# AccessProv: Tracking the Provenance of Access Control Decisions

Frank Capobianco
The Pennsylvania State University
fnc110@cse.psu.edu

Christian Skalka
The University of Vermont
skalka@cs.uvm.edu

Trent Jaeger
The Pennsylvania State University
tjaeger@cse.psu.edu

## Abstract

Access control protects security-sensitive operations from access by unauthorized subjects. Unfortunately, access control mechanisms are implemented manually in practice, which can lead to exploitable errors. Prior work aims to find such errors through static analysis, but the correctness of access control enforcement depends on runtime factors, such as the access control policies enforced and adversary control of the program inputs. As a result, we propose to apply provenance tracking to find flaws in access control enforcement. To do so, we track the inputs used in access control decisions to enable detection of flaws. We have developed AccessProv, a Java bytecode analysis tool capable of retrofitting legacy Java applications with provenance hooks. We utilize AccessProv to add provenance hooks at all locations that either may require access control enforcement or may impact access control policy decisions. We evaluate AccessProv on OpenMRS, an open-source medical record system, detecting access control errors while incurring only 2.1% overhead when running the OpenMRS test suite on the instrumented OpenMRS program.

## 1. Introduction

Access control provides a line of defense to prevent exploits by blocking unauthorized access. However, implementing access control correctly is a complicated task. Programs that implement an access control mechanism must achieve the requirements of the *reference monitor concept* [2], which demands complete mediation of all security-sensitive operations among other requirements. In addition, each deployment must enforce an access control policy that enforces the security intentions of that deployment.

Since both the access control mechanisms and policies are implemented manually, there is the possibility of exploitable errors. However, systems tend to assume that access control is implemented correctly, so forensic analysis often does not focus on limitations in access control enforcement. As a result, forensic analysis may miss the root cause of an attack that occurred because the access control implementation was incorrect, which may leave the vulnerability to enable further exploits.

Researchers have proposed to improve the effectiveness of forensic analysis by building systems that track whole-system provenance [3, 6, 12]. Whole-system provenance typically tracks the system calls performed by programs to collect information about the accesses to system objects. However, if the program being tracked is trying to restrict use of such objects itself by enforcing its own access control policy, errors in such enforcement will be missed.

In this paper, we explore mostly-automated methods to track the provenance of access control decisions. First, we propose to log the inputs to access control decisions. We propose a static analysis to construct *access dependence graphs* (ADGs), which relate all the inputs that may impact access control decisions for related security-sensitive operations. Second, we propose to record the possible sources of such inputs in the log. We then use the ADGs to instrument programs to log inputs and their possible sources. Third, instrumented programs generate *access provenance graphs* (APGs) upon execution, recording access decision inputs and relating them to the ADGs. We find that using ADGs supports forensic analysis, enabling analysts to compare operations authorized by the same hooks (or not mediated) and compare related operations authorized by different hooks.

We developed a system called AccessProv to perform the above steps, which is applied to OpenMRS [11], an open-source medical record system implemented in Java and deployed throughout the world. OpenMRS implements its own access control enforcement, but we identify several cases where the enforcement may be insufficient depending on the deployment's policy and intended security goals. AccessProv instruments OpenMRS to log the inputs to access control decisions and their possible sources.

Running the OpenMRS test suite on the instrumented OpenMRS produces a log of 2.6MB of information, only costing 2.1% overhead over the base implementation. Using the ADGs to guide analysis of the resulting log, we find evidence that three ambiguous cases of access control enforcement are flawed.

## 2. Problem Definition

Typically, provenance methods assume that the application of access control is correct, capturing the system calls executed that may have led to a problem. However, the problem itself may be caused by errors in the access control implementation, enabling processes that should not have been authorized in the first place to modify or access data objects being tracked. Thus, we argue that the provenance of access control decisions should also be tracked.

