Digging into Big Provenance (WITH SPADE)

ASHISH GEHANI,
RAZA AHMAD,
HASSAAN IRSHAD,
JIANQIAO ZHU,
AND JIGNESH PATEL

A USER INTERFACE FOR QUERYING PROVENANCE

Data provenance describes the origins of a digital artifact. It explains the creation of an object, as well as all the modifications and transformations that transpired over its lifetime. When the historical record is detailed, spans long periods, or both, the information collected can become voluminous. Analysis of provenance is often used even while it is continuously being extended through a series of computations that act upon it. This necessitates a framework that supports performant streaming ingestion of new elements with concurrent querying that yields responses that incorporate data as it becomes available.

Operating systems and blockchains are two of many domains where collection and analysis of *big provenance*[9] has had useful applications. In the case of operating systems, system-call information collected by a kernel's audit framework can form the basis of trustworthy provenance metadata. This facilitates tracking all activity that occurs across a machine or even a federated system. This *whole network provenance*[1] is particularly useful for applications such as malware detection and ensuring the reproducibility of computation.

Bitcoin is a blockchain-based cryptocurrency where individuals can perform transactions with each other. Each transaction between two or more users contains payment information that should be stored in the blockchain. These records form the basis for tracking the provenance of any given digital object in the blockchain. In addition to its primary purpose of tracking currency ownership, the provenance has other applications, such as detecting anomalous behavior in order to identify illegal activity.

Provenance metadata may be stored in a database to facilitate efficient querying. The process of interrogating the system must be intuitive and convenient to use since finding the relevant fragment in big provenance is akin to the proverbial search for a needle in a haystack. In particular, these goals need to be met despite the system's use in a variety of domains, from profiling complex application workflows to performing forensic and impact analyses after attacks on a system have been uncovered.

Several interfaces exist for querying provenance. Many of them are not flexible in allowing users to select a database type of their choice. Some interfaces provide query functionality in a data model that is different from the graph-oriented one that is natural for provenance. Other interfaces have intuitive constructs for finding results but have limited support for efficiently chaining responses, as needed for *faceted search*. This article presents a user interface for querying provenance that addresses these concerns and is agnostic to the underlying database being used.

First, relevant background on data provenance is provided, along with how it is modeled and how

the representation is realized in an open-source implementation. Then the design of the query surface is presented, its core functionality outlined, illustrative use cases described, and salient aspects of the system highlighted.

MODELING COMPUTATIONAL HISTORY

A common way of reporting data provenance is to model it as a graph structure, where vertices represent elements in a historical record, and edges represent events that relate and order the elements. A provenance graph $G(V,E)$ then contains a set of vertices, V, and edges, E. A member, v, of the set V can be an *agent*, *process*, or *artifact* that was involved in an event. Each edge e belonging to set E represents the operation that occurred and relates two vertices, v_i and v_j.

Multiple data models have been developed to represent data provenance. Notable variants are the OPM (Open Provenance Model),[11] published in 2010; the W3C PROV specification,[13] released in 2013; and the DARPA Transparent Computing program's CDM (Common Data Model),[10] finalized in 2019. They have some similarity. Each includes vertices for three categories of elements: agents or principals; processes, activities, or subjects; and artifacts, entities, or objects. They differ in detail based on their intended domain of use: OPM was designed to be domain-agnostic; W3C PROV was created to aid the publication of semantically enriched web content; and CDM is focused on the specific domain of operating systems.

The semantics of the activity domain being monitored

are captured with a custom schema. By using a property graph to represent the provenance, these details can be embedded directly. Vertices and edges are each accompanied by a (possibly empty) set, A, of domain-describing annotations, $A = a_1, a_2,..., a_n$. Each annotation a_i is a key-value pair—that is, $a_i = key_i : value_i$—that reports an aspect of the domain, such as `program:firefox` or `path:/etc/passwd`. In this manner, a provenance graph captures the relative order of events, as well as the salient aspects of the monitored domain.

As an example, consider a vertex representing an operating system process. This vertex could have annotations conveying information such as its name, identifier, or start time. An edge could relate the process to a file that has been read. In the case of bitcoin provenance,[7] a vertex representing a transaction would have annotations such as the hash that identifies it and the earliest time that it is valid. An edge could relate unspent bitcoin to a payee with an annotation specifying the amount.

CANONICAL PROVENANCE QUERIES

The simplest provenance query consists of searching for vertices that match a specification, defined by an expression over the annotations that describe it. Such queries are useful to locate vertices that can serve as the starting point of more complex queries, like the ones described in this article. Akin to retrieving vertices are queries for identifying individual edges. For example, a user may want to learn about all cases where the permissions of a particular file were changed. Since this is an atomic

system event, the user can search for all associated provenance edges. Similarly, once an edge (or a set of them) has been located, the user can extract the endpoint vertices. In the prior example, this would allow the identification of processes that performed the action.

