Leveraging Data Provenance to Enhance Cyber Resilience

Thomas Moyer*, Karishma Chadha*, Robert Cunningham*, Nabil Schear*, Warren Smith*, Adam Bates†, Kevin Butler‡, Frank Capobianco§, Trent Jaeger§, and Patrick Cable¶

*MIT Lincoln Laboratory, Email: {tmoyer,karishma.chadha,rkc,nabil,warren.smith}@ll.mit.edu
†University of Illinois Urbana-Champaign, Email: adammbates@ufl.edu
‡University of Florida, Email: butler@cise.ufl.edu
§The Pennsylvania State University, Email: fcapobianco01@gmail.com, tjaeger@cse.psu.edu
¶Threat Stack, Inc., Email: pat@threatstack.com

Abstract—Building secure systems used to mean ensuring a secure perimeter, but that is no longer the case. Today’s systems are ill-equipped to deal with attackers that are able to pierce perimeter defenses. Data provenance is a critical technology in building resilient systems that will allow systems to recover from attackers that manage to overcome the “hard-shell” defenses. In this paper, we provide background information on data provenance, details on provenance collection, analysis, and storage techniques and challenges. Data provenance is situated to address the challenging problem of allowing a system to “fight-through” an attack, and we help to identify necessary work to ensure that future systems are resilient.

I. INTRODUCTION

Creating bigger and better walls to keep adversaries out of our systems has been a failing strategy. The recent attacks against Target [13] and Sony Pictures [15], to name a few, further emphasize this. It is untenable to assume that a system, even with designed-in security, can successfully repel all attacks. The next generation of secure systems must also be able to withstand successful attacks using cyber resilience. Cyber resilience broadly encompasses many areas including traditional fault tolerance, moving target techniques, and data provenance. In this paper we focus on the challenges and approaches to creating resilient systems using data provenance.

Data provenance is the history of ownership/processing or modification that we can use to guide its authenticity. Provenance is typically represented as a directed acyclic graph of nodes and edges that define the relationships between data, the processes that act upon them, and the users and others systems who controlled those processes. At face value, this sounds simple, even mundane. But it turns out that this kind of information is critical to answering questions that we have about our systems that are typically very difficult to answer. For example: Where are all my data? Where did they come from? Should I trust the data? How can I recover if something has gone wrong? These questions are only becoming more difficult to answer as the scale and complexity of our systems grows. It is critical that we know what a system has already done with its data, if we have any hope of fixing it after a successful attack.

Though provenance has hundreds of years of use in the art world and several decades of study in the literature surrounding data management [60], [20], it has only very recently been applied to improving system security and resilience [7]. Using provenance for securing systems presents a number of new challenges including efficiently collecting the data, securely encoding and storing it, and the timely analysis of the data to answer security-relevant questions. In order to leverage data provenance to enable secure and resilient systems, provenance data must be collected and analyzed. Today, few systems treat provenance as a first-class citizen. As a result, operators and engineers must be able to integrate provenance into their applications. This requires the appropriate technologies to support easy integration of provenance into their applications.

As a result of our experience developing both academic and operational provenance systems, we have developed both an architecture and best practices for addressing the challenges that surround provenance. In this paper we discuss these challenges and survey the solutions to each of the phases of the data provenance lifecycle from collection to its use in resilient self-healing systems. We discuss methods to limit the invasiveness and overhead of provenance on both developers and the operation of the system.

In this paper, we make the following contributions:

- provide an overview and background on existing tools to integrate provenance collection into resilient systems
- outline the challenges for securely leveraging data provenance in decision making
- detail gaps in existing tools to support end-to-end secure data provenance for resilient systems

Section III looks at provenance collection mechanisms that exist today. Sections IV and V describe how provenance is used, what tools exist for leveraging data provenance, and
what tools are still needed to fully utilize provenance. Finally, Section VI concludes.