Errors in access control are possible because access control enforcement is implemented and configured manually. While the *reference monitor concept* [2] specifies the requirements for deploying access control correctly, some of the requirements necessitate judgments on the part of programmers. For example, consider the requirement that the reference monitor be invoked for each *security-sensitive operation*. Programmers have to determine which of their program statements imply security-sensitive operations, and add a call to the reference monitor, called an *authorization hook*, to mediate all accesses to all security-sensitive operations. If every one of a program's security-sensitive operations is preceded by an authorization hook that invokes a reference monitor, the program enforces *complete mediation*.

To make these notions concrete, we examine the implementation of the OpenMRS system [11], an open-source medical records system. OpenMRS enables remote users to request access to medical records in a back-end database. To control access to the database, OpenMRS implements a reference monitor that mediates access to database operations on medical records. If we assume that any database operation may be a security-sensitive operation, we find that OpenMRS exhibits three types of ambiguities in its reference monitor implementation. First, OpenMRS performs some database operations that are not mediated by any authorization hook. While one may jump to the conclusion that such code patterns are vulnerable, the operation may be secure if: (1) no untrusted subject can actually request the operation; (2) all subjects would be authorized to perform the operation (i.e., no security checking is needed if all are authorized); or (3) these database resources are not security sensitive. Second, OpenMRS reuses mediation for one operation for other database operations. Whether such mediation is sufficient depends on whether the first authorization hook blocks all of the unauthorized subjects for the second operation. Third, OpenMRS may mediate two distinct operations using the same permission. As these two operations differ, one might assume that using the same permission would lead to a vulnerability, but this form of mediation is secure if the same set of subjects is always authorized for both operations.

If any of the requirements are violated in the ambiguous cases above, then there is an error in the access control enforcement that may be exploited. Such errors cannot be identified statically from source code [5, 7, 14–16, 18], as they depend on access policies at the deployment sites and the intentions of the policies that are not typically expressed explicitly [10]. As a result, provenance may be useful for enabling analysts to detect such errors, either as part of runtime testing of a deployment or as part of forensic analysis after an exploitation. However, current provenance systems [3, 6, 12] typically focus on operations on system objects, such as the database operations in the case of OpenMRS, so they would miss whether an error in access control was the original cause that allowed the exploit to occur. In this paper, we propose that programs that enforce access control should also track the provenance of their access control decisions to enable explanation to analysts when exploits occur to detect errors.

## 3. Design Overview

The goal of this work is to record the provenance of access control decisions to enable analytics to detect errors. Figure 1 shows the architecture of our proposed solution AccessProv. First, given a program, AccessProv computes the program statements that affect access control decisions and organizes statements related by control and data flow as *access dependence graphs* (ADGs). Second, AccessProv uses ADGs to instrument the program to record provenance events. When the program is run provenance events are saved in *access provenance graphs* (APGs). Third, analytics apply the ADGs to pick out the relevant events from the APGs to present to analysts to detect errors.

The first challenge is to compute how the various program statements that affect access control decisions are related. In general, subjects provide inputs to the program (e.g., an access request) that should be mediated before the program performs a security-sensitive operation. So, the statements associated with receiving inputs (e.g., system calls), mediation (e.g., authorization hooks), and security-sensitive operations will be identified as events impacting provenance. AccessProv uses a static analysis to collect such statements and the control-flow relationships among these statements into *access dependence graphs* (ADGs).

