A Method for the Systematic Generation of Audit Logs in a Digital Preservation Environment and Its Experimental Implementation In a Production Ready System

Hao Xu
DICE Center
University of North Carolina at Chapel Hill
Chapel Hill, NC
xuh@email.unc.edu

Jason Coposky
iRODS Consortium
RENCI
Chapel Hill, NC
jasonc@renci.org

Dan Bedard
iRODS Consortium
RENCI
Chapel Hill, NC
danb@renci.org

Jewel H. Ward
SILS
University of North Carolina at Chapel Hill
Chapel Hill, NC
jewel_ward@unc.edu

Terrell Russell
iRODS Consortium
RENCI
Chapel Hill, NC
tgr@renci.org

Arcot Rajasekar
SILS
University of North Carolina at Chapel Hill
Chapel Hill, NC
sekar@renci.org

Reagan Moore
SILS
University of North Carolina at Chapel Hill
Chapel Hill, NC
rwmoore@renci.org

Ben Keller
iRODS Consortium
RENCI
Chapel Hill, NC
kellerb@renci.org

Zoey Greer
iRODS Consortium
RENCI
Chapel Hill, NC
tempoz@renci.org

ABSTRACT

In a digital preservation environment there is a need for a complete auditing of the change of the system state. A complete log ensures that the properties of the objects in the system can be verified. Modern data management systems such as the integrated Rule-Oriented Data System (iRODS) allow administrators to configure complex policies. Pre- or post-operation, these policies can trigger other state changing operations. In this paper, we describe a method that allows us – given a complete list of state changing operations – to generate a complete audit log of the system. We also describe an experimental implementation of the framework. An important advantage of our method is that not only do we build on sound theoretical foundations, but we also validate the methodology in a production ready environment which has undergone substantial quality control. The implementation of our method can be distributed as a turnkey solution that is ready to deploy, which significantly shortens the gap between theoretical development and practical applications.

General Terms

Infrastructure opportunities and challenges

Keywords

audit log, production system, implementation, digital preservation, policies, automated log generation

1. INTRODUCTION

Researchers and practitioners at the Digital Curation Centre (DCC) have defined digital curation as involving "maintaining, preserving and adding value to digital research data throughout its lifecycle" [10]. A data manager begins curation at the time the collection is assembled or acquired. He or she actively manages the collection in order to "mitigate the risk of digital obsolescence" and "to reduce threats to [the data's] long-term research value" [11]. According to DCC researchers and practitioners, auditing is one part of the active curation of a preservation system, and provides a means to ensure stored data has integrity and may be trusted.

When an organization audits a digital repository, two primary standards are used: ISO 14721:2012 [12], the Open Archival Information System (OAIS); and, ISO 16363:2012 [13], the Audit and Certification of Trustworthy Digital Repositories. The former is an ISO standard and reference model that defines an archive as something "consisting of an organization of people and systems, that has accepted the responsibility to preserve information and make it available for a Designated Community". The latter recommendation

is based on the OAIS Reference Model [7]. ISO 16363 defines a recommended practice for assessing the trustworthiness of digital repositories. It may be used for all types of digital repositories, regardless of content type, and as the basis for certification of the archive as "trusted" by independent auditors. An important aspect of such auditing activity is to show that the dynamic behavior of the digital repository and the digital curation activities are actually being implemented with regards to the objects in the digital preservation system. In previous work [1], we've shown that a large part of this type of auditing can be checked by inspecting an audit log of state changes. In this paper, we further develop this idea by providing an implementation framework.

Auditing has important implications beyond digital preservation for its own sake. The healthcare and financial services industries, for example, are subject to government privacy and records retention regulations. Administrators of healthcare data need to be able to prove to regulatory agencies that patient records have only been available upon patient consent. Financial records are subject to retention policies, and need to be protected from tampering.

Auditing can also play an important role in industries that are not subject to extensive regulatory requirements, where it can provide insight into illegal hacking activity. Sensitive internal records–HR data and corporate finances–must be protected from unintended access and release. Auditing, with appropriate detection algorithms, can provide administrators with real-time insight into unusual file system activity. In the event that data is compromised, auditing can provide an evidence trail for prosecution, as well as the ability to deconstruct an attack to develop methods to interrupt similar attacks in the future. Data management auditing provides the ability to guarantee regulatory compliance and to safeguard against malicious activity.