II. RELATED WORK

Data provenance provides a means of reasoning about the flow of information through computer systems. However, data provenance differs significantly from previous work in the area. Below, we compare the goals and properties of provenance-aware systems to past work on information flow.

Many projects have aimed to support Information Flow Control (IFC) through instrumenting applications, operating systems, and programming languages. FlowwolF is a web browser that provides labeling support for distributed mandatory access control at the application layer [25]. Decentralized IFC systems like Asbestos [14], HiStar [59], and Flume [30] make use of a decentralized labeling scheme that allows processes to compartmentalize data that is protected by the operating system. This facilitates the use of a lattice-based model for information flow [12]. DStar [58] extends this approach to support distributed environments. In each case, the primary thrust of these works is to provide system-layer support to applications wishing to isolate user data, thus reducing the consequences of a compromise. IFC systems thus require a priori knowledge of desired flow properties, and are unable to answer questions such as “How did data object \( x \) come to have label \( a \)?” Provenance provides a means of reasoning about flows and answering these questions.

Another area of information flow study is dynamic taint analysis, in which the goal is to track the propagation of select pieces of data across a system. Automated instrumentation for taint tracking has been developed for x86 binaries [49] and smartphones [16], and dynamic taint analysis in sandbox environments has also been used to secure off-the-shelf applications [61]. Provenance can offer a more complete explanation as to how an object became tainted. It is also more flexible: taint tracking relies on an immutable policy that requires that data be flagged at runtime, while a provenance-aware approach to support distributed environments. In each case, the primary thrust of these works is to provide system-layer support to applications wishing to isolate user data, thus reducing the consequences of a compromise. IFC systems thus require a priori knowledge of desired flow properties, and are unable to answer questions such as “How did data object \( x \) come to have label \( a \)?” Provenance provides a means of reasoning about flows and answering these questions.

III. PROVENANCE COLLECTION

A necessary prerequisite to the use of data provenance to improve system resilience is its reliable capture and management, which is made possible through provenance-aware systems and applications. There are a variety of ways in which operators and engineers can deploy provenance-aware mechanisms to facilitate this capture. Provenance-aware mechanisms can also be divided into disclosed systems [17], [18], [36], [38], [45] and automatic systems. Disclosed systems can take the form of manual annotations or by processing documentation volunteered by the operator. In automatic systems provenance metadata is procedurally generated in the software itself. For the remainder of this work, we focus on automatic systems for their ability to provide more comprehensive provenance descriptions of system activity.

Automatic provenance-aware mechanisms can be deployed at various layers of system operation, including operating systems, infrastructure (i.e., middleware), and applications. Operating system sensors provide low-level detail on data processing from the system perspective [7], [34], [41], [39], [46]. System layer provenance also offers gapless descriptions of user space activity, effectively making every application that runs on the system provenance-aware. Unfortunately, the semantic gap between kernel and user space can make system provenance difficult to interpret, as exemplified by the dependency explosion problem in which each new output from a long running process appears to be dependent on all prior inputs [31]. In contrast, deploying capture mechanisms in infrastructure leads to provenance that is more semantically rich, while still offering broad coverage for a class of applications that make use of that infrastructure [18], [17], [50]. Due to the ubiquity of database backends in complex application workflows, an important subclass of provenance-aware infrastructure considers database management systems [11], [21], [27], [39]. However, to obtain the most precise and expressive provenance for an application workflow, it is necessary to invest in a manual instrumentation effort of the software. Doing so ensures a higher signal-to-noise ratio in the captured provenance, as the developer’s understanding of the workflow is encoded in the provenance itself. This effort is made easier through the presence of special APIs and libraries dedicated to provenance instrumentation [7], [20], [24], [35], [37], [39].

In practice, designing resilient provenance-aware systems does not require selecting a single provenance capture mechanism, but deploying a composition of the above mechanisms in order to provide total transparency to mission critical system activities. Below, we describe our past efforts in the design and implementation of interoperable provenance-aware components at different software layers.