The second challenge is to determine what information to record at each provenance event to instrument the program to collect that information. The goal is to tell analysts about what is possible at each decision point to enable analysts to detect whether the access control implementation has considered the possible executions of the program. For example, rather than simply telling analysts the specific object accessed in a security-sensitive operation, analysts may want

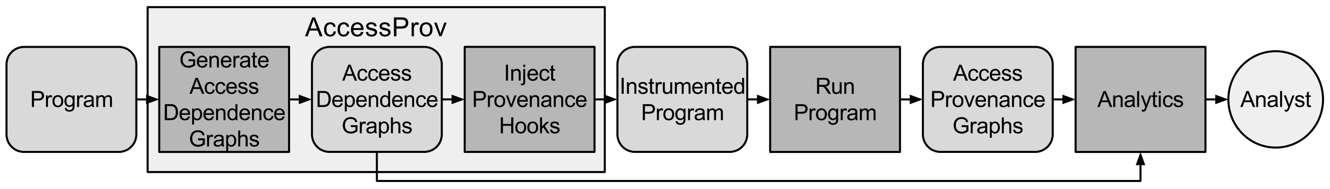

Figure 1: Design overview for proposed solution.

to know what objects may have been accessed at that operation to determine whether a hook is necessary to control that operation. ACCESSPROV logs program data flows, external inputs, and/or authorized objects to each object logged, to help the analysts as part of the *access provenance graphs* (APGs) collected by running instrumented programs.

The third challenge is to winnow the APG traces down to information relevant for analysts to detect errors. We propose for ACCESSPROV to apply the ADGs to match the events in the APG that are relevant to access control decisions that we want to compare. For example, when the program chooses no mediation for an operation, ACCESSPROV can collect all executions that run that operation to show analysts the subjects who had access to the operation (e.g., to determine whether an untrusted subject may perform the operation) and the objects they could access through that operation (e.g., to determine whether all untrusted subjects will be permitted access). In addition, ACCESSPROV can present ADGs for different operations that may be related, such as checking for the same permission, accessing objects of the same type, or accessing the same or intersecting sets of fields in the objects.

## 4. Design

In this section, we detail a design for the proposed approach outlined above.

### 4.1 Building Access Dependence Graphs

To instrument access control decisions, ACCESSPROV identifies and relates events upon which the decision may depend. We define an *access dependence graph* (ADG) as a graph $G = (V, E)$, where $v \in V$ are vertices that represent *events that impact access control decisions* and $E = (u, v)$ are directed edges that represent *program-flow dependence of event $v$ upon event $u$*. We identify two types of events of interest. First, there are events related to subjects requesting potentially security-sensitive operations. Subjects provide input, and the program performs operations that may be security-sensitive using those inputs. We want to capture both the input events (i.e., to identify subjects) and operation events (i.e., to identify objects and the impact of the operation). Second, there are events related to mediating security-sensitive operations. These correspond to the existing authorization hooks placed in the program.

The ADG edges show that their is both a control-flow dependence between the program statements and data-flow dependence between variables in those statements. Examples are shown in Figure 2, where operations lack mediation or are mediated differently. Similarly, Figure 3 shows the same two operations being authorized using different permissions in the authorization hook. In all cases, there is a control-flow dependence and data-flow dependence corresponding to the edges.

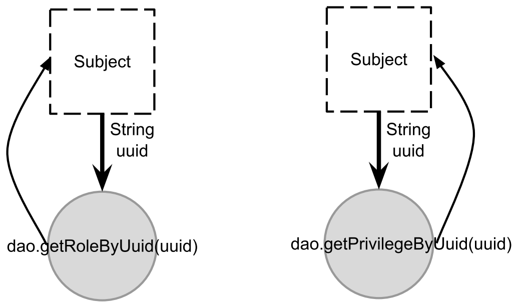

Figure 2: Access dependence graph example where operations are unmediated.

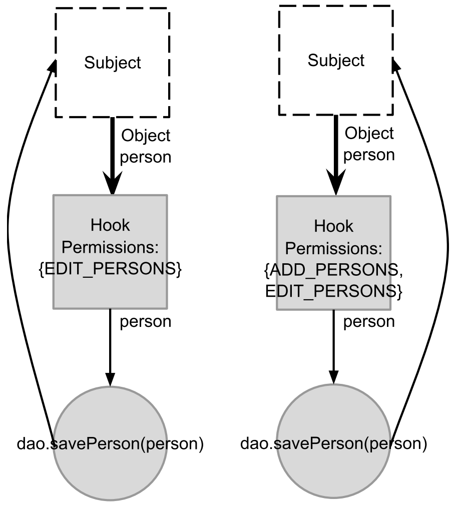

Figure 3: Access dependence graph example showing the same two operations being mediated using hooks checking for different permissions.

