Automated Provenance Analytics: A Regular Grammar Based Approach with Applications in Security *

Mark Lemay
Boston University
lemay@bu.edu

Wajih Ul Hassan
University of Illinois at Urbana-Champaign
whassan3@illinois.edu

Thomas Moyer
Nabil Schear Warren Smith
MIT Lincoln Laboratory
{tmoyer,nabil,warren.smith}@ll.mit.edu

Abstract

Provenance collection techniques have been carefully studied in the literature, and there are now several systems to automatically capture provenance data. However, the analysis of provenance data is often left "as an exercise for the reader". The provenance community needs tools that allow users to quickly sort through large volumes of provenance data and identify records that require further investigation. By detecting anomalies in provenance data that deviate from established patterns, we hope to actively thwart security threats. In this paper, we discuss issues with current graph analysis techniques as applied to data provenance, particularly Frequent Subgraph Mining (FSM). Then we introduce **D**irected **A**cyclic **G**raph **r**egular grammars (DAGr) as a model for provenance data and show how they can detect anomalies. These DAGr provide an expressive characterization of DAGs, and by using regular grammars as a formalism, we can apply results from formal language theory to learn the difference between "good" and "bad" provenance. We propose a restricted subclass of DAGr called **d**eterministic **D**irected **A**cyclic **G**raph **a**utomata (dDAGa) that guarantees parsing in linear time. Finally, we propose a learning algorithm for dDAGa, inspired by Minimum Description Length for Grammar Induction [1].

1. Introduction

Conventional security wisdom assumes that the best way to secure a system is to build high, strong walls to keep attackers out of the system entirely. However, attackers continually breach these walls, gaining access to sensitive data and control over sensitive systems. Until these boundary defenses are sufficient, techniques will be needed to detect intrusions.

Data provenance provides a detailed history of the ownership and processing of data. Provenance supports a wide-range of applications, from network troubleshooting [22, 23] and forensic analysis of attacks [18] to intrusion detection [16, 17] and secure auditing [6, 13]. In each case, the developers built analytic frameworks tailored for the system in question and targeting specific types of attacks. Unfortunately these "hand-crafted" analytics techniques do not scale [12], and we need a general solution that can be applied to any system. Fortunately, data provenance are commonly represented using Directed Acyclic Graphs (DAG)s [20]. By viewing provenance as DAGs we can simplify provenance analysis to DAG analysis.

For any significant attack, the provenance DAG should be different from the provenance DAGs of normal execution. If we can differentiate normal graphs from bad graphs, we could sort through the noise of benign behavior and identify anomalies that require troubleshooting and remediation. This strategy would make the attack provenance analytics easier, general, and scalable.

In this paper, we propose a general purpose DAG analysis that extends regular grammars to DAGs. Like regular expressions, we expect these to be a general purpose tool to analyse DAGs in general and provenance DAGs specifically. We will focus our examples and analysis on a security scenario that aims to identify when a given graph differs from known good graphs, with the end goal of identifying those provenance graphs that require further analysis. First we explore existing graph analysis techniques, particularly Frequent Subgraph Mining (FSM), to analyze provenance graphs. There are several deficiencies with FSM in the context of provenance DAGs that we will illustrate. Then

* DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

This material is based, in part, upon work supported by the Assistant Secretary of Defense for Research and Engineering under Air Force Contract No. FA8721-05-C-0002 and/or FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Assistant Secretary of Defense for Research and Engineering.

TaPP 2017, June 22-23, 2017, Seattle, Washington.

we will present 2 new DAG modeling frameworks inspired by existing work in *graph grammars*, (string) regular grammars, and finite state automata. These frameworks do not suffer from the problems of FSM.

We are not the first to explore general purpose analytics for data provenance. The PLUS system proposes the use of statistical properties of graphs as a rough filtering mechanism [4]. Several query languages have been proposed including OPQL [19] and Datalog [11]. Our system is unique in that it is both automatic and can learn causal relationships between nodes.

The main contributions of this paper are as follows:

- The definition of DAGr that provides a simple model of DAGs analogous to regular string grammars.
- A proof that characterizing graphs with DAGr is NP-hard in general.
- Defining a strict subset of DAGr called dDAGa, where characterizing graphs takes linear time in the size of the graph.
- A framework for learning dDAGa from examples.

The rest of this paper is structured as follows: we first review the background on different graph learning techniques in § 2. In § 4, we give motivating examples for our grammatical approach. In subsection 6.4 and § 7 we discuss our approach for general purpose analytics. In § 8 we give some rough preliminary results. Finally, we propose future directions for our work in § 9 before concluding in § 10.

2. Background

2.1 Frequent Subgraph Mining

There has been a large amount of research into general graph analytics [2, 3, 8]. Since provenance data is represented as a DAG, we had hoped existing graph analytics techniques would help us develop the analytics framework we wanted. Unfortunately little of it seems applicable to data provenance and DAG analysis more generally.