In this paper, we propose a implementation framework that allows us to systematically generate a complete audit log of the system, given a complete list of state change operations. We also describe an experimental implementation of the framework and discuss which features enable such implementation. Another innovation in our implementation is that we use the same policy enforcement mechanism for implementing application domain policies to implement auditing, making auditing part of the policies. This reduces duplicate code paths and enable higher test coverage. It also enables interesting use cases such as auditing the auditing mechanism itself, and raises questions concerning how to ensure the termination of such auditing rules.

2. THE METHODOLOGY

In previous work [1], we have shown that a digital repository can be seen as a state transition system and policies related to preservation properties can be described in terms of legal or illegal state transitions. With this process, we tie policy certification to checking the legality of a sequence of state changes. This allows us to implement auditing in a digital preservation environment by providing a complete log of the change of the system state. Existing ad hoc methods do not guarantee the completeness of the audit log. To add to the complexity, modern data management systems such as iRODS allow administrators to configure complex policies to meet the requirements of different application domains. Pre- or post-operation, these policies can trigger other state changing operations. Further, the policies can be nested. These policies are usually executed from a policy language, which are sometimes Turing-complete programming languages, in which case the list of operations performed cannot be easily determined statically. Also, because of the complexity of policies, it would be inefficient to keep track of all commands in the rule language. Rather, we only want to audit the commands that change the system state.

An audit log is considered "complete" for making assertions about preservation properties when we capture all state changes. In a computer system, an important type of state change occurs when a state change operation is applied. The majority of state change operations include: user interaction, time triggered operations, and action trigger operations. The notable exception are state changes by hardware failure, which cannot be entirely addressed on the software level. (This type of state change is often partially addressed by redundancy. When redundancy is set up, we can indirectly capture this kind of state change through certain verification and recovery operations, for example, checking the checksum. The method described in this paper is therefore applicable, albeit indirectly, to this type of state change).

In order to systematically capture this type of state change, we need to find a way to systematically enumerate all state change operations and their applications and keep track of every state change operation.

We can systematically enumerate all state change operations by categorizing them by the different types of effects. For example, a subset of operations supported by the integrated Rule-Oriented Data System (iRODS) [3] is shown in Table 1. The list of database operations, resource operations, and network operations, etc. are fixed, whereas other types of plugins such as microservice are designed for extensibility, therefore no fixed operations are listed.

There is a question of whether the list of operations generated thus far is complete. To show the completeness, we can separate the part of the software that changes the state, or is effectful, from the part of the software that does not change the state, or is effect-free. The effect-free part of the system talks to the effectful part of the system through a well-defined application programming interface (API). The operations in the API map directly to the operations we enumerated. If we capture all API calls across the effectful-effect-free boundary, we capture all state changing operations.

Given a complete list of operations, and a mechanism to capture every call of every operation, we can ensure the completeness of the log. Immediately before and after the application of the operations, we record the event in the log. The implementation, which we will go into details in the next section, will discuss how we capture this information in a production ready system. A complete history of the system can be reconstructed when an administrator inspects the log. By providing the availability of the history, we can verify that the digital repository is compliant with

Plugin Type		Plugin Operation
Resource		create open read write stagetocache synctoarch registered unregistered modified resolve_hierarchy rebalance
Authentication		establish_context agent_auth_verify
Network		client_start client_stop agent_start agent_stop read_header read_body write_header write_body
Database	replica	reg_replica unreg_replica
	data object	reg_data_obj rename_object move_object
	collection	reg_coll_by_admin reg_coll mod_coll rename_coll del_coll_by_admin del_coll
	metadata	mod_data_obj_meta set_avu_metadata add_avu_metadata_wild add_avu_metadata mod_avu_metadata del_avu_metadata copy_avu_metadata del_unused_avus
	resource tree	add_child_resc reg_resc del_child_resc del_resc mod_resc mod_resc_data_paths mod_resc_freespace get_hierarchy_for_resc substitute_resource_hierarchies
	zone	reg_zone mod_zone rename_local_zone del_zone get_local_zone
	user	del_user check_auth make_temp_pw mod_user make_limited_pw reg_user
	access control	mod_access_control gen_query_access_control_setup
	quota	calc_usage_and_quota set_quota check_quota
		start open close rollback commit
Microservice		<microservice_name>
API		<api_name>

Table 1: Plugin Operations

the pre-established policies. Furthermore, the API can be modularized such that the effectful part can be encapsulated into modules and they can be loaded dynamically at run time. This provides flexibility of features yet still guarantees the completeness of the audit log.