Among the most frequently needed functionality when operating on provenance records is support for finding the lineage of an element. In a lineage query, the ancestry of a data artifact is traced back a specified number of steps. Similar functionality that operates in the other direction is useful for identifying descendants. The ancestors or descendants of a given data artifact are found by recursively locating the parent or child vertices in the graph structure, respectively. The ancestral lineage of an item provides a picture of what transpired leading up to the creation of that item, while the descendant variant describes what was derived from it after its creation. With operating system provenance, the lineage of a file can explain how, when, and by whom that file was created. It can support the enumeration of all the system processes (and their owners) that wrote to or read from a file. In the case of bitcoin, the lineage of a payment can reveal details about the participating users and all the transactions linked to them.

Another important operation consists of searching for paths between a pair of elements. Two variants of this operation often arise in practice. The first involves finding all the paths between two vertices, while the second focuses on finding the shortest path between them. A path between two vertices demonstrates how different data elements influence each other through the events

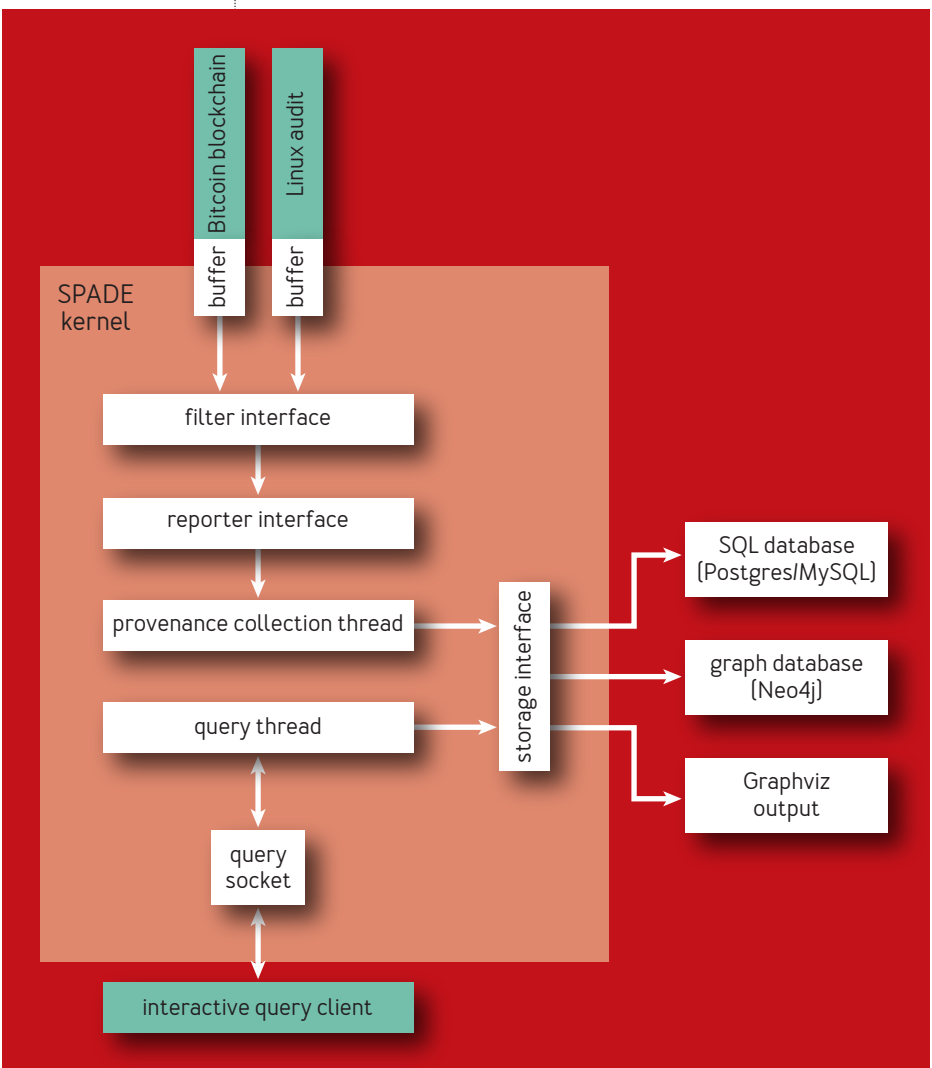
that have transpired in the system. For example, finding a path between a web-browser application and a file downloaded from the Internet shows the complete set of steps required by the browser's user for bringing the file from a remote server to a local machine. In the bitcoin context, paths between two addresses can be used to find the transactions that link those two addresses. This, in turn, can be used to calculate the assets that have flowed from one user to another, even when they go through intermediaries.

PROVENANCE SYSTEM ARCHITECTURE

The open-source SPADE project[3] provides software for inferring, storing, and querying data provenance. It is cross-platform and can be used with diverse sources such as blockchains, online social networks, and multiple operating systems, including Linux, macOS, and Windows. The collection of provenance is done without requiring any change in the applications or the target platform. SPADE is easy to install and configure. It provides a simple mechanism for users to select from a number of storage formats.