The most relevant existing techniques for graph analysis are in the field of Frequent Subgraph Mining (FSM). FSM is generally hard (the problem of subgraph isomorphism alone is NP-complete). However, there are many algorithms that mitigate this difficulty with different assumptions and heuristics [14]. Of particular interest is Chen et. al.'s algorithm that efficiently finds frequent rooted DAGs [7]. Fundamentally, FSM algorithms, target noisy graphs, and recognize a small amount of structure within them. We argue that highly structured DAGs (such as provenance graphs) are better modeled with simpler more precise models.

2.2 Grammars and Automata

Grammars and Automata provide a guide for how interesting and precise properties of graphs can be modeled. Classes of graph derived this way will be called graph grammars. By reframing the question of graph characterization to one of graph grammar parsing, we can leverage the extensive literature on grammars and automata. Since graph grammars should degenerate into (string) grammars, classic string algorithms provide best case scenarios, and an intractable string problem will certainly be intractable when extended to graphs. Equivalences to automata and other mathematical structures are well studied. The learning problem for (string) grammars is also well studied [9, 10, 21], and there are well known limits on what can be learned.

Regular (string) grammars are the simplest languages in the Chomsky Hierarchy and are perhaps the most studied grammar in this heavily studied field. Regular grammars are famously equivalent to Nondeterministic Finite State Automata (NFA) which are in turn equivalent to Deterministic Finite State Automata (DFA). Because of this equivalence Regular grammars can be parsed in linear time. There exists a standard framework for different DFA learning tasks: algorithms share a standard preprocessing step, state merging procedure, and search strategies. For the sake of simplicity and tractability, we limit the discussion in this paper to regular grammars.

For our security analysis we assume there are many examples of "good" provenance data, for the target system. Hopefully we will have few, if any examples of attack provenance data. This corresponds to the case of grammar learning with only positive examples. The textbook algorithm for this case is Adriaans and Jacobs's Minimum Description Length DFA learning algorithm [1]. Their algorithm makes use of the standard grammar learning framework, which we will generalize to DAGs. Researchers that hope to characterize DAGs under different assumptions can use our generalizations and apply them to the appropriate algorithms in the DFA literature.

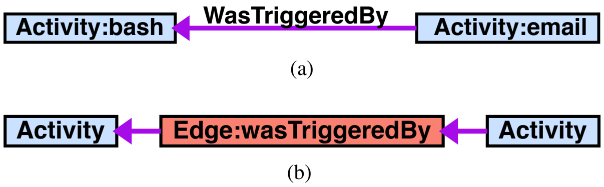
2.3 Graph Grammars and Automata

We are not the first to advocate for the use of grammars and automata to characterize graphs. Regular tree grammars are well studied and have very nice properties. Like regular (string) grammars there exist variants of DFA's that are equivalently expressive to regular tree grammars. This means parsing can be performed linearly in the size of the tree. With a little ingenuity DFA learning techniques can be applied to these tree DFAs. Babic‘ et. al. have even applied these in a system security context [5]. Babic‘ et. al.'s work differs from ours in precision and scope, they attempted to learn malicious tree structures from positive and negative examples in general. We attempt to characterize specific programs from their provenance DAGs and report any structural deviation.

Another strain of research is centered around the Subdue system [15]. Subdue leverages Graph Grammars to find FSM in general graphs. Subdue also uses the MDL principle as it's learning metric. However, Subdue frames the problem with Context-Sensitive Graph Grammars and interprets the results over possibly cyclic or undirected graphs, the algorithm can only be tractable by relying on heuristics. Subdue like other FSM algorithms, has issues analyzing highly structured data such as provenance DAGs.

3. Simplifying Provenance DAGs

Though the OPM [20] defines DAGs with information encoded on the edges, we will apply a simplifying transformation that encodes edge labels as nodes. Before transformation W3C OPM compliant graph is shown in the Figure 1a and after transformation show in Figure 1b.



(a)

(b)

Figure 1. **(a)** Before transformation **(b)** After transformation

Additionally, most specific information is removed and the remaining more general information becomes the node label. This reduces provenance data to the much more standard form of a labeled directed graph (specifically a DAG).

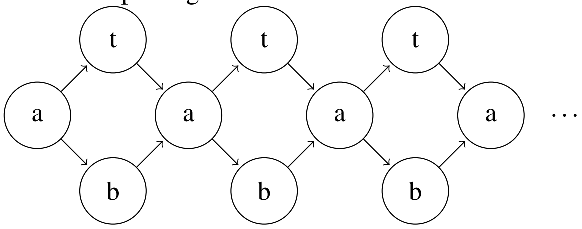
This paper will address the algorithms and example graphs in this more general form. We also stress that the insights from this paper are applicable to any situation where causal DAGs occur, such as the other W3C PROV [1] provenance model.