To prevent users from inadvertently circumventing our software abstraction,[1] we ensure that the user can only modify the system state through these operations by virtualizing the storage and providing strict access control. The virtualization of the storage ensures that the users are not exposed to low level APIs that could potentially modify the system by bypassing the system-provided operations. Strict access controls ensure that the user cannot inadvertently bypass the virtualization.

3. IMPLEMENTATION

We describe an implementation of our method in iRODS. We choose to implement our method in iRODS because it provides several key features that enable a direct translation of our framework code. Also, the industrial level code quality allows us to bring our implementation to the production system.

iRODS is a state-of-the-art open source software system for addressing the key data management tasks that face users as the size and complexity of digital data collections continue to grow rapidly. Because the principal data management tasks are highly interrelated, rather than taking a piecemeal approach or addressing just a single task, the iRODS system takes a comprehensive approach to full data life-cycle management.

At the same time, the system design is highly user-driven and avoids the pitfalls of a "one size fits all" design by building on a comprehensive generic platform with a highly configurable architecture. In addition, iRODS offers multiple paths to interoperation with outside systems such as repositories, interfaces, and applications. This lets users adapt iRODS to the details of their own environment in a wide range of production applications that can emphasize different aspects of data management in diverse domains.

Furthermore, iRODS has undergone strict quality assurance. We repaired over 1100 identified defects in the 4.1 core code. Using Coverity alone has vastly improved iRODS stability, and coupled with the other tools deployed within our continuous integration (CI) infrastructure, iRODS is in an enterprise production-ready state. In continuous topology testing of multiple machines, our JSON-based Zone descriptions are now ingested by an Ansible-driven engine which deploys a full iRODS topology into our VMWare cloud infrastructure. The current basic test deployment runs a full feature testing suite from multiple types of configurations on every commit to our GitHub repository.

In the nine years since iRODS was first released, the software has been adopted for the support of a variety of research activities. iRODS is in use at over one hundred universities around the world, not only for preservation activities in digital repositories, but also in support of domain-specific research. This utility has begun to spread into the commercial sector, beginning with the life sciences industry. Bioinformaticians use iRODS for its ability to associate data with user-defined metadata and to track the provenance of data as it matures from raw data into a final work product. Gradually, iRODS uptake has begun to spread to other fields, with proofs of concept emerging in the oil exploration and entertainment industries. We expect that iRODS will continue to find use in additional fields, such as the financial services and manufacturing industries.

In the following subsections, we describe iRODS components that enable the design of a high performance auditing system, and an overview of how the auditing system is imple-

[1]Defending against Byzantine error is out of the scope of this paper.

mented.

3.1 Plugin architecture

iRODS has a plugin architecture. This can be seen as a design effort is to move all effectful operations into plugins, and leave the core effect-free. This separation of effectful code from effect-free code allows us to make assertions about state-changing operations through just the observation of interactions with plugins, by defining rules that are dynamically enabled with the dynamic loading of plugin operations. Besides the default supported plugins, the set of supported effectful operations can be extended through microservice plugins.

3.2 Policy enforcement points

iRODS implements the concept of pre- and post-operation policy enforcement points, or PEPs. These PEPs allow system administrators to define rules to be executed either before or after each operation. The policies in a preservation system can then be encoded as rules.

iRODS contains two types of built-in PEPs:

Pre- and post- operation PEPs: these PEPs are triggered before and after an operation is executed. Each operation has a pair of pre- and post- PEPs. User defined rules can be executed at these PEPs to customize the execution of the operations.