ACCESSPROV computes ADGs using static analysis. External inputs and authorization hooks can be identified syntactically, but identifying security-sensitive operations in programs has found to be a complex problem [4, 9, 14, 15, 18]. Researchers have proposed both syntactic methods, such as code patterns, and semantic methods, based on static taint tracking, to identify security-sensitive operations. Since there is no one solution, ACCESSPROV supports applying modules to identify security-sensitive operations using static analysis. In our approach, applied to OpenMRS [11], we identify database operations as security-sensitive operations [15] and identify operations on objects returned from the database as security-sensitive operations [9].

### 4.2 Instrumenting Programs Using ADGs

Once ACCESSPROV computes ADGs for programs, it is clear *where* to put instrumentation to collect provenance regarding enforcement of access control, but we still have to define *what* information ACCESSPROV should collect. The aim is to help analysts determine the permissions (i.e., objects and accesses to those objects) available to subjects authorized to perform a security-sensitive operation.

A key question is what objects may be accessible to an authorized subject. If there is an authorization hook, then the requested object is accessible to all authorized subjects, which can be determined from the access control policy directly. However, if the object is not directly authorized, then there is a question which objects may be accessible to subjects. For example, Figure 4 shows a case where the `userid` object is authorized, but others that are dependent on it are not. The analyst may not be able to determine the values of other objects, so they cannot determine whether someone authorized to get user information is always authorized to access the `loginCredential` of that user as shown in Figure 4.

To solve this problem, we propose that ACCESSPROV records the source of unmediated objects. To do this, ACCESSPROV computes a static reverse taint slice for all unmediated objects. Often, an unmediated object is extracted from a field of another object that may itself be mediated. Nonetheless, the possible values of the unmediated object may vary depending on how the field's values may be modified. By presenting the reverse slice, an analyst can learn where the unmediated object value was set.

### 4.3 Analyzing Access Provenance Graphs

In the last step, ACCESSPROV aids analysts in assessing the data collected at runtime. There are two aims of access control provenance. One is to learn how a particular object is accessed in authorized (or unmediated) operations to detect possible misuse of authorized access. The second is to examine and compare authorized operations to detect errors.

An *access provenance graph* (APG) is a graph $G_p = (V_p, E_p)$, where $V_p$ are recorded events of the type identified in the ADG above and $E_p = E_t \cup E_o$ where $E_t$ are edges linking each event with the preceding and following events to form an *event trace* and $E_o$ are edges that link events referring to events that access the same object $o$ to form an *object trace*. Object traces allow analysts to see how a particular object of interest was accessed to assess how authorized (or unmediated) operations may have impacted it.

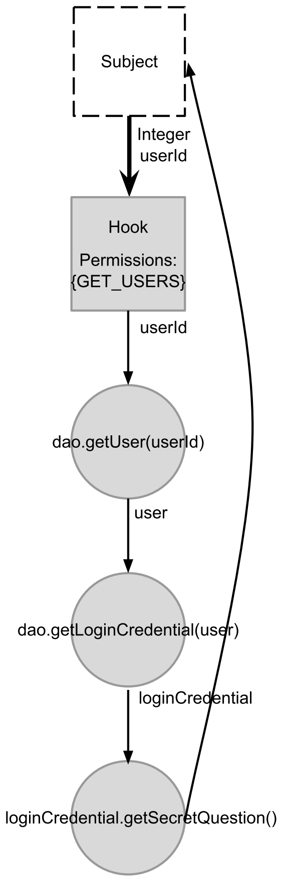

Figure 4: Access flow graph example depicting partial mediation where the first security sensitive operation has an associated hook, but the second and third operation do not.