4. Motivating Examples

In this section, we present two DAG patterns to motivate our work. Note that the observations we make on DAGs are applicable to provenance graphs as they are restricted DAGs.

4.1 Diamond Dag

We present the pattern of diamond DAGs as a simple representative of repeating structures:

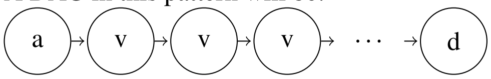


A DAG is in this pattern if it is composed of only of repeating diamonds. This pattern could correspond to the provenance of a system that has some data (m) and runs two processes (t, b) whose outputs overwrite the old data, this may happen any number of times.

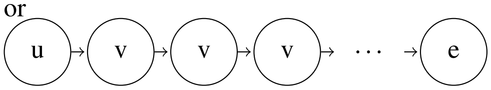
4.2 Only Admins Delete

If we have a system where there are users (u) and administrators (a), who both will view files (v) for some period of time. Finally administrators will delete some file (d), but users can only exit (e).

A DAG in this pattern will be:



or

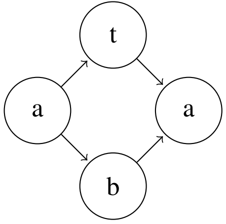


5. Frequent Subgraph Mining (FSM)

One key property of provenance graphs is that there are repeated substructures in the graph. For example, in the provenance graph of an Apache Webserver the provenance subgraphs of each user request will be repeated. There are many repeated patterns in our motivating examples for this reason. FSM uses repeated patterns to determine how similar a graph is to a set of training graphs by comparing their repeated subgraphs.

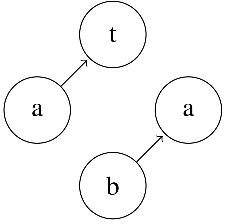
5.1 Example: Diamond Dag

If we try to discover the repeating diamond DAG pattern from examples we will eventually learn the repeated subgraph:

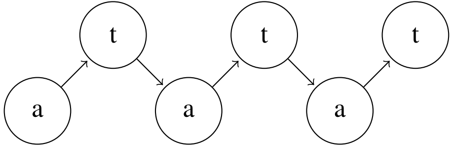


Which is the subgraph that defined the pattern.

However, depending on the specific FSM algorithm used and its parameters, we may also learn these repeated subgraphs

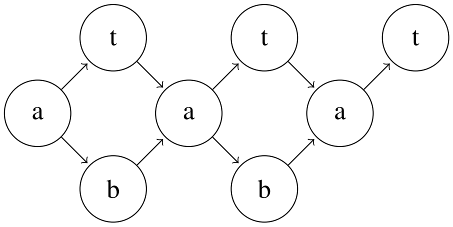


or



If a FSM system is asked if the following DAG is a diamond DAG

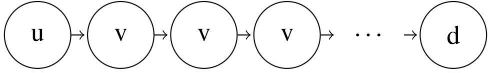
[1] https://www.w3.org/TR/prov-primer/



The FSM system will only measure a small deviation since this new graph shares all the common subgraphs of the diamond DAG pattern and the extension is also a frequent subgraph of the diamond DAG pattern.

5.2 Example: Only Admins Delete?

Worse, consider the "Only Admins Delete" pattern. If an attacker, logged in as a user gains privileges and deletes a file they leave the following data provenance:



FSM will consider the graph to be fine since every proper subgraph is represented in the training set.

Worse still, many FSM algorithms calculate a score to determine how similar a questionable graph is to the training graphs. If this is the case, an attacker can simply boost their score by performing many mundane actions that will eventually outweigh any unusual behavior.

5.3 What is FSM's role for provenance?

These examples show that we want more than FSM can offer for data provenance. We want to characterize patterns and causality, not just superficial common substructure. FSM is a tool designed for finding the most structured features of noisy data, but a poor tool for characterizing highly structured DAG patterns.

There are potential uses for FSM in data provenance, we expect reasonable results when the data is noisy. If there is no recursion in the DAG data, FSM should perform acceptably. FSM may be sufficient for tasks that do not depend on the causality of nodes. We do not recommend FSM for security analysis. It can also serve as a benchmark that dedicated systems for provenance should easily surpass[2].

6. DAGr

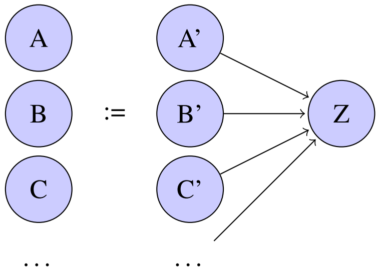
Our first contribution is a definition of a **D**irected **A**cyclic **G**raph **r**egular grammars.