Configuration PEPs: these PEPs are triggered at certain points of configuration. Each configuration has one PEP. User defined rules can be executed at these PEPs to customize the configuration of the system.

Built-in PEPs can be extended by dynamic PEPs. For every plugin operation that is called, two policy enforcement points are constructed (both a pre- and post- variety), and if it has been defined in any other loaded rulebase file, they will be executed by the rule engine. The PEP will be constructed of the form `pep_`P`_pre` and `pep_`P`_post`, where P is the operation. For example, for `resource` plugin type, `create` operation type, the two PEPs that are dynamically evaluated are `pep_resource_create_pre` and `pep_resource_create_post`. If either or both have been defined in a loaded rule base, they will be executed as appropriate.

A formal definition of the semantics of PEPs are given in [2]. The flow of information from the pre- PEP to the plugin operation to the post- PEP works as follows: `pep_`P`_pre` should produce information that will be passed to the calling plugin operation. The calling plugin operation will receive any information defined by `pep_`P`_pre` and will pass its own information to `pep_`P`_post`. `pep_`P`_post` will receive any information from the calling plugin operation. A map data structure is made available within the running context of each dynamic PEP based on the plugin type of interest. They are available via the rule engine in the polices.

For example, when running

```
iput -R myOtherResc newfile.txt
```

a `create` operation is called on a resource plugin to create the file. This delegates the call to the actual plugin instance's `create` operation. When `pep_resource_create_pre` PEP rule is evaluated, the values about the file are available for the policy. This allows rule authors to make decisions at a per-resource basis for this type of operation.

3.3 Pluggable rule architecture

The policies are defined at pre- and post-operation PEPs as rules. These rules are executed through a set of rule engines. The pluggable rule architecture allows multiple rule engines to be dynamically and concurrently loaded. Different rule engines can support different languages with the libraries of that language. Every rule engine is automatically equipped with the capability of calling microservices through a single interface. Through the same interface one rule engine can call rules across the rule engine boundary from another rule engine.

This way different rules can be written while taking advantage of the features of different languages, yet still work coherently together. Full compatibility is guaranteed by design with rules written for earlier version of iRODS. Currently, the available rule engine plugins include the iRODS rule language and Python. High performance, natively executed rules can also be written in C++, eliminating the need to go through the microservice interface. Our implementation takes advantage of this capability to provide high performance auditing of the system.

3.4 Auditing policies plugin

The semantic goal of the auditing policy plugin is to provide a complete auditing history to the system without significantly modifying the behavior of the system, including the built-in behavior of the operations and user defined policies. By "not significantly", we mean, low runtime overhead and no change to the semantics of the operations[2].

The auditing plugin provides a turnkey solution to providing the auditing capability to an existing iRODS deployment. The pluggable rule architecture allows users to enable auditing through one switch without interference with existing rules in the system. The code is written in C++ and is compiled and run natively, imposing a much smaller overhead compared to written in an interpreted language such as Python or the iRODS rule language. The events can be arbitrarily filtered, further reducing the overhead for diagnosing a specific type of issue.

The auditing plugin is implemented as a rule engine plugin. The plugin listens to a specific set of events on the server. This set of events include all plugin operation calls. When it receives the event of a plugin operation call, it serializes the calls and the parameters and writes them to the log. This way the log can be parsed and sent to the ELK stack for analysis (Elasticsearch, Logstash, and Kibana) [5].

Since we have shown that the audit log is complete with regard to effectful operations, we can ask the same question of the auditing mechanism itself. How do we know that the

[2]in contrast to data management policies which may change the semantics of operations

audit mechanism does what it says it does? How do we audit the auditing rule? Why not let the auditing rules audit their own execution, which would close the loop?

Letting the auditing rules audit themselves may lead to an infinite loop. Consider the following example: User A initiates Action B. Prior to Action B, the auditing rules are triggered. This leads to an action which is the execution of the auditing rules, before which the auditing rules are triggered again to audit this new action. This leads to an infinite loop. This rules out the simple solution of self-auditing, if we want the auditing rules to terminate.

