Is Modeling Access Control Worth It?

David Basin
basin@inf.ethz.ch
ETH Zürich
Zurich, Switzerland

Juan Guarnizo
juan.guarnizo@inf.ethz.ch
ETH Zürich
Zurich, Switzerland

Srđan Krstić
srdan.krstic@inf.ethz.ch
ETH Zürich
Zurich, Switzerland

Hoang Nguyen
hoang.nguyen@inf.ethz.ch
ETH Zürich
Zurich, Switzerland

Martín Ochoa
ocho@zhaw.ch
Zurich University of Applied Sciences
Zurich, Switzerland

ABSTRACT

Implementing access control policies is an error-prone task that can have severe consequences for the security of software applications. Model-driven approaches have been proposed in the literature and associated tools have been developed with the goal of reducing the complexity of this task and helping developers to produce secure software efficiently. Nevertheless, there is a lack of empirical data supporting the advantages of model-driven security approaches over code-centric approaches, which are the de-facto industry standard for software development.

In this work, we compare the result of implementing the same functional and security requirements by multiple developer groups in the context of a security engineering graduate course. We thereby obtain evidence on the security and efficiency of a tool-based model-driven approach to security from the literature compared to a direct implementation in a well-known, modern web-development framework. For example, the projects using model-driven development pass up to 50% more security tests on average with less development effort. Also, we observe that models are twice as concise as manual implementations, which improves system maintainability.

CCS CONCEPTS

• **Software and its engineering → System modeling languages**; • **Security and privacy → Software security engineering**; **Access control**.

KEYWORDS

access control, security engineering, modeling languages

ACM Reference Format:
David Basin, Juan Guarnizo, Srđan Krstić, Hoang Nguyen, and Martín Ochoa. 2023. Is Modeling Access Control Worth It?. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS '23), November 26–30, 2023, Copenhagen, Denmark.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3576915.3623196

CCS '23, November 26–30, 2023, Copenhagen, Denmark.
© 2023 Association for Computing Machinery.
ACM ISBN 979-8-4007-0050-7/23/11…$15.00
https://doi.org/10.1145/3576915.3623196

1 INTRODUCTION

Developing secure web applications is difficult and error-prone. In particular, the incorrect implementation or configuration of access control is the most critical web application security risk according to the OWASP Top 10 report for 2021 [26]. For example, in 2019 the financial services company giant First American had a data breach that put 885 million sensitive customer records at risk because the company did not have proper access control [37]. A logged-in user could access other customers' information simply by changing one value in the URL. The leaked data included bank account numbers, social security numbers, driver's license information, and more.

Abstractly, access control determines whether actions (like those for creating, reading, updating, and deleting data) are authorized and prevents unauthorized actions. Typically, applications determine which actions are authorized based on contextual information, like the data affected by the action, or data about the user carrying out the action. A special case is *role-based access control* (RBAC) where only the user's role is considered. But, more generally, access control may be *programmatic* or *fine-grained* (FGAC) where the authorization decision may depend on arbitrary parts of the application's current state. Ideally, the application should enforce the *least privilege* and *complete mediation* principles [30, 31]. The former requires that each user only performs actions that are necessary to carry out their business role within the application's domain. The latter requires that every action is checked for authority.

In practice, these two seemingly simple principles are difficult to implement in a code-centric way as the implementation process is error-prone and does not scale with the application size. Many development frameworks offer libraries [1, 3, 20, 22, 23, 25] that can simplify the implementation of FGAC, but they still require non-trivial effort to use, maintain, and evolve. Alternatively, developers can use a model-driven security (MDS) methodology and tools [5, 6, 8] that tightly couple the system implementation with FGAC policies defined using design models that integrate security into the design process. These design models resemble typical abstract access control models: they are simple, concise, and expressive. They also have formal semantics that supports automatically checking least privilege using model checking techniques [4]. Moreover, MDS tools can automatically generate correct-by-design implementation of the authorization decisions based on the models and they can instrument every application's action to achieve complete mediation.

In this paper, we investigate why developers struggle to implement access control correctly and whether the main cause is the complexity resulting from the low abstraction level of existing libraries combined with the need for programmatic access control. Moreover, we investigate whether using higher-level abstractions, namely those provided by MDS, would significantly improve correctness and reduce development effort. To answer these questions, we have conducted the first empirical study of developers' behavior when tasked with implementing access control for a simple web application, both directly and using MDS. We collect and analyze quantitative and qualitative data from both types of development, such as the quality of the implementation and the experience of the developers, in order to derive and support our insights.