6.1 DAGr Definition

Our definition is intended to be similar to the definition of right regular (string) grammars.

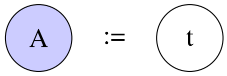
DAGr is defined as the tuple (N, Σ, P) where:

- N is a nonempty, finite set of non-terminal nodes.
- Σ is a finite set of node labels.
- P is a set of production rules, each of one having the form



where $A, B, C, ..., A', B', C', ..., Z \in N$. This means that A can be transformed into A', B can be transformed into B', and so on with an outgoing edge connected to a new rewrite node.

Or



where $A \in N$ and $t \in \Sigma$.

Unlike right regular (string) grammars, we don't need a starting set because the 0-arity case of P rules makes it redundant.

We will restrict our definition with one more condition: our definition must be symmetric, for every language defined the inverse must also be definable by a DAGr. Without this restriction very complicated relationships may exist between outgoing edges. This also gives our language symmetry which reflects the symmetry of string grammars (right regular string grammars are equivalent to left regular string grammars). One important and motivating consequence of this restriction is that all nodes have bounded indegree and outdegree.

A language defined by a DAGr contains a DAG if the DAG can be produced by the grammar's production rules. If the indegree and outdegree of nodes is limited to 0 or 1, then this formulation is equivalent to regular string grammars (with strings encoded as DAGs). If the indegree is limited to 0 and 1, then this formulation is equivalent to regular regular tree grammars.

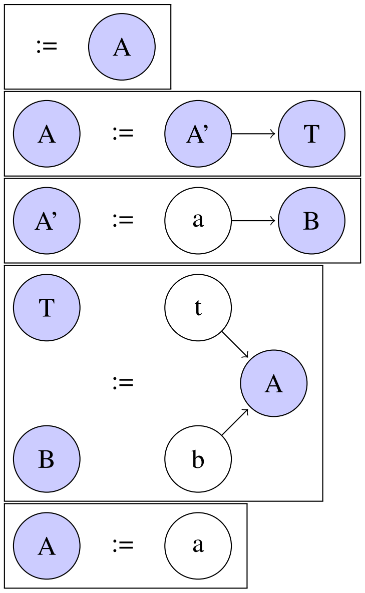
We considered several definitions for DAGr, including transition rules with unbounded inputs, and further work could be done in that direction. We chose the most restrictive definition because even parsing DAGr is NP-hard.

In the graph grammars we define below, we will not define variables that we believe are clear from context. When there is no choice of the transition rules, we will compose them. All examples are symmetric according to our restriction, but we will be merciful and withhold the often redundant definition of the reverse graph.

6.2 Example: Diamond DAG

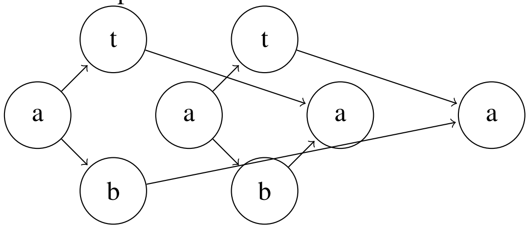
Consider the diamond DAG from before. We can define a reasonable DAGr grammar with the rules:

[2] However the number of FSM algorithms, heuristics and assumptions could make it a misleadingly easy benchmark.



This demonstrates the flexibility of such an atomic graph grammar description. Subgraphs can be built out of these grammar productions and do not need to be explicitly encapsulated (like in Subdue).

However, by restricting the class to regular grammars we do lose some precision.



is also parsable from the above grammar. Like their string counterparts DAGrs have a bounded memory on each node, and some structures can only be partially captured. A context free grammar specification would solve some of these problems. For patterns like the diamond DAGs the problem can be mitigated if the number roots or leaves are fixed, which we assume to be the case in the context of data provenance.

6.3 Example: Only Admins Delete!

The indegree and outdegree of every node in the “Only Admins Delete” pattern is 0 or 1 so this pattern perfectly reduces to the equivalent regular expression.

```
    uv*e|av*d
```

6.4 DAGr Parsing

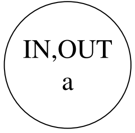
Our second contribution is to prove DAGr parsing is NP-hard (which we present in the Appendix). This should set our expectations for what can be achieved with DAG grammar parsing in general, since our intentionally restricted definition of DAGr will be a subcase of many reasonable graph grammars. As such, we do not recommend using DAGr (or more powerful grammars) for real time security sensitive tasks, though they and DAGr could be useful in other situations. The hardness of parsing presumably increases the difficulty of grammar learning tasks, which is why we will not present a learning algorithm for DAGr.