The architecture of SPADE is shown in figure 1. It is composed of multiple modules, each playing an independent role in processing provenance records. These modules are managed by the SPADE *Kernel* at the core. Provenance graph elements are inferred about activity domains and sent to the Kernel by *Reporter* modules. After being operated on by any *Filters* present, the elements are sent to *Storage* modules that insert the elements into databases configured with a custom schema. The presence

FIGURE 1: **THE SPADE ARCHITECTURE HAS A KERNEL AT ITS CORE**



of independent threads for ingestion and query processing allows clients to make provenance inquiries while the underlying graph is changing. The various types of modules are described further later.

A Reporter module acts as the producer of provenance metadata. It receives streams of events from diverse sources, extracts relevant information from them, and infers provenance relationships, constructing graph vertices and edges in the process. SPADE provides a multitude of reporters that generate provenance about diverse domains, including operating systems, blockchains, intra-application calls, and user-defined schema.

A Filter module acts on the provenance stream emitted by a reporter. It performs a selection operation on the provenance according to programmed criteria. For example, some filters allow only the vertices and edges that match a specification to pass through. Other filters abstract or remove information in vertices or edges. The output of a filter is the processed provenance information that is meant to be persisted. Several filters can be inserted to operate sequentially on the output of each preceding one.

A Storage module takes the final output of the sequence of filters (if any are present) and stores it in one of the available databases. The module provides an abstraction over the underlying persistent store. This subsystem could be a relational database such as MySQL or Postgres, a graph database such as Neo4j, or any data store. The module provides interfaces for storing and retrieving data that are agnostic to the underlying database.

An analyzer module provides an interface to the user for retrieving provenance records stored in SPADE. It is responsible for receiving a query, sending it to the appropriate storage, processing the information, and sharing the result with the user. The default implementation receives queries from the command line.

DESIGN OF THE QUERY SURFACE

The first generation of SPADE and its precursors introduced support for querying a file's lineage, specified by its path and version (as of a given date and time). It included several optimizations for transferring provenance metadata across hosts[4] and accelerating cryptographic verification of such records.[5] It was not until the second generation[6] that a richer query surface was added, including support for retrieving vertices, edges, paths, and lineage.

In an effort to make querying more usable for navigating big provenance and performing faceted searches, a new surface was developed. Its name, QuickGrail, derives from the fact that its design was inspired by the Grail project[2] and initially implemented atop the Quickstep database.[12] Subsequently, SPADE added support for using QuickGrail with the Neo4j graph and Postgres relational databases.

QuickGrail provides an abstraction over the underlying database so that users can work with a uniform query language, regardless of the data model and serialization below. This allows users to focus on the provenance analysis task at hand without concern for how the queries will be translated into the native language of the database. In addition to being efficient, the surface

provides a uniform mechanism for exporting responses for visualization and other external uses.

The query interface provides a collection of functions to search for provenance records and manipulate the responses retrieved from the underlying database. The databases supported are Neo4j, Postgres, and Quickstep. Each function in the query surface is implemented in an intermediate representation that is translated into the query language of the target database.

The system provides several features that facilitate *faceted search*, allowing the user to home in on information of interest to them. One such feature is the ability to assign a query response to a graph variable. Such variables can be operated upon in subsequent queries to refine the search. Another feature that facilitates efficient user interaction in the presence of big provenance is the ability to limit the size of responses. This allows users to send queries and quickly receive partial responses, which they can inspect to refine their search.

Variables

A special variable, `$base`, represents the entire provenance graph stored in the currently selected database. This variable serves as the universe of provenance for most queries. When a variable is used to store the *response graph* from a query, it will appear on the left-hand side of an assignment (denoted with =), and its name must start with `$`. If a variable is used to define a *query constraint*, its name must start with **%**. Such variables are a convenience, providing a succinct way to represent constraints that may need to be passed repeatedly as arguments to queries.

Constraints

To scope the elements that match a query, a selection *constraint* can be specified. In its most basic form, it consists of an annotation key, a relational operator, and a value. (Recall that annotations were introduced earlier in this article.) The supported operators include `==`, `!=`, `<`, `>`, `<=`, `>=`, and `LIKE`. The last of these facilitates matching a string (with `%` used as a wildcard). For example, `name LIKE '/bin/%'` will match vertices with an annotation-key name that has a value starting with `/bin/`. A constraint of `uid == '0'` can be used to select vertices of processes that ran as `root` (since its `uid` is 0). To simplify reuse, constraints can be stored in variables—for example, `%system_procs = name LIKE '/bin/%'`.