To detect errors, we propose to leverage the ADGs to identify the events relevant to a particular scenario to compare to other scenarios. Thus, ACCESSPROV records unique identifiers for ADGs and their events with events when they are recorded in the APG to enable these events to be found. Thus, analysts may compare all of the events related to a particular ADG (e.g., to assess the implications of an unmediated operation, see Figure 2) and compare events related to comparable ADGs (e.g., to compare operations mediated by different hooks, see Figure 3).

## 5. Implementation

ACCESSPROV was implemented on top of Soot [17], an open-source Java bytecode analysis framework that was originally developed in the Sable Research Group at McGill University. Soot's intermediate representation *Jimple* is easy to analyze and modify making it a suitable choice for our analysis and injecting provenance hooks into bytecode.

In order to generate access dependence graphs, we need to identify security-sensitive operations within the program. We achieve this by first building a program dependence graph (PDG) to capture all of the control and data flows within the program. Using our PDG we compute a forward taint analysis to track user input to identify database operations accessible to adversaries and the objects retrieved from the database as security-sensitive operations. Once these security-sensitive operations are identified, we use a reverse-taint analysis to track the data dependencies for arguments to database operations to find possible sources of those inputs.

Provenance hooks are injected into the bytecode by adding our own *Jimple* instructions to the necessary locations in the bytecode. Our additions include a public Java class that manages an internal buffer to stage the writing provenance information. Thus, any class that imports our buffer management class will have the necessary ability to record provenance data. Then, at every security-sensitive operation in the program ACCESSPROV injects a set of instructions. First, ACCESSPROV adds an import for our buffer management class if one does not exist. Secondly, ACCESSPROV initializes a string variable with the provenance data to be recorded. Finally, ACCESSPROV initializes a buffer management variable and call the *addProvenance()* method passing along the newly created string. The *addProvenance()* method internally checks the size of the buffer and automatically writes information to a log file when full.

Once our hooks are in place throughout the application we need to write the newly modified bytecode out to corresponding *.class* files. ACCESSPROV performs this step by using *Jasmin*, which is an assembler for the JVM. Jasmin comes with Soot and will automatically write all of our changes to generate class files that are suitable for execution and now provenance aware.

## 6. Evaluation

In this section we evaluate ACCESSPROV by analyzing OpenMRS, an open source medical record system. OpenMRS is used widely around the world. According to their latest annual report [11] OpenMRS is used in roughly 80 countries, 1200 clinics, and manages approximately 5 million patients' data. Below, we examine three possible problems with the access control implementation of OpenMRS. Note that while static analysis can identify these as issues, analysts may not recognize problems until they see access control actions or even perform forensic analysis.

### 6.1 No Mediation

One of our initial questions when analyzing OpenMRS was whether there are operations we consider to be security-sensitive, but the developers either felt were not security-sensitive or forgot to place an authorization hook. In order to identify these cases, ACCESSPROV identifies ADGs with unmediated operations. Figure 2 shows two such ADGs; the first shows an operation *getRoleByUuid()*, and the second shows an operation *getPrivilegeByUuid()*, both of which are not mediated. ACCESSPROV uses the ADGs' identifiers and associated events to retrieve 2 instances, for each operation, in the APG trace that depict these operations being executed by a subject that is not mediated. We show these cases in Figure 5. UUIDs are not treated as secret values necessarily, so an unauthenticated user may be able to extract the OpenMRS access policy by repeatedly querying roles and privileges for known or guessable UUIDs to find a user with permissions to exploit.

### 6.2 Consistency

In order to identify whether the developers are consistent with their use of permissions for authorizing operations, ACCESSPROV can retrieve ADGs that check for a particular related permissions to compare the operations performed on particular objects.