7. dDAGa

The proof of NP-hardness gives us insight into how we can restrict DAGr for efficient parsing. The difficulty comes from “lateral dependencies.” A node that is parsed with one production rule can have drastic effects on how another node must be parsed, even when the nodes are not descendants of each other. However, we want to characterize causal phenomenon: every node should be judged valid or invalid by only its inputs and outputs. Analogously to how a state assigned to a symbol from a DFA is completely determined by the path taken to reach it, we want a state assigned to be determined only by its input and output. By restricting DAGr, we can have something like regular expressions for DAGs, which can be used as causal models for provenance DAGs.

We can achieve this efficiently by assigning two sets of symbols: input symbols, $Q_{\rightarrow}$, and output symbols, $Q_{\leftarrow}$. Input symbols can only be produced by looking at other input symbols that form a node’s input and the label of that node. Likewise output symbols can only be produced by looking at other output symbols that form a node’s output and the label of that node.

We will use the following convention when writing nodes that have been marked with input symbols, output symbols, and node labels:



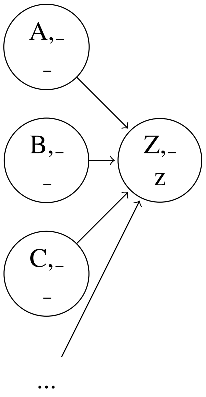
Where IN $\in Q_{\rightarrow}$, OUT $\in Q_{\leftarrow}$ and $a \in \Sigma$

7.1 dDAGa Definition

With this in mind, we propose a definition for dDAGa extending standard notation for DFAs.

dDAGa is defined as the tuple $(Q_{\rightarrow}, Q_{\leftarrow}, \Sigma, M, \delta_{\rightarrow}, \delta_{\leftarrow})$ where:

- $Q_{\rightarrow}$ and $Q_{\leftarrow}$ are nonempty, finite and disjoint sets of input and output symbols.
- $M : (Q_{\rightarrow} \times Q_{\leftarrow})$ are acceptable pairings of input and output symbols.
- Σ is a finite set of node labels.
- $\delta_{\rightarrow}$ and $\delta_{\leftarrow}$ are input and output transition functions. $\delta_{\rightarrow}$ is defined where $A, B, C, ..., Z \in Q_{\rightarrow}$, and $z \in \Sigma$. $\delta_{\rightarrow}(z, (A, B, C, ...)) = Z$, this will reflect in our graph as:



Where _ is a wildcard that highlights what information is ignored by $\delta_{\rightarrow}$. $\delta_{\leftarrow}$ is defined symmetrically.

This formulation is equivalent to that of having an input and output deterministic tree automata with a map making explicit where the trees can join with what symbols.

Like in the DAGr case, dDAGa is expressive enough to model regular string grammars and regular tree grammars.

7.2 dDAGa Parsing

A dDAGa can be parsed in 4 linear passes (in proportion to the number of edges + nodes). The following pseudo-code makes this explicit

7.3 dDAGa Learning

Due to the deterministic nature of dDAGa, we can extend the standard DFA learning infrastructure to DAGs. DFA learning algorithms are generally constructed out of the following three functions:

- The bf preprocessing function create a maximal prefix DFA that encodes all observations and does not recurse.
- The bf state merging function takes an automata and makes a new automata such that 2 states indistinguishable and is consistent with the previous automata.
- Finally a bf search strategy a searches through the space of automata created from merges. There are several different strategies with the most popular being “evidence driven” or greedy.

These components are explained further in subsequent subsections.

7.3.1 Preprocessing Function

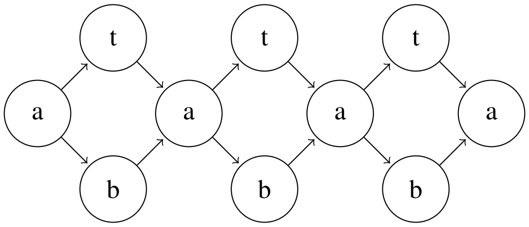
In DFA learning a maximal prefix tree automata is created from all data as a preprocessing step. In our extended function we create maximal prefix and suffix dDAGa.

This process is similar to the parsing code except that a fresh input and output symbol is created when new transitions are discovered.

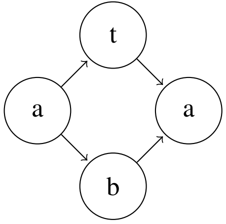
For example, if we are given

```
def parse(a: dDAGa, dag: DAG):
  # A map from nodes to their input symbols
  input_map = dict()
  # A map from nodes to their output symbols
  output_map = dict()

  order = dag.topological_sort

  # Move forward through the topological
      sort and assign the fully determined
      input symbol for every node
  for node in order:
    parent_symbols = map(node.parents,
                         input_map)
    in_symbol = a.δ→(node.label,
                     parent_symbols)
    input_map[node] = in_symbol

  # Move backward through the topological
      sort and assign the fully determined
      output symbol for every node
  for node in order.reverse:
    child_symbols = map(node.children,
                        output_map)
    out_symbol = a.δ←(node.label,
                      child_symbols)
    output_map[node] = out_symbol

  # Finally confirm every node has an
      appropriate symbol pair composed from
      its input and output symbols
  for node in order:
    if not a.M(input_map[node],
               output_map[node]):
      return "couldn't parse"

  return "could parse"
```