A. Provenance-Aware Operating Systems

The Linux Provenance Modules (LPM) project is not only a provenance-aware operating system, but a generalized framework for the capture of data provenance that serves as
a trust anchor for other provenance-aware mechanisms [7].
LPM was designed specifically to provide reference monitor
guarantees [2] in the presence of an attacker that attempts
to subvert the provenance collection agent. For example, an
attacker may wish to manipulate provenance records in order
to commit fraud or inject uncertainty into data processing
results, as was the case in the “Climategate” controversy [47].

An overview of the LPM architecture is shown in Figure
1. LPM instruments the Linux kernel with a 178 dedicated
provenance collection hooks; these hooks are registered by a
provenance module, which also registers several Netfilter
hooks. As system events occur, the provenance module exam-
ines the event context and generates provenance records that
are shuttled out to user space for storage. To allow provenance
information to be securely transmitted between hosts, LPM
defines Netfilter functions that enforce a system-wide mes-
sage commitment protocol. The message commitment protocol
forces all messages transmitted between provenance-aware
hosts to be cryptographically verified using the Digital Sig-
nature Algorithm (DSA). We show in [7] how a machine can
securely boot into LPM through use of an Intel Trusted Boot
procedure, ensuring that LPM is able to collect provenance
prior to the start of mission-relevant system activities. We also
demonstrate the runtime integrity of LPM’s trusted computing
base through use of the SELinux MLS policy [26].

An especially important capability provided by the LPM
architecture is attested disclosure. As we discussed above,
resilient provenance-aware deployments require a composi-
tion of mechanisms and different system layers, but doing
so in a manner that preserves reference monitor guarantees
is an especially challenging problem. Applications in user
space, particularly network-facing services, are most at risk of
compromise; compromised applications may attempt to issue
false reports about their activities in order to inject uncertainty
into the provenance log. LPM addresses this by verifying the
integrity of applications with the Linux Integrity Measurement
Architecture (IMA) [48]. When an application wishes to report
provenance, it sends lineage metadata over a UNIX domain
socket to the Provenance Recorder. The Recorder recovers the
application’s process id over the UNIX socket, uses the /proc
filesystem to find the full path of the binary, and then uses this
information to look up the application in the IMA measure-
ment list. The disclosed provenance is recorded only if the
signature of application matches a known-good cryptographic
hash. This validation ensures that only known applications can
add provenance to the system record, preventing malware from
overflowing the log with extraneous provenance data. This
extra provenance information is recorded along with the LPM
provenance records, providing a more comprehensive look at
the processing of data. This layering is discussed in more detail
in Section III-E.

We performed a rigorous evaluation of the LPM system
in [7], and determined that the runtime overhead imposed by
provenance collection during heavy system load was just 2.7% -
7.5% (I/O intensive activities experienced the higher of these
overheads). We also showed that LPM provenance could be
queried to determine the expansive anarchies of system objects
in just tens of milliseconds, enabling its real time use in the
complex deployments explored in Section IV. The limiting
factor to LPM’s performance is high storage overhead, which
was on the order of gigabytes per hour under heavy load. We
are currently exploring new techniques to reduce provenance
storage overhead [6], as well as investigating how to adapt
existing techniques to LPM [10], [31], [32], [55], [56], [57].

Source code for the Linux Provenance Modules and sup-
porting utilities is available at http://linuxprovenance.org.

B. Provenance-Aware Infrastructure

The Database-Aware Provenance (DAP) [5] architecture
provides provenance capabilities to software infrastructure
through minimal-cost retrofits to existing application work-
flows. In designing DAP, we leveraged the observation that,
in many workflows, instrumentation efforts could be avoided
through introspection on the messages exchanged between
application components. Ubiquitous protocols such as HTTP
and SQL, as well as data marshaling languages like XML,
provide an open interface through which to infer the prove-
ance of the workflow. DAP is therefore comprised of a set
of communication proxies that parse application messages,
extract relevant semantics, and then record those semantics as
provenance. In [5], we consider a web service infrastructure
as an exemplar deployment scenario, shown in Figure 2. Red
lines mark a traditional web service data flow – incoming
HTTP/HTTPS requests are received by a web application,
prompting a series of database transactions that occur over a
local network socket, which are then used to craft a response
that is returned to the client.