This study was conducted as part of a project in a graduate security engineering course offered by the authors. The course covers software engineering principles for building secure systems. In the course project, students are asked to develop a secure web application twice: once using an MDS tool called ActionGUI [5], and once by manual programming in Python (Section 2). Both alternatives have an educational value as students learn to implement authorization at two very different abstraction levels. We base our study on computer science students from a software engineering course as they are an acceptable proxy for developers [17, 29, 33].

Our study (Section 3) considers each student's submission consisting of the two alternative implementations of the secure web application (Section 4). Projects are assessed using more than 1,000 predefined unit tests that evaluate the correctness of the access control policies. In particular, we design unit tests to exhaustively exercise both authorized and unauthorized actions within the application. Additionally, the students are asked to complete an anonymous survey where they provide their subjective assessment of the two alternative approaches' difficulty and effectiveness.

Our contributions and key insights (Section 5) are:

- We carry out and report on the first empirical study on MDS, and more generally on implementing access control, and we compare it against the current state of practice.
- Modeling access control policies and relying on MDS result in 50% fewer failed tests in the resulting applications. This improvement additionally carries over to application maintenance and evolution.
- Complete mediation provided by model transformations is crucial in improving security, while least privilege improves only marginally with manual model inspection. We hypothesize that automatic model analysis would improve the enforcement of least privilege.
- Modeling access control policies before *manually* implementing them in code provides only limited benefit. It is important to use MDS tools to *automatically* generate configuration, code, and other artifacts from the models.
- The effort required for using each technique is similar. However, models are twice as concise as the manual implementation, which improves maintainability.

Finally, as this is the first empirical study on using MDS for access control (Section 6), we hope that our relatively small, yet challenging, web application case study can serve as a benchmark for other studies on access control implementation and configuration. All the material used to conduct this study is provided in our publicly available artifact [7].

2 PRELIMINARIES

System design is an often neglected aspect in system development, although a detailed design is critical in bridging the gap between (high-level) system requirements and (low-level) implementations. In the design phase, system models are built and later refined into a system implementation. These models remain useful artifacts that connect the requirements and the implementation. Model-driven security (MDS) [6] is a model-driven development [2, 9, 19] methodology that connects the system implementation with *security requirements* via design models that support security. As the model semantics are defined in terms of the allowed system executions, MDS can use semantic-preserving model transformations to generate code that checks every system action (i.e., achieves *complete mediation*) and prevent system executions disallowed by the modeled security requirements.

In particular, MDS focuses on a specific class of security requirements: fine-grained access control (FGAC) policies that base authorization decisions on the application's current state. In MDS, a software engineer first creates a design model in a design modeling language (e.g., a UML class diagram). Next, a security engineer creates a security model in a security modeling language (e.g., like SecureUML [21]). Via model transformations, a secure-by-design implementation is obtained from the models.

ActionGUI (AG) [5] is a tool that supports the MDS methodology. It implements a model transformation that takes data, security, and GUI models as input and produces a fully functional web application with a configured enforcement mechanism for the FGAC policies specified by the security model. The goal of the design process is to obtain a security model that satisfies the principle of *least privilege*: users are granted only the permissions necessary for them to carry out their business role within the application.

A *data model* in AG is used to define the structure of the well-formed states of the system. It is a simplified UML class diagram, i.e., a set of classes C that can be related by binary associations $A \subseteq C^2$ and may have attributes $attr(c)$, for $c \in C$. Each association $a \in A$ is binary: it has two association-ends el and er connecting two classes. Each association a declares both association-ends $end(a) = \{(el, ml, cl), (er, mr, cr)\}$ with their multiplicities ml, mr (1 or *), and (possibly) uniqueness constraints cl, cr (Set or Bag).

Example 2.1. Consider the UML class diagram drawn in black in part of Figure 1 on page 5. The corresponding AG data model has three classes $C = \{\texttt{Person}, \texttt{Event}, \texttt{Category}\}$ and seven associations in A. All associations' multiplicities are unconstrained, except for the one between the `Person` and `Event` classes: each instance of an `Event` can be associated with at most one `Person` instance. All association ends with multiplicity $*$ are considered as sets by default.

Given a data model, the set of actions $\mathcal{A}$ that a user may take is

$$\begin{aligned}&\{\mathsf{create}(c), \mathsf{delete}(c) \mid c \in C\} \cup \\ &\{\mathsf{read}(c,n), \mathsf{add}(c,n), \mathsf{remove}(c,n) \mid (n, *, \_) \in end(a),\ a \in A\} \cup \\ &\{\mathsf{read}(c,n), \mathsf{update}(c,n) \mid n \in attr(c),\ c \in C,\ \text{ or} \\ &\qquad\qquad\qquad\qquad\qquad (n, 1, \_) \in end(a),\ a \in A\}.\end{aligned}$$

For example, a user may create an `Event` object (`create(Event)` action), set its name (`update(Event, name)` action), and add it to a category (`add(Event, categories)` action).