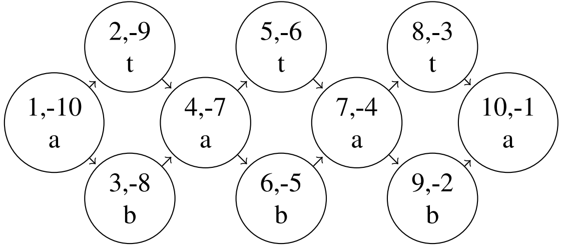
Listing 1. Pseudocode for linear time parsing



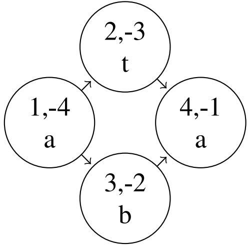
and



as inputs to the preprocessing function. The preprocessing function will create a dDAGa that assigns each input node the states



and

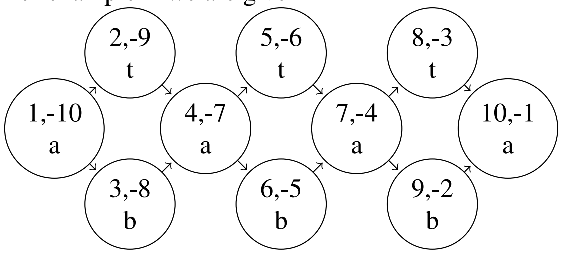


Where input symbols are positive numbers and output symbols are negative numbers. Note that we can only differentiate some nodes by looking at both input and output. If we had 2 indistinguishable nodes they would be implicitly merged.

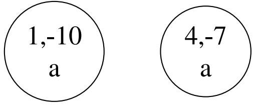
7.3.2 State merging

Most DFA learning algorithms make use of state merging so we will extend it to dDAGa. State merging in DFA learning makes 2 states equivalent, possibly inducing cycles that serve to embed a structure's inherent repetition. In dDAGa we merge 2 states by merging the input tree states and merging the output tree states.

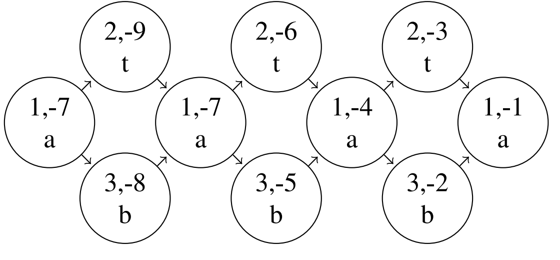
For example if we are given



and we merge

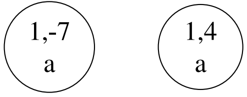


it will result in:

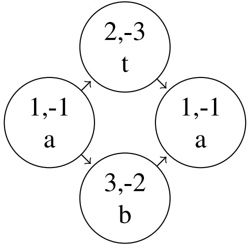


This can be interpreted as any number of diamond repeats followed by 2 additional diamond patterns.

If we merge again



we will get:



Which captures the repeating diamond pattern as best as possible.

7.3.3 Search

The out final learning algorithm pairs every 2 differentiable states, greedily chooses the best available state, and repeats, until there are no more states or the process is terminated. The best dDAGa is then presented.

Using the MDL principle we compute the cost to encode the input and output tree automata. Then we add the cost of generating the DAG as if it were just an input and output tree.

In general the cost function is tuneable, can make use of domain knowledge, and can prevent incompatible states from being incorrectly merged.

8. Preliminary Results

8.1 Provenance Results

We tested an early version of dDAGa learning on simulated application level provenance data. The application modeled a typical internal government web application. Unfortunately the data did not have a recursive structure so the more advanced features of the analysis were not used. Only an early version of the preprocessing step was run, this version of the preprocessing step only dealt with input states, $Q_{\rightarrow}$.

We compared this simplified dDAGa learning algorithm to a FSM analysis. The FSM algorithm was able to discover all the frequent subgraphs, which is a best case scenario. We configured our FSM search to characterize graphs as similar when the computed similarity fell under a given threshold. The threshold was tuned on some experimental data, lowering the threshold would increase the false positive rate, but decrease the true negative rate and vice versa.