DAP creates a modified, provenance-aware version of this
workflow as follows: (1) a remote client transmits a request
to the Web Application; (2) a small modification to the Web
Server notifies the Provenance Recorder that it has started a
new autonomous unit of work for fielding the request; (3)
the application’s query to the database is proxied by a
Database Capture Agent, which parses the query and extracts
the relevant operational semantics; (4) after observing the
Database Engine’s response to the query, the Database Capture
Agent (5) creates a new provenance event and transmits it
to the Provenance Recorder; (6) after the Web Applications
returns response to the remote client, (7) the Web Server

![Diagram of Database-Aware Provenance (DAP) architecture. Provenance-Aware components are shaded in orange. This deployment, DAP is able to capture provenance without requiring changes to the Database Engine or Web Application; instead, provenance is generated by interposing on the connection between the Web Application and Database Engine. A small change to the Web Server is required to facilitate execution partitioning.](image-url)
signals the Provenance Recorder that the current unit of work has ended. Because system events that occur outside of the web service architecture may also inform its execution, DAP interoperates with LPM, which generates provenance for all system activities that are not being explicitly reported by the Web Server or Database Capture Agent.

In [5], we benchmarked end-to-end delay and determined that DAP imposed just 5 ms overhead per web request.\footnote{17\% of total latency with client and server in VMs on the same host} We performed microbenchmarking to determine that the primary source of this overhead was processing delays at the Provenance Recorder, which could be addressed through the introduction of redundant or multi-threaded components. We also considered several scenarios in which DAP can be deployed to quickly determine the root cause of system attack, including SQL injection attempts, reverse shell invocations, and a vulnerable system library exploit. Finally, through the introduction of a security mechanism that authorizes individual server responses in real time, we can discard older workflow provenance for responses that have already been released to a client. This reduces the growth of the DAP provenance log from linear to logarithmic with respect to the number of web requests.

We are not the first to explore the modification of application infrastructure to capture provenance. Many database and scientific workflow engines have been modified to capture provenance without modifying the applications themselves. Example systems include databases like Trio [54], scientific workflow applications including VisTrails [9] and Taverna [44], and purpose-built systems for research [23]. Hadoop is another popular application that has been modified to collect provenance without requiring Hadoop users to modify their code [1].

C. Provenance-Aware Applications

In addition to OS and infrastructure collection, making applications provenance-aware allows for contextually-rich provenance information from applications. In order to achieve the goal of provenance-aware applications, libraries are needed for developers to emit provenance from their applications. One such library is ProvToolbox [37], a library that presents an implementation of the W3C PROV specification for Java applications. In addition to using these libraries for applications, other tools that collect provenance, such as the ones described above can leverage these libraries to emit provenance in a standard way. This ensures that provenance collected at multiple levels within the system have a common representation.

D. Provenance Storage Considerations

With many of the systems described above, storage is often an afterthought, especially for the OS-level provenance collectors. Flat-files are used because they are easy to create and manage, but present challenges when provenance analytics need to access the data. Using a database is an option, but the volume of data can quickly become problematic. The LPM system explored the use of several different storage mechanisms, ultimately leveraging an in-memory graph database built on the SNAP library [33].

In order to address the volume of data generated by the provenance system, others have looked at deduplication and web encoding techniques [8]. This hybrid approach to provenance storage shows promise in reducing the overall storage overhead of provenance collection. Such approaches can help alleviate, but not eliminate, potential denial of service attacks where an attacker fills the provenance store with garbage data.