To support more complex filtering, constraint expressions can be built by combining constituents using the logical operators `AND`, `OR`, and `NOT`. This allows the constraint expressions to be framed over multiple annotations. In addition, it allows the user to combine an existing constraint with a new criterion, thereby facilitating faceted search. For example, consider the constraint `%proc_python = name == 'python'` used to find vertices representing python executions. The querier may be interested in the subset that ran as the `root` user. In this case, the querier could define another constraint `%proc_root = uid == '0'`. The two constraints can then be combined into a single expression `%proc_python AND %proc_root` for use in subsequent queries.

Extracting elements

Since provenance graphs consist of vertices and edges, the

most basic functions provided are `getVertex` and `getEdge`. They can be used with any existing graph variable to extract a subset of elements, as specified by a constraint. The output of these functions must be assigned to a graph variable. Note that even though these functions result in sets of vertices or edges, these sets are treated as graphs. In the following example, vertices are extracted from the special variable `$base`, which represents the global graph. They are constrained to the subset with an annotation type of `Process`:

```
%only_processes = type == 'Process'
$all_processes = $base.getVertex(%only_processes)
```

Similarly, every edge is extracted if it has an annotation of operation with value `fork`:

```
%all_forks = operation == 'fork'
$fork_edges = $base.getEdge(%all_forks)
```

Using the set of edges just obtained, the next query extracts the processes that performed the `fork` operations, as well as those that were created as a result:

```
$fork_vertices = $fork_edges.getEdgeEndpoints()
```

Identifying origins and impacts

Given an element in a provenance graph, a central concern is understanding what gave rise to it. Of equal importance is understanding what has been affected by a particular element. In both cases, this is done by starting with the

element, finding its parents or children, respectively, and then recursing. This is supported with the `getLineage` function, which is generalized to operate on a set of seed elements. It takes three arguments: (1) a set of *seed vertices*; (2) the maximum *number of levels* to traverse from the seed vertices, which must be a positive integer; and (3) the *direction* of traversal, which can be `ancestors`, `descendants`, or both. The following example extracts two levels of the ancestral lineage of vertices with a `firefox` annotation:

```
%firefox = name == 'firefox'
$init_vertex = $base.getVertex(%firefox)
$firefox_lineage = $base.getLineage($init_vertex, 2, 'ancestors')
```

Connecting the dots

A preliminary analysis (through a faceted search using increasingly specific constraints, for example) may lead to two sets of vertices being identified: One may consist of the ingress points of network flows into the system, while the second set may have indicators of compromise, such as processes whose privilege was escalated or files whose ownership changed in a particular window of time.

Knowing whether a connection exists between two sets is a key concern. This can be ascertained with the `getPath` function, which takes three arguments: (1) a set of *source vertices*; (2) a set of *destination vertices*; and (3) the maximum *path length* between any source and destination vertex, which must be a positive integer. The semantics of provenance imply that a path will be found only when a destination is in the provenance—that is, it is an ancestor—

of a source. The following example searches for paths with a length of at most three edges between a `firefox` process vertex and the `/etc/passwd` file vertex:

```
%source = name == 'firefox'
$firefox = $base.getVertex(%source)
%destination = path == '/etc/passwd'
$etc_passwd = $base.getVertex(%destination)
$paths = $base.getPath($firefox, $etc_passwd, 3)
```

If the set of paths discovered is large, it can be refined by specifying one or more sets of intermediate vertices. For example, if it is known that a `$compromised_process` set lies on the paths of interest from `$firefox` to `$etc_passwd`, the query can be made more specific:

```
$paths = $base.getPath($firefox,
  $compromised_process, 3, $etc_passwd, 3)
```

Filling in missing pieces

Early in an investigation, specific agents, activities, and artifacts may be known to be of interest, but not all elements (including the relationships between them) may be known. In such cases, the analyst can define a set of interesting vertices and then ask the system to describe how they are related to each other. For example, a set of suspicious network connections and files with modified ownership may be identified in a specified timeframe. The analyst may then wish to know if and how any of these elements are related.

In a generalization, the analyst can include edges in the

set to incorporate information about known provenance relations of interest. This can be effected *in toto* with the `getSubgraph` function, which takes as input a *skeleton graph*. The skeleton is a set of vertices and edges known to be of interest *a priori*. The function returns the provenance subgraph that spans all elements in the skeleton, as well as those that lie on paths between vertices and edge endpoints in the skeleton.

In the example below, the provenance subgraph returned will show a vertex for each thread of the several dozen that Firefox creates, each configuration and cache file that is accessed, and each socket used for interprocess communication, as well as the provenance relations between them.

```
%firefox_threads = name LIKE 'firefox%'
$firefox_skeleton = $base.getVertex(%firefox_threads)
$firefox_process = $base.getSubgraph($firefox_skeleton)
```

Going native

Commodity databases provide diverse query surfaces. The set of primitives supported depends on factors such as the data model employed and the indexing implemented in the underlying engine. Relational databases such as Postgres and Quickstep offer an interface based on SQL (Structured Query Language). Graph databases, such as Neo4j, use Cypher, a graph-oriented declarative analog. Since each database may support custom queries that could be useful to an analyst, a facility is provided to access them. If a query is preceded with the keyword `native`, it

will be passed unmodified to the underlying database. The response will be returned as lines of text rather than as a graph. This allows arbitrary native queries to be invoked.