We tested against several attack scenarios such as: inserting data into the database without attribution (Unreported Generator), when data that should be added to the database was not (Not Added to Database), internal tasks being repeated that should only happen once (Two Merges), an unexpected overwrite in the database (Overwrite in Database),

	FSM	dDAGa learning[3]
Normal Run	5.2%	0.0%
Restart Database	4.8%	0.0%
Unreported Generator	9.0%	100.0%
Not Added to Database	100.0%	33.3%[4]
Two Merges	2.2%	100.0%
Overwrite in Database	34.7%	33.3%
Generate Twice	0.7%	100.0%

Table 1. Shows how many trails were flagged as dissimilar from the training set by each approach.

and running the entire process twice on the same data when it should only be run once (Generate Twice). We also tested against a benign scenario where the database was restarted (Restart Database).

The first two cases were the regular operations, dDAGa learning had a 0% false positive rate. In total 27,396 sessions were analyzed, though many had the same data provenance. The false positive rate for FSM was the result of an unusual number of repeated access patterns.

The problems with FSM are well highlighted by this data, repeating commons structures makes a "bad" graph appear more normal than a "good" graph (Two Merges and Generate Twice fall well below the false positive rate).

The limited dDAGa learning performs much better on this data. But because our provenance data had no real recursive structure, the more interesting features of the analysis were not tested on this data.

9. Future Work

The most important next step is to test on real provenance data to confirm that we can make this theory fully practical. But there are a number of other interesting directions for future work:

- Improvements to the cost function. Our choice to minimize over the tree automata was somewhat arbitrary, and there may be more principled MDL approaches. The MDL principle is a good hypothesis when nothing is known about data, but there may be many things that are known about provenance that could produce better models.
- Learning from positive and negative examples. In DFA learning this is an "easier" task than learning from only positive examples.
- Applying more sophisticated search strategies instead of our greedy strategy. There are more optimized search procedures for DFA learning, it would be interesting to see if they apply to dDAGa.
- Context free grammars are still considered learnable. It would be interesting to explore their DAG counterpart.
- Solving classic automata problems for dDAGa such as minimization, equality, and subset testing.

10. Conclusion

We consider graph grammars and automata to be the best path forward for automating data provenance analytics in general and for security applications particularly. Current graph analytics techniques are insufficient for data provenance. We have given definitions for DAGr and dDAGa that we hope can help guide future research. Finally, we have explored some of the problems associated with these systems and have characterized their algorithmic properties.

Availability

An implementation of dDAGa is available at `https://github.com/marklemay/GraphAutomata`.

Acknowledgements

This work sprung out of an internship at MIT Lincoln Laboratory. Special thanks goes to Hannah Flynn who gave good advice on the theoretical aspects of this work, and the reviewers who gave excellent comments.

References

[1] Pieter Adriaans and Ceriel Jacobs. Using MDL for grammar induction. *Grammatical Inference: Algorithms and Applications*, pages 293–306, 2006.

[2] Charu C Aggarwal, Haixun Wang, et al. *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*. Springer US, Boston, MA, 2010.

[3] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph based anomaly detection and description: a survey. *Data Mining and Knowledge Discovery*, 29(3):626–688, may 2015.

[4] M David Allen, Adriane Chapman, Len Seligman, and Barbara Blaustein. Provenance for collaboration: Detecting suspicious behaviors and assessing trust in information. In *Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom), 2011 7th International Conference*, pages 342–351, Oct 2011.

[5] Domagoj Babić, Daniel Reynaud, and Dawn Song. Recognizing malicious software behaviors with tree automata inference. *Formal Methods in System Design*, 41(1):107–128, 2012.

[6] Adam Bates, Dave Tian, Kevin Rb Butler, and Thomas Moyer. Trustworthy Whole-System Provenance for the Linux Kernel. *Usenix Security 2015*, page 2015, 2015.

[7] Yen-Liang Chen, Hung-Pin Kao, and Ming-Tat Ko. Mining dag patterns from dag databases. In *International Conference on Web-Age Information Management*, pages 579–588. Springer, 2004.

[8] Diane J Cook and Lawrence B Holder. *Mining Graph Data*. Wiley-Interscience, 2007.

[4] Only a modified preprocessing step was run.

[4] There were 2 crashes in the training data that prevented some data from going to the database. dDAGa learning added these to its model.

[9] Colin De La Higuera. A bibliographical study of grammatical inference. *Pattern recognition*, 38(9):1332–1348, 2005.

[10] Colin De La Higuera. *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[11] Saumen Dey, Sven Köhler, Shawn Bowers, and Bertram Ludäscher. Datalog as a lingua franca for provenance querying and reasoning. In *Workshop on the theory and practice of provenance (TaPP)*, 2012.

[12] Ashish Gehani, Hasanat Kazmi, and Hassaan Irshad. Scaling spade to "big provenance". In *Proceedings of the 8th USENIX Conference on Theory and Practice of Provenance*, TaPP'16, pages 26–33, Berkeley, CA, USA, 2016. USENIX Association.