Another consideration for provenance storage is securing the stored provenance against malicious modification and deletion. As more systems are built to leverage provenance as a resilience mechanism, protecting the data becomes vitally important. Digital signatures and encryption are common techniques that are currently being integrated into database systems [52], [28]. As cryptographic primitives become available, provenance systems can leverage signatures and encryption to protect the data. One problem that remains is detecting deleted records. Even using hash chains, an adversary can truncate the record. One possible solution is to leverage blockchains (e.g. bitcoin [43]) that have periodic, public, commitments of data to prevent truncation of provenance records.

E. Provenance Layering

While each of the above collection methods provides a view of the data history, a more complete view is achieved by layering provenance collection. With application-level provenance, developers are required to manually instrument their applications. Any missed operations results in missing provenance. Using infrastructure and OS provenance to fill-in these gaps ensures a complete provenance record. By layering sensors, developers can also focus on those applications that are most critical, and leave less critical applications to the other sensors. This targeting of effort reduces the overall workload on the developer.

Layering provenance sensors provides additional security benefits as well. If an application is reporting provenance, but becomes compromised, the other provenance sensors can still reliably collect provenance on the actions taken by the now malicious application that is reporting provenance. This defense-in-depth approach to collecting provenance ensures a complete record, even in the face of attack. As noted in [40], layering also presents challenges that must be addressed.

IV. LEVERAGING PROVENANCE

The United States Department of Defense and the Intelligence Community operate a wide variety of distributed data processing applications. These applications are critical to the success of the missions of these organizations and it is therefore important that the applications are resilient to security issues as well as the types of failures that occur in a distributed system.

Provenance information can be used for a number of purposes in such data processing systems. If a data item enters the system and is later determined to be invalid, a taint tracking service can use provenance information to locate all data derived from that invalid data item. A security service can use provenance data to identify suspicious activity such as data access from unexpected users, unexpected locations, or at unexpected volumes. An application monitoring service
can use real-time provenance information to identify service failures, such as when a transformer is not creating new data. Finally, a data validation service can use provenance information to determine if data from unknown sources enters the system.

Even though data processing applications are created for different purposes, our experience indicates that many of them have an architecture similar to the one shown in Figure 3. This similarity currently makes it easier for us to manually add provenance instrumentation, but in the future, we plan to take advantage of these similarities and perform automated instrumentation and analytics.

The common data processing architecture contains a set of services that ingest data into the system from external sources and then publish it as messages to a message bus such as ActiveMQ [51] or RabbitMQ [53]. These messaging systems support a publish/subscribe model where consumers subscribe to receive information (for example, based on a topic), publishers send messages to the messaging system, and the messaging system delivers messages to the appropriate subscribers.

One type of subscriber we often see is an archive, or persister, service that receives messages that contain data to be stored and writes that data to a database. The database may be a vertically-scaled relational database such as Oracle [22] or a horizontally-scaled database such as Accumulo [3]. The archived data may come from ingesters or from transformers (described next) that generate derived data.

Transform services subscribe to the bus and when they receive data, they operate on it and publish derived information back to the bus. In addition, if a transformer needs data from the archive, it can use the message bus to request it from a query service that retrieves data from the database and replies over the message bus. Finally, users often interact with this sort of system via a web interface that receives information from application-specific web applications running in a framework such as Apache Tomcat [4]. The web applications typically interact with a query service that performs database queries and may also interact with the message bus to control the overall system. One example of a transform service is a logistics planning application that receives requests to move goods between two locations. The service aggregates these requests into a set of requirements describing what must be moved where and by when. It then performs an optimization on to achieve the best possible schedule and cost while preserving the requirements.

To verify the effectiveness of provenance in data processing systems that follow the architecture above, we instrumented a logistics planning application to emit provenance information. This application ingests requirements that describe the items to be transported, transforms those requirements and the state of the transportation system into a movement plan, archives the requirements and plan to a database, and presents information to users via a web interface. We implemented real-time analytics on the provenance information to determine if logistics data from unknown sources entered the system. We found that provenance data can be used to accomplish this goal, but the detection required knowledge of the application. We wrote application-specific code that analyzed the provenance graph for each planning phase and determined if it matched expectations. For example, the analysis code checked that the requirements used in planning were the same requirements generated by the web-based user interface for the logistics application. The result of this analysis was presented in the user interface by showing the provenance graph with a green or red background and if red, an explanation why the provenance graph was incorrect. Without data provenance, the system is unable to detect deviations from the known-good workflow (i.e. users submitting requests via the web interface) when the adversary injects new data into the database. To detect this without provenance would require auditing of database and OS logs. A process typically done manually that takes substantially longer than our automated provenance analytics.