The *object constraint language* (OCL) [15] is a first-order logic used to define properties of data models. OCL syntax has a predefined part (e.g., integer ($\mathbb{Z}$) addition, string (String) concatenation, and collection functions like **forall**, **select**, **size**, etc), and a variable part that depends on the given data model. Every OCL expression evaluates to a value of some type. We denote with θ_t all OCL expressions that evaluate to instances of type t. OCL expressions that evaluate to Booleans ($\theta_{\mathbb{B}}$) are called OCL constraints.

OCL's dot-operator is used to access an object's attribute (e.g., p.name), or an association end, i.e., the collection of objects linked with the object via an association (e.g., p.events). For calling collection functions, an arrow notation is used (e.g., p.events→**size**()). When used with collections, the dot-operator maps over them. OCL expressions can be written in the context of an object, and such an object can be referred to using the keyword **self**.

Example 2.2. If the OCL constraint **self**.managedBy→**forall**(m | **self**.`attendants`→**includes**(m)) is given as an invariant of the `Event` class, it would restrict the possible objects of the `Event` class to those that always have all its managers as attendants.

The set of free variables of an OCL expression ϕ is fv(ϕ), e.g., fv(**self**.events→**forall**(b|a.concat(b.`title`).size() < c)) = $\{a_{\text{String}}, c_{\mathbb{Z}}\}$. Here the variable b_{Event} is bound by the **forall** function. Note that we subscript variables with their types for clarity.

A *security model* defines which actions are authorized to be carried out by users in different roles, i.e., an FGAC policy. Let $\mathcal{R}$ be a set of roles and $\preceq_{\mathcal{R}}$ a partial order on these roles representing the role hierarchy on $\mathcal{R}$. Intuitively, if $r_1 \preceq_{\mathcal{R}} r_2$, then r_2 has all of r_1's permissions. The set $\mathcal{R}$ can be modeled as an enumeration type.

Example 2.3. Suppose an application has the roles $\mathcal{R}$ = {Freeuser, Premiumuser, Moderator, Admin} and a role hierarchy Freeuser $\preceq_{\mathcal{R}}$ Premiumuser $\preceq_{\mathcal{R}}$ Moderator, and Premiumuser $\preceq_{\mathcal{R}}$ Admin. Then, for example, the Admin role has all the permissions of the Freeuser role, whereas it is incomparable with the Moderator role.

AG assumes that a data model has a class c representing the application's users with role $\in$ $attr(c)$ with (enumeration) type $\mathcal{R}$ (e.g., Person class in Figure 1).

Authorization constraints are OCL constraints that express conditions under which a user (e.g., Alice) in some role (e.g., Freeuser) may carry out some action (e.g., update(`Event`, `title`)). Such a constraint is written in the context of the object representing the resource; hence **self** evaluates to that object (e.g., an instance of the `Event` class). The user object is represented as a free variable caller in the constraint. Depending on the action, other factors may also be relevant when making the authorization decision. For example, when updating an attribute (e.g., `title`), its new value can influence the decision, which is captured by the free variable value. Given a valuation for the variables, the OCL constraint evaluates to a Boolean that encodes whether the user can carry out the action.

Given a data model, let Constr(V) = $\{e \mid e \in \theta_{\mathbb{B}}, \text{fv}(e) \subseteq V\}$ be the set of OCL constraints containing some of the free variables in V. Given an action $a \in \mathcal{A}$, the set of authorization constraints $C(a)$ is:

$$C(a) = \begin{cases} \text{Constr}(\{\text{caller}\}), & \text{if } a \text{ is a create, delete, or read} \\ \text{Constr}(\{\text{caller}, \text{value}\}), & \text{if } a \text{ is an update} \\ \text{Constr}(\{\text{caller}, \text{target}\}), & \text{if } a \text{ is an add or remove.} \end{cases}$$

The free variable target represents the object that is added to or removed from **self**'s association end.

In addition to roles and their hierarchy, AG's security model defines a relation $\mathcal{PA} \subseteq \bigcup_{a\in\mathcal{A}} \mathcal{R} \times \{a\} \times C(a)$ representing a set of FGAC permissions. In practice, AG defines this relation hierarchically starting from roles, resources, actions, and finally authorization constraints. Intuitively, an action by a user is allowed if it is associated with a permission containing an authorization constraint satisfied on the current instance of the data model, and the user's role is greater than or equal to the role in the permission.