[13] Ragib Hasan, Radu Sion, and Marianne Winslett. The case of the fake picasso: Preventing history forgery with secure provenance. In *FAST*, pages 1–14. USENIX, 2009.

[14] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28(01):75–105, 2013.

[15] Nikhil S Ketkar, Lawrence B Holder, and Diane J Cook. Subdue: Compression-based frequent pattern discovery in graph data. In *Proceedings of the 1st international workshop on open source data mining: frequent pattern mining implementations*, pages 71–76. ACM, 2005.

[16] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 223–236, New York, NY, USA, 2003. ACM.

[17] Samuel T King, Zhuoqing Morley Mao, Dominic G Lucchetti, and Peter M Chen. Enriching intrusion alerts through multi-host causality. In *NDSS*, 2005.

[18] Kyu Hyung Lee, Xiangyu Zhang, and Dongyan Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*. The Internet Society, 2013.

[19] Chunhyeok Lim, Shiyong Lu, A. Chebotko, and F. Fotouhi. Opql: A first opm-level query language for scientific workflow provenance. In *Services Computing (SCC), 2011 IEEE International Conference*, pages 136–143, July 2011.

[20] Luc Moreau, Juliana Freire, Joe Futrelle, Robert E McGrath, Jim Myers, and Patrick Paulson. The open provenance model: An overview. In *International Provenance and Annotation Workshop*, pages 323–326. Springer, 2008.

[21] Wojciech Wieczorek. *Grammatical Inference: Algorithms, Routines and Applications*, volume 673. Springer, 2016.

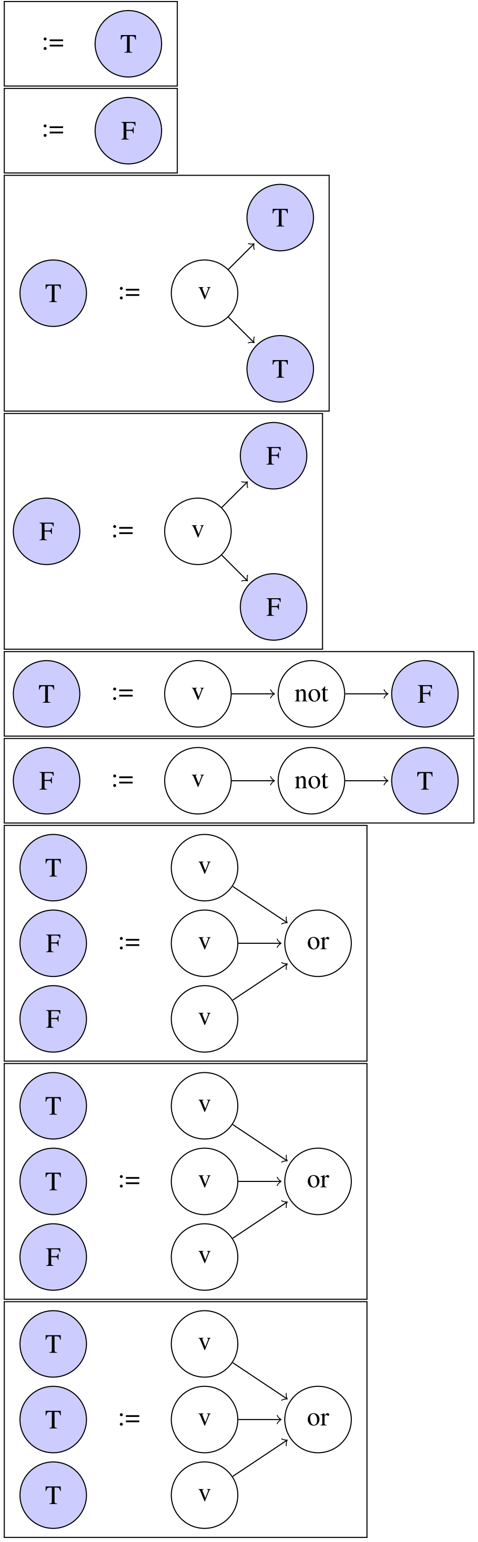
[22] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated network repair with meta provenance. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, HotNets-XIV, pages 26:1–26:7, New York, NY, USA, 2015. ACM.

[23] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

11. Appendix

11.1 proof DAGr Parsing is NP-Hard

Consider a grammar minimally defined by the production rules:



- It is easy to see that this grammar accepts a DAG iff a corresponding instance of 3SAT is satisfiable.
- An instance of 3SAT can be converted to a faithful DAG in linear time. This DAG will be at most twice as large as the 3SAT instance (to accommodate for the branching of "v" nodes).
- If there is a polynomial algorithm for DAGr parsing then it can be used to construct a polynomial algorithm that solves 3SAT (we leave this for future work).

Therefore parsing DAGr in general is NP-hard. If the indegrees and outdegrees are bounded DAGr parsing is NP-complete.