We find that the overhead of generating and analyzing provenance information in this logistics application is minimal. Figure 4 shows that collecting provenance information increases the execution time of the planning phase of the logistics application by 4%. The planning phase consists of generating and ingesting requirements, archiving and querying requirements and plans, and generating a logistics plan. The provenance data was published to the message bus in the same threads that execute application tasks. The execution time overhead could therefore be lowered using techniques such as using a separate thread to publish provenance information. Figure 5 shows that storing provenance information in a relational database increases the amount of storage needed by approximately 1% when compared to the amount of application data generated for a planning phase (for example, requirements and log files). Furthermore, the storage overhead is much less than 1% when compared to the application data used during a planning phase (the definition of the state of the available transportation system).

In future work, we will develop automated methods to detect anomalies in provenance graphs so that we do not have to implement application-specific detection. We expect that these methods will combine general heuristics as well as automated comparison to past provenance graphs that have been labeled as valid or invalid. Furthermore, since many data processing systems have architectures similar to Figure 3, we will investigate whether adding provenance instrumentation to common services such as message buses, databases, and web application frameworks will allow us to reduce the amount of instrumentation that needs to be added into application code.

Fig. 3. A common architecture for data processing applications.
provenance data requires systems with more processing power than the machines who are sending it. Curator (another tool going through the open source process) is our answer from taking provenance from multiple sources on a system (e.g. LPM, Userspace Provenance Library, etc.), then shipping it off to a variety places where it can be stored, such as Accumulo, Neo4j, or other delimited file formats.

B. Future Tooling

While many tools currently exist for collecting, storing, and analyzing provenance, there is still work to be done. First, many collection mechanisms require a software agent on the system to store provenance on a remote server. Existing prototype agents (SPADE and Curator) are large and require heavyweight infrastructure, such as the JVM [20]. While this may work for certain environments, other environments cannot support a full JVM, and smaller, self-contained agents are required. These agents should be compatible with existing infrastructure, and integrate cleanly into the many different systems that we anticipate will benefit from provenance.

Another limitation of leveraging provenance at the application layer is the lack of provenance-aware applications. In order to make applications provenance-aware, developers must manually instrument the applications to emit provenance information. For certain high-value applications this is reasonable, but for many legacy applications, there is little incentive to provide the appropriate instrumentation. Instead, automated hook placement techniques can be used to instrument applications. This is work that we are just beginning to explore, leveraging existing work in automated authorization hook placement [19], [29], [42].

VI. Conclusion

Bigger and better walls around our systems will not prevent adversaries from gaining access to our systems, and wreaking havoc. Instead, building in protections that will allow systems to gracefully recover from attack are needed. Data provenance is one such protection that is reaching a point where developers and engineers can leverage existing tools to collect, store, and analyze provenance data. This data allows engineers to answer difficult questions about data being processed, and protect that data, even in the face of adversaries that have breached the perimeter defenses. While many of the tools and libraries are mature, there is still work left to fully realize the potential of data provenance in systems. This work looked at the overall landscape and presented the reader with an overview of the tools that exist today, and what tools are actively being developed. What is still needed is a community-driven effort to build and maintain these tools over time, and enhance the capabilities of data provenance.
ACKNOWLEDGEMENTS

We would like to thank the anonymous reviewers for the valuable feedback, and the members of the Secure Resilient Systems and Technology group. This work was supported in part by the US National Science Foundation under grant numbers CNS-1540216 and CNS-1540217.

REFERENCES
