015.

[3] J. Han, J. Pei, Y. Yin, and R. Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *DMKD*, 2004.

[4] A. Kamra, E. Terzi, and E. Bertino. Detecting anomalous access patterns in relational databases. *VLDBJ*, 2007.

[5] G. Kul and S. Upadhyaya. Towards a cyber ontology for insider threats in the financial sector. *JOWUA*, 2015.

[6] F. Li and H. V. Jagadish. Constructing an interactive natural language interface for relational databases. *pVLDB*, 2014.

[7] S. Mathew, M. Petropoulos, H. Q. Ngo, and S. Upadhyaya. A data-centric approach to insider attack detection in database systems. In *RAID*, 2010.

[8] R. Maxion and T. N. Townsend. Masquerade detection using truncated command lines. In *DSN*, 2002.

[9] A. S. McGough, B. Arief, C. Gamble, D. Wall, J. Brennan, J. Fitzgerald, A. van Moorsel, S. Alwis, G. Theodoropoulos, and E. Ruck-Keene. Ben-ware: Identifying anomalous human behaviour in heterogeneous systems using beneficial intelligent software. *JOWUA*, 2015.

[10] Ponemon Institute. 2015 Cost of Cyber Crime Study: Global. October 2015.

[11] N. Shervashidze, P. Schweitzer, E. J. V. Leeuwen, K. Mehlhorn, and K. M. Borgwardt. Weisfeiler-lehman graph kernels. *JMLR*, 2011.