This is analogous to Russell's Paradox [4], and a classic solution is stratification – we can define a hierarchy of rule execution levels, the lowest being normal rule execution. Each upper level is responsible for auditing the level below. This way, we can provide arbitrary levels of auditing. However, this approach has the following limitation: the execution of the highest level auditing rules is not audited by any other level. We have to trust that they do what they say they do. This can usually be remedied by extensive testing.

More formally, we assign an integer "level" to each action. The normal actions are on level 0. The action of execution of auditing rules triggered by level x action is on level $x+1$. We define a cutoff level, say 2, such that actions of this level do not trigger auditing rules. This allows us to show that the rules for generating the audit log always terminate, which is necessary, because a diverging policy modifies the underlying system in a significant way.

4. RELATED WORK

Currently, practitioners and researchers in the digital library community have developed a series of self-auditing mechanisms and independent certification of a repository as "trustworthy". The Center for Research Libraries [6] has audited a handful of digital libraries and archives and certified them for trustworthiness based on ISO 16363:2011 and ISO 14721:2012. Further work is ongoing to define the requirements for certification of an organization that wishes to provide certification services [8, 9], and to define how those certification requirements will be upheld and monitored themselves. The research outlined in this paper provides a method for proving that required state changes have occurred when certifying a digital repository against a set of policies.

5. CONCLUSION

In this paper, we propose a framework that allows us to generate a complete auditing log of the system, given a complete list of state changing operations. We also describe an experimental implementation of the framework in iRODS.

6. ACKNOWLEDGEMENT

This research is partially supported by the National Science Foundation under Grant Number OCI 0940841

7. REFERENCES

[1] Ward, Jewel H., et al. "Using Metadata to Facilitate Understanding and Certification of Assertions about the Preservation Properties of a Preservation System." Metadata and Semantics Research (2013): 87.

[2] Xu, Hao, et al. "Building an extensible file system via policy-based data management." Proceedings of the 1st ACM International Workshop on Programmable file systems. ACM, 2014.

[3] Ward, Jewel, et al. The Integrated Rule-Oriented Data System (iRODS 3.0) Micro-Service Workbook. DICE, Data Intensive Cyberinfrastructure Foundation, 2011.

[4] Wikipedia. Russell's Paradox. http://en.wikipedia.org/wiki/Russell%27s_paradox

[5] ELK. https://www.elastic.co

[6] Center for Research Libraries. (2015). Certification and Assessment of Digital Repositories. Retrieved April 19, 2015, from http://www.crl.edu/archiving-preservation/digital-archives/certification-assessment

[7] CCSDS. (2012). Reference Model for an Open Archival Information System (OAIS) (CCSDS 650.0-M-2). Magenta Book, June 2012. Washington, DC: National Aeronautics and Space Administration (NASA).

[8] CCSDS. (2011). Requirements for bodies providing audit and certification of candidate trustworthy digital repositories recommended practice (CCSDS 652.1-M-1). Magenta Book, November 2011. Washington, DC: National Aeronautics and Space Administration (NASA).

[9] CCSDS. (2011). Audit and certification of trustworthy digital repositories recommended practice (CCSDS 652.0-M-1). Magenta Book, September 2011. Washington, DC: National Aeronautics and Space Administration (NASA).

[10] Digital Curation Centre. (2010). What is digital curation? Retrieved April 10, 2015, from http://www.dcc.ac.uk/digital-curation/what-digital-curation

[11] Digital Preservation. (2009). Introduction - definitions and concepts. Digital Preservation Coalition. Retrieved April 10, 2015, from http://dpconline.org/advice/preservationhandbook/introduction/definitions-and-concepts

[12] ISO/IEC 14721. (2012). Space data and information transfer systems – Open archival information system – Reference model. Retrieved April 10, 2015 from http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=57284

[13] ISO/IEC 16363. (2012). Space data and information transfer systems – Audit and certification of trustworthy digital repositories. Retrieved April 10, 2015, from http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=56510

[14] Steinhart, G., Dietrich, D., and Green, A. (2009). Establishing trust in a chain of preservation the TRAC checklist applied to a data staging repository (DataStaR). D-Lib Magazine 15(9/10). Retrieved April 10, 2015 from http://www.dlib.org/dlib/september09/steinhart/09steinhart.html