As an example, consider an operating system provenance graph in OPM, with *Artifact* vertices refined by subtype, including `file`, `link`, `directory`, `block device`, `character device`, `named pipe`, `unnamed pipe`, `unix socket`, and `network socket`. In a preliminary analysis, the distribution of these elements may be of interest to identify unusual patterns. In this case, counts for each subtype can be obtained from Postgres with a user-defined function histogram:

```
native 'SELECT * FROM histogram(vertex, subtype)'
```

COMPLEX PROVENANCE ANALYSIS

When large data sets are analyzed, the process is often iterative. An analyst may construct numerous hypotheses, checking whether each is valid or not by querying the data. As an investigation unfolds, maintaining the workflow's efficiency requires that intermediate results are represented succinctly to avoid I/O bandwidth becoming a bottleneck. In practice, search is often faceted, with the results of one step reused in subsequent ones. It may also involve backtracking and comparing the extracted subsets of data. When results of potential interest are retrieved, visualization or other external processing may allow an analyst to obtain a broad understanding of a selected subset. The query surface has several features that address these concerns. Together, they facilitate agile exploration.

Efficient representation

SPADE models provenance as a property graph. The annotation schema is selected to ensure that hashing them will produce a unique content-based identifier for each vertex and edge. When a query is executed, only the identifiers implicated in the response are associated with the graph variable used to track a response. Effectively, only a skeletal representation that consists of an adjacency list for the corresponding subgraph is constructed. The enriched representation with graph properties in the form of key-value annotations is not immediately materialized. These properties, which describe the domain about which provenance was inferred, use the vast majority of the storage needed to hold the complete graph. Their retrieval is avoided until a response is explicitly exported, either to the console or a file with the `dump` command, respectively. This allows an analyst to make queries that may yield large responses without disrupting their interactive workflow (as would occur if the complete response were to be materialized).

Consider the following sequence:

```
$sources = $base.getVertex(name == 'firefox')
$destinations = $base.getVertex(path == '/etc/passwd')
$paths = $base.getPath($sources, $destinations, 3)
```

The graph variables `$sources`, `$destinations`, and `$paths` track only the identifiers of the implicated vertices and edges. Annotations of elements in the graph `$paths` are retrieved from the database only when `dump $paths` is explicitly issued, for example.

Response reuse

When the query client initiates a session, a local workspace is created to store the graph responses received. Each graph is bound to a variable name, simplifying its repeated use. Such variables can be used in one of two ways. First, since a variable represents a graph, it can be treated as the universe that will be operated upon by subsequent queries. Second, the variable can instead be passed as the argument of a query.

In the following example, the last query uses `$processes` instead of `$base` as the provenance universe in which to search for all process vertices that have a name that starts with `firefox`:

```
%type_process = type == 'Process'
$processes = $base.getVertex(%type_process)
%firefox_threads = name LIKE 'firefox%'
$firefox_parents = $processes.getVertex(%firefox_threads)
```

As a session progresses, the set of currently defined variables can be identified with the command `list graph`. The `stat` command can be used to get statistics about a particular graph. For example, `stat $paths` reports the number of vertices and edges in the graph named `$paths`. The reuse of variable names is supported by destroying a binding with `erase <variable name>`. This eliminates the skeletal representation associated with the variable.

Set manipulation

Initial inspection of the provenance may leave an analyst

with a collection of large subgraphs that require further refinement. For example, knowledge about the activity domain can be leveraged to identify subsets of the graph that are of particular interest, as described earlier. More specifically, queries framed over the domain-specific annotations can lift collections of vertices and edges from the underlying database into the workspace; these seed sets may then be expanded through path and lineage queries.

To facilitate symbolic manipulation of the graphs, a complementary suite of operations is provided. They realize intuitive mathematical set operations in the setting of graphs. In particular, pairs of graphs can be transformed into a union of constituents with the `+` operator. The result contains a vertex set that is the union of vertices in the operand graphs. Similarly, the resulting edge set is the union of edges in the operands. Alternatively, the intersection of two graphs can be calculated with the **&** operator. The resulting graph will contain only vertices and edges that were present in both the operand graphs. Finally, elements in a graph can be removed based on the specification of a second graph. This is effected with the *difference* operator `-`.

Consider a situation where an analyst wishes to determine the set of processes that changed their identity during the course of execution. First, they extract the set of all edges that report a change in identity. Next, they extract the endpoints of these edges, representing the processes that issued the `setuid()` call. The subset initially running as `root` , however, is not of interest in this context. Hence, such processes are removed by subtracting the corresponding set in the last step.

```
$setuid_operations = $base.getEdge(operation == 'setuid')
$chameleons = $setuid_operations.getEdgeDestination()
$privilege_escalated = $chameleons - $chameleons.getVertex(uid != 0)
```

Graph export

Since the graph that results from a query may be large, it is not immediately materialized. Instead, a graph can be used in three ways. First, it can be printed to the console in JSON (JavaScript Object Notation) format. The output is an array of vertices and edges. Each element consists of one or two identifiers—depending on whether it is a vertex or an edge—and the annotations that describe it.