Example 2.4. Consider the data model from Example 2.1 and the roles from Example 2.3. The `Person` class represents the application's users. Suppose one wants to model the security requirement: *All authenticated users can view the list of attendants of a public event. Only attendants of a private event have access to this information.*

The requirement can be expressed as the FGAC permission (Freeuser, read(Event, `attendants`), ϕ) $\in \mathcal{PA}$ where ϕ is the OCL constraint: not **self**.`private` or **self**.`attendants`→**includes**(caller). The permission for a Freeuser role is sufficient as other roles inherit its permissions. In AG's syntax the permission is written as follows:

```
role FREEUSER {
  Event {
    read(attendants) constrainedBy [not self.private
        or self.attendants->includes(caller)]
    //other FREEUSER permissions on the Event resource
  }
  //other FREEUSER permissions
}
//other roles' permissions
```

A *GUI model* defines the widgets contained in each page of the web application (e.g., labels, text boxes, buttons, tables, and more) and their event handlers' behavior (e.g., the functionality executed on a button click). The behavior can be specified using simple control-flow structures (like assignments, if-then-else statements, and loops) and actions from $\mathcal{A}$.

AG's model transformations generate a Java web application in the Vaadin framework [35]. Intuitively, every action from $\mathcal{A}$ is wrapped in an if-then-else statement that checks if the action is allowed based on the current user and application state. In this way AG achieves complete mediation by design.

Flask [12] is a Python library for developing web applications. It binds URLs to Python functions (called endpoints) that implement the functionality that the web application provides when a user visits a URL. Flask applications typically rely on other libraries: SQLAlchemy [32] for managing persistent state in a database, and Flask-User [34] for authentication. Whenever a Flask endpoint is called by an authenticated user, the Flask-User library initializes the object called `current_user` with the instance of that user. Flask applications can also use existing authorization libraries, such as Flask-OSO [23], Flask-Principal [25], Flask-Authorize [3].

3 STUDY DESIGN

We now define the scope and the goals of our study. We first outline our main research questions (Section 3.1). We then describe the project's design and timeline (Section 3.2). The project is implemented by students as a course project within a security engineering course and we also describe the survey that we conduct after the project (Section 3.3). Along the way, we explain how the project and the survey data enable us to answer the research questions.

3.1 Research Questions

In this paper we focus solely on access control, which includes the process of implementing authorization decisions and enforcing them in a system. Secure coding aspects of building secure systems, like input sanitization or control-flow integrity, are out of our study's scope. Furthermore, as authorization decisions depend on the current user's attributes, we assume the existence of an appropriate authentication mechanism. The mechanism should correctly associate each user of the application with an object representing the user within the application's data model.

Hereinafter, security requirements refer to access control policies. During system development, it is common to distinguish between the design and implementation phases. We assume that the modeling of system requirements happens during the design phase, whereas their implementation happens during the implementation phase. When developing systems using the MDS methodology, no implementation effort is needed to meet security requirements as, given the design, the relevant parts of the implementation are generated directly by model transformations. In manual code-centric approaches the design phase is skipped and all system requirements are addressed in the implementation phase.

To compare the model-driven and code-centric approaches for implementing access control, we aim to answer the following research questions:

RQ1 Can, and to what extent, modeling access control improve the enforcement of least privilege?
RQ2 Can, and to what extent, modeling access control help achieve complete mediation?
RQ3 What is the effort needed to implement, fix, and evolve security in terms of lines of code?
RQ4 What is the effort needed to implement, fix, and evolve security in terms of time?
RQ5 Is implementing (resp., designing) access control more effective given a design (resp., reference implementation)?
RQ6 What are the most challenging aspects of access control to specify (resp., implement)?

3.2 Project Timeline and Format

Table 1 presents the timeline of the course project, where each row represents a project phase, and the columns describe the input initially provided, the final deliverable to submit, and the phase's duration. Within the course project, students were tasked with securing (i.e., implementing access control for) a case study web application (see Section 4.1) using both AG and Flask. The project consists of four consecutive phases, where Phases I and II each last two weeks, and Phases III and IV are each one week long. The project is carried out individually by each student.

In Phase I, the students were divided arbitrarily into two groups, A and B. Group A was tasked with securing the AG application, whereas group B was tasked with securing the Flask application. In Phase II, the two groups switched roles, with Group A securing the Flask application and Group B securing the AG application. The reasons for dividing the students into groups is twofold. First, we wanted to ensure that we address all students' questions about the project description (i.e., requirements engineering projects), simultaneously for both versions of the application in Phase I. Otherwise, if all students use a single approach in this phase, some requirements may diverge across the two implementations, making them incomparable. Second, we aimed to answer RQ1 and RQ2, i.e., if, and to what extent, modeling access control can improve enforcement of least privilege and complete mediation.

Table 1: Project Schedule

	Input	Deliverable	Duration