Within OpenMRS there are *patients*, *users*, and *persons* objects, where a *person* can be either a *patient* or a *user*. Examining the ADGs, we find that hooks checking for *EDIT_PERSONS* permissions permit a subject with this permission to save a *person* to the database as well as void (delete) a *person* from the database. Figure 3 shows the ADGs for both of these operations. Using the ADGs, we retrieve 4 instances from the APG trace where an authorized subject voids a person from the database and 6 instances where an authorized subject saves a person to the database.

By comparing these traces to those of other ADGs with comparable hooks for the *EDIT_PATIENTS* and *EDIT_USERS* permissions, we see that a subject cannot void a patient or user with these permissions. OpenMRS developers separated the operations to void a patient or user with *DELETE_PATIENT* and *DELETE_USER* permissions respectively, but neglected to do the same for *persons*. Showing the analyst the difference between these ADGs in the traces highlights the inconsistency, which might lead to a user being given permissions to edit *persons* without realizing they are also giving them permissions to delete *persons*.

### 6.3 Partial Mediation

The ADG reveals an interesting case that is depicted in Figure 4. In this case, a subject that is authorized to get users from the database (e.g the subject has the *GET_USERS* permission) is capable of querying that user's login credentials (including that user's secret question) without additional authorization. The test suite did not run this operation directly, so we modified the test suite to perform the operation as shown in Figure 7.

Using ADGs, we can compare this ADG to other ADGs that operate on users' login credentials to compare the authorizations. We found another case that retrieves the login credentials from a user object in the same way (i.e., using the reverse slice described in Section 4.2) that requires an *EDIT_USER_PASSWORDS* permission, while the operation *getSecretQuestion(User user)* does not. Similar to the section above we can imagine a scenario where a user is assigned permissions to get users from the database, possibly to make benign modifications, but now has the capability to leak sensitive information regarding that user's secret question.

### 6.4 Performance

OpenMRS is a mature code base consisting of approximately 110,000 source lines of code. Even though our analysis is written in Java, ACCESSPROV is able to analyze OpenMRS and generate a program dependence graph in roughly 1 minute. Identifying security-sensitive operations as well as injecting provenance hooks takes an additional 1-2 minutes.

Typically, provenance collection taxes a program with overhead for computing the runtime information while the program is executing. Our method aims to compute a majority of the information statically. Leaving a small amount of information (e.g input variable values, operation argument values, etc), that cannot be computed statically, to be gathered from the runtime traces using our hooks. When running OpenMRS's test suite, which performs 3,380 tests, we notice a 2.1% overhead in performance, which generates 2.6MB of provenance data (APGs).

## 7. Related Work

Automated provenance collection mechanisms can be implemented at various layers of a system. Pohly et al. [12] presented a kernel-level system leveraging [3] to capture high fidelity whole-system provenance. This method creates a provenance-aware environment that applications can run on top of. Although applications executed in this environment are automatically provenance aware, the metadata collected is at too low of a level to reason about the access control mechanisms of a specific application. SPADE [6] is a software infrastructure that allows you to collect and manage provenance data. To date the most accurate method of capturing application level provenance data is to invest in manual provenance instrumentation within the application itself and supplement the data with libraries such as ProvToolBox [8] and systems described above. Unfortunately, this approach is expensive and time consuming and has motivated us to develop a tool that can automatically make an application provenance-aware.

Muthukumaran et al. [9] presents a method for automatically identifying authorization hook placements within C code, while [10] extend this work and propose algorithms to minimize the placement to enforce a specified access control policy. [9] also presents a method for identifying security-sensitive operations in C code by identifying containers, but this cannot be directly applied to Java as sensitive operations are typically stored as fields of Java objects or as database operations in many web-based applications.