In the next example, an analyst inspects a subset of the contents of a graph. This is done by extracting a sample (10 elements in this instance) using the `limit` function and then printing them with the `dump` command. This motif is instrumental in the course of a faceted search, where an analyst may iteratively refine the queries based on a study of successive intermediate results.

```
$firefox_vertices = $base.getVertex(name LIKE 'firefox%')
$firefox_sample = $firefox_vertices.limit(10)
dump $firefox_sample
```

The second way to use a graph is by exporting it to a file or pipe in JSON format. This allows it to be imported or ingested by an external tool. To effect this, an `export` directive is used to specify the file-system path

immediately before using `dump`. For example, the graph variable `$firefox_vertices` can be serialized to the file `/tmp/firefox.json` with:

```
export > /tmp/firefox.json
dump $firefox_vertices
```

Finally, support is provided for exporting the graph to the widely used Graphviz DOT format. This allows it to be visualized in a number of forms, depending on the layout tool used to render it. The mechanics are similar to the previous method, with an `export` (specifying where the DOT data should be sent) preceding use of the `dump` command:

```
export > /tmp/firefox.dot
dump $firefox_vertices
```

ILLUSTRATIVE USE CASES

This section presents use cases from two domains that were introduced earlier: an operating system and a blockchain. Provenance is queried in a post-event analysis scenario.

Operating systems

Consider a setting where provenance is inferred from system calls, as it is with SPADE's Audit reporter on Linux, OpenBSM on macOS, and ProcMon on Windows. The resulting graph captures the interactions among users, processes, and data artifacts. As a motivating use case, consider the challenge a system administrator is faced

with after a compromise. The nature and extent of the damage inflicted on the target host have to be identified. This can range from determining a malware infection's source to identifying which data has been exfiltrated and which system configurations have been modified.

Now consider an example inspired by attacks seen in practice, as illustrated in figure 2. Understanding the steps of an attack is simplified by analyzing the abstracted provenance relations between processes and artifacts in the system. Assume that an application (`firefox`) accepts a malicious request via a remote connection. This exploits an existing vulnerability in the program. It causes the executing process to be hijacked, with the adversary gaining control of it. Data is written to the location of a binary (`tcexec`). The permissions of the modified file are updated to ensure it is executable. Subsequently, when this binary runs, it accesses system files and exfiltrates them to a remote host.

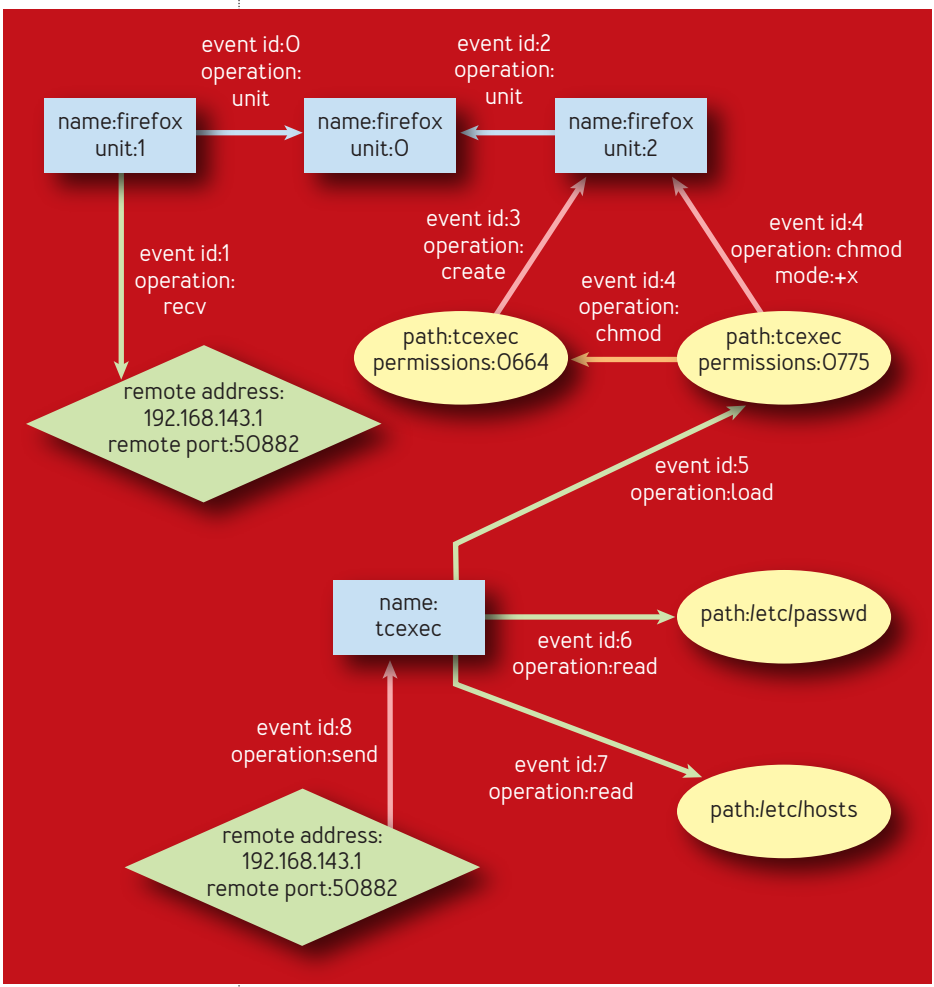
A forensic analyst can reconstruct what transpired with the following set of queries. At the outset, the analyst is assumed to know *a priori* that it was the `firefox` process that was hijacked after browsing a malicious website.