Retrospective security methods such as [1] propose provably correct code re-writing algorithms utilizing formal logging specifications. They developed their tool specifically for OpenMRS. Additionally, [13] implement a relationship based access control mechanism for OpenMRS to be used in place of the current role based access control mechanism. Although providing additional constraints on who can access certain information, this approach does not solve the access control problem present in applications like OpenMRS. This in part motivates the need for a provenance based approach to log access control information to understand the implications of currently implemented access control systems.

## 8. Conclusion

In this paper, we presented ACCESSPROV, a Java bytecode analysis tool capable of retrofitting legacy Java applications with provenance hooks regarding access control decisions. ACCESSPROV statically generates access dependence graphs, which are used to inject provenance hooks that will record access control decisions, in the form of access provenance graphs, for retrospective analysis. We evaluate ACCESSPROV on OpenMRS an open source medical record system and show that our injected hooks contribute low performance overhead when running OpenMRS's test suite. We also identify multiple cases within OpenMRS that are either missing mediation, contain partial mediation, or are inconsistent with access control in other areas of the program.

## References

[1] S. Amir-Mohammadian, S. Chong, and C. Skalka. Correct audit logging: Theory and practice. In *International Conference on Principles of Security and Trust*, pages 139–162. Springer, 2016.

[2] J. P. Anderson. Computer security technology planning study, volume II. Technical Report ESD-TR-73-51, HQ Electronics Systems Division (AFSC), October 1972.

[3] A. Bates, D. J. Tian, K. R. Butler, and T. Moyer. Trustworthy whole-system provenance for the linux kernel. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 319–334, Washington, D.C., Aug. 2015. USENIX Association.

[4] V. Ganapathy, T. Jaeger, and S. Jha. Automatic placement of authorization hooks in the Linux Security Modules framework. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 330–339, Nov. 2005.

[5] V. Ganapathy, D. King, T. Jaeger, and S. Jha. Mining security-sensitive operations in legacy code using concept analysis. In

*Proceedings of the 29th International Conference on Software Engineering (ICSE)*, May 2007.

[6] A. Gehani and D. Tariq. Spade: Support for provenance auditing in distributed environments. In P. Narasimhan and P. Triantafillou, editors, *Middleware 2012*, volume 7662 of *Lecture Notes in Computer Science*, pages 101–120. Springer Berlin Heidelberg, 2012.

[7] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee. Merlin: specification inference for explicit information flow problems. *ACM Sigplan Notices*, 44(6):75–86, 2009.

[8] L. Moreau, T. D. Huynh, M. Jewell, A. S. Keshavarz, J. A. Hussein, and D. Michaelides. Provtoolbox, 2014.

[9] D. Muthukumaran, T. Jaeger, and V. Ganapathy. Leveraging "choice" to automate authorization hook placement. In *CCS'12: Proceedings of the 19$^{th}$ ACM Conference on Computer and Communications Security*, page TBD. ACM Press, October 2012.

[10] D. Muthukumaran, N. Talele, T. Jaeger, and G. Tan. Producing hook placements to enforce expected access control policies. In *International Symposium on Engineering Secure Software and Systems*, pages 178–195. Springer, 2015.

[11] Openmrs. `http://openmrs.org/`.

[12] D. J. Pohly, S. McLaughlin, P. McDaniel, and K. Butler. Hi-fi: Collecting high-fidelity whole-system provenance. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 259–268, New York, NY, USA, 2012. ACM.

[13] S. Z. R. Rizvi, P. W. Fong, J. Crampton, and J. Sellwood. Relationship-based access control for an open-source medical records system. In *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*, pages 113–124. ACM, 2015.

[14] S. Son, K. S. McKinley, and V. Shmatikov. Rolecast: finding missing security checks when you do not know what checks are. *ACM SIGPLAN Notices*, 46(10):1069–1084, 2011.

[15] S. Son, K. S. McKinley, and V. Shmatikov. Fix Me Up: Repairing Access-Control Bugs in Web Applications. In *ISOC Network and Distributed System Security Symposium (NDSS)*, 2013.