1. Determine if a web browser executed a file that was downloaded from a remote network connection.

(a) Get the vertices that represent a Firefox web browser.

```
$firefox = $base.getVertex("command line" LIKE '%firefox%')
```

FIGURE 2: **PROVENANCE RELATIONS BETWEEN PROCESSES AND ARTIFACTS**



(b) Get the vertices that represent a file that is world readable, writable, and executable.

```
$executableFiles = $base.getVertex(subtype
  == ‘file’ AND permissions == ‘0777’)
```

(c) Get the vertices that represent network connections.

```
$networkConnections = $base.getVertex(subtype
  == ‘network socket’)
```

(d) Get the paths where (1) a Firefox process reads data from a network connection, and (2) the same Firefox process updates permissions of an executable file.

```
$potentialAttackersEntryPath
  = $base.getPath($executableFiles, $firefox,
  1, $networkConnections, 1)
```

(e) Get the files that were executable and written by Firefox.

```
$potentiallyExecutedFiles
  = $potentialAttackersEntryPath &
  $executableFiles
```

2. Determine whether any written files were executed by the web browser.
(a) Get the vertices that represent processes.

```
$allProcesses = $base.getVertex(type == ‘Process’)
```

(b) Get the vertices that represent processes that were started by Firefox.

```
$firefoxChildren = $base.getPath($allProcesses,
  $firefox, 1).getEdgeSource()
```

(c) Get the Firefox children that accessed the written files.

```
$firefoxChildrenAccessedExecutableFile
  = $base.getPath($firefoxChildren,
$potentiallyExecutedFiles, 1).getEdgeSource()
```

3. Determine whether a process accessed sensitive system files and then sent information out through a network connection.

Get the vertices that represent system files `/etc/passwd`, `/etc/group`, and `/etc/hosts`.

```
$systemFiles = $base.getVertex(path
  == '/etc/passwd' OR path == '/etc/group' OR path
  == '/etc/hosts')
```

Get the paths from network connections that were written to by Firefox children that read system files.

```
$exfiltrationPath = $base.
getPath($networkConnections,
  $firefoxChildrenAccessedExecutableFile, 1,
  $systemFiles, 1)
```

Bitcoin

Bitcoin is used in dark web (and other) markets.[8] Each payment is made to a specific address that denotes

a user. Every successful transaction is recorded in a block that becomes part of a public ledger, the bitcoin blockchain. SPADE's Bitcoin Reporter can be used to infer the provenance graph that relates individual addresses, transactions, and blocks together. The next example assumes the blockchain has been imported into a database supported by QuickGrail. This allows forensic analysts to track the flow of funds through the bitcoin ecosystem. For example, they may wish to identify all the sources of a particular transaction. Alternatively, they may want to check if there is a path from one bitcoin address to another.

In this example, the analysts start with a bitcoin address found on a website soliciting donations to support illegal activity. Initially, they check whether a specific address has sent any payment. The search is limited to five levels of indirection.

```
$donation_address = $base.getVertex
  (address == '13Pcmh4dKJE8Aqrhq4ZZwmM1sbKFcMQEE')
$payer_candidate = $base.getVertex
  (address == 'ZwmbK4ZdKJ3PcQEmh8MEAqrhq41FcEM1s')
$paths = $base.getPath($donation_address,
$payer_candidate, 5)
```

Next, the analysts retrieve all payers whose funds reached the donation address either through direct payment or via an intermediary.

```
$payers = $base.getLineage($donation_address, 2, 'descendants')
```

LIFECYCLE OF A QUERY

Instructions to download, build, and run SPADE are available online.[3] Assuming it is running, the query client can be used interactively after it is started with the command **spade query** executed at the command line of a shell. It is also possible to pipe commands to it and responses from it by redirecting standard input and standard output, respectively.

The directive `set storage <name>` can be issued in the client to change the current default database. This assumes that the corresponding SPADE storage has been added previously. At this point, a session is created. Any queries made now will be sent to the selected database. A query session will continue until an `exit` command is issued.

Acknowledgments

This material is based on work supported by the National Science Foundation under Grant ACI-1547467. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Ahmad, R., Jung, E., de Senne Garcia, C., Irshad, H., Gehani, A. 2020. Discrepancy detection in whole network provenance, *Proceedings of the 12th USENIX Workshop on the Theory and Practice of Provenance (TaPP)*; https://www.usenix.org/conference/tapp2020/presentation/ahmad.

2. Fan, J., Gerald, A., Raj, S., Patel, J. 2015. The case against specialized graph analytics engines. *Proceedings of the 7th Biennial Conference on Innovative Data Systems (CIDR)*; http://cidrdb.org/cidr2015/Papers/CIDR15_Paper20.pdf.
3. Gehani, A., SPADE, http://spade.csl.sri.com.
4. Gehani, A., Kim, M., Zhang, J. 2009. Steps toward managing lineage metadata in grid clusters. *Proceedings of the First Usenix Workshop on Theory and Practice of Provenance (TaPP)* 7, 1-9; https://dl.acm.org/doi/10.5555/1525932.1525939.
5. Gehani, A., Kim, M. 2010. Mendel: efficiently verifying the lineage of data modified in multiple trust domains, *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC)*; https://dl.acm.org/doi/abs/10.1145/1851476.1851503 227-239.
6. Gehani, A., Tariq, D. 2012. SPADE: support for provenance auditing in distributed environments. *Proceedings of the 13th ACM/IFIP/Usenix Middleware Conference*; https://dl.acm.org/doi/pdf/10.5555/2442626.2442634.
7. Gehani, A., Kazmi, H., Irshad, H. 2016. Scaling SPADE to "Big Provenance." *Proceedings of the 8th Usenix Workshop on Theory and Practice of Provenance (TaPP)*,26-33; https://www.usenix.org/conference/tapp16/workshop-program/presentation/gehani.
8. Ghosh, S., Das, A., Porras, P., Yegneswaran, V., Gehani, A. 2017. Automated categorization of onion sites for analyzing the dark web ecosystem. *Proceedings of the 23rd ACM International Conference on Knowledge*

Discovery and Data Mining (KDD), 1793-1802; https://dl.acm.org/doi/10.1145/3097983.3098193.
9. Glavic, B. 2012. Big data provenance: challenges and implications for benchmarking. *Revised Selected Papers of the first Workshop on Specifying Big Data Benchmarks*, Volume 8163, 72-80; https://dl.acm.org/doi/10.1007/978-3-642-53974-9_7.
10. Khoury, J., Upthegrove, T., Caro, A., Benyo, B., Kong, D. 2020. An event-based data model for granular information flow tracking. *Proceedings of the 12th Usenix Workshop on the Theory and Practice of Provenance (TaPP)*; https://www.usenix.org/biblio-4496.
11. Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Plale, B., Simmhan, Y., Stephan, E., Van den Bussche, J. 2011. The Open Provenance Model core specification. *Future Generation Computer Systems* 27(6); https://dl.acm.org/doi/10.1016/j.future.2010.07.005.
12. Patel, J., Deshmukh,H., Zhu, J., Potti, N., Zhang, Z., Spehlmann, M., Memisoglu, H., Saurabh, S. 2018. Quickstep: a data platform based on the scaling-up approach. *Proceedings of the VLDB Endowment* 11(6), 663-676; https://dl.acm.org/doi/10.14778/3184470.3184471.
13. W3C Working Group. 2013. PROV-overview; https://www.w3.org/TR/prov-overview/.

Ashish Gehani *is a principal computer scientist at SRI in Menlo Park, California. His research interests are in data provenance and security. He holds a Ph.D. in computer science*

from Duke University and a B.S. in mathematics from the University of Chicago.

Raza Ahmad *is a research engineer at DePaul University. He developed SPADE's QuickGrail back ends for Neo4j and Postgres. His publications span machine learning, data science, provenance, and reproducibility.*

Hassaan Irshad *is a software engineer in the computer science laboratory at SRI. He is the primary maintainer of the open-source, distributed-data provenance framework SPADE. His work has focused on provenance data collection and analysis and has been published in multiple peer-reviewed venues.*

Jianqiao Zhu *is a software engineer at Google. He is a technical lead on the kernel execution team of the F1 Query engine. Before joining Google, he received his Ph.D. in computer science from the University of Wisconsin. He developed SPADE's QuickGrail and its back end for Quickstep.*

Jignesh Patel *is a professor of computer science at the University of Wisconsin, where he also holds affiliations with the biostatistics and medical informatics department and the Center for Creative Destruction Labs (which he co-leads). His research is in the area of database systems. He is a serial entrepreneur, and Fellow of the ACM and IEEE.*