[16] G. Tan and J. Croft. An empirical security study of the native code in the jdk. In *Usenix Security Symposium*, pages 365–378, 2008.

[17] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot-a java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*, page 13. IBM Press, 1999.

[18] X. Zhang, A. Edwards, and T. Jaeger. Using CQUAL for static analysis of authorization hook placement. In *Proceedings of the 11th USENIX Security Symposium*, pages 33–48, August 2002.

## Appendix A: Access Provenance Graphs

In this section we display our APGs that were identified from the runtime traces generated using OpenMRS's test suite. These APGs are presented to the analyst to aid in understanding potential errors in the access control and present information that could not be identified statically.

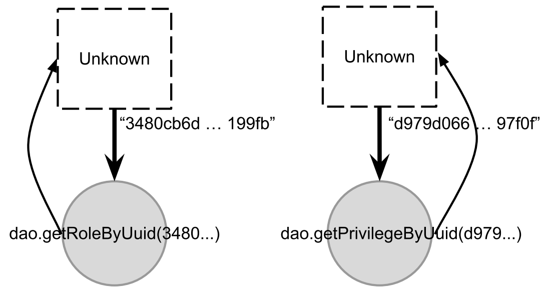

Figure 5: Access provenance graph example where operations are unmediated.

Figure 5 depicts two APGs. Both are cases where a subject *Unknown* performs an operation that is not mediated. In both of these cases we can see the unique id that is used to query the database for a specific role or privilege respectively.

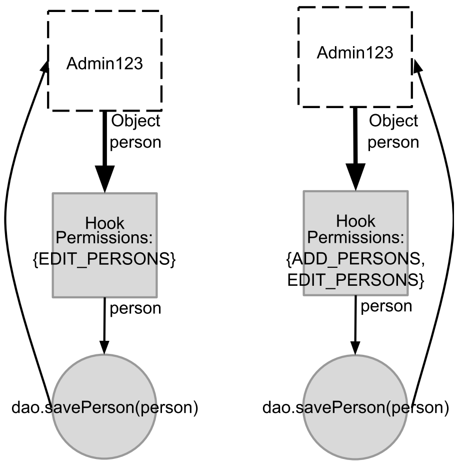

Figure 6: Access provenance graph example showing the same two operations being mediated using hooks checking for different permissions.

In Figure 6 we see two cases where the same subject *Admin123* is performing two different operations. The first is to void a person from the database, while the second is modifying an existing person in the database. We can see that the hooks require the same sets of permissions.

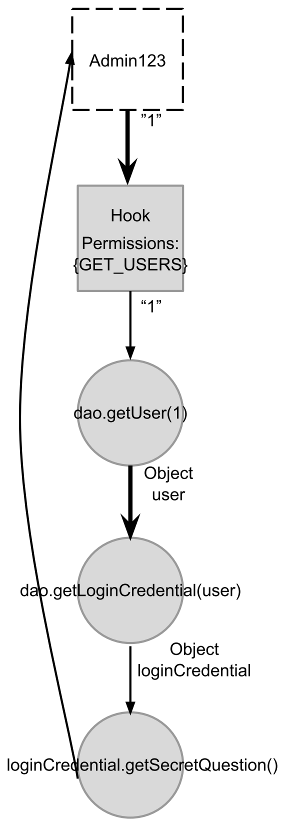
Admin123
"1"
Hook Permissions: {GET_USERS}
"1"
dao.getUser(1)
Object user
dao.getLoginCredential(user)
Object loginCredential
loginCredential.getSecretQuestion()

Figure 7: Access provenance graph example depicting partial mediation where the first security sensitive operation has an associated hook, but the second and third operation do not.

Figure 7 depicts a single case where a subject *Admin123* uses the userid *1* to query for a user from the database and subsequently uses that user to gather that user's login credentials, including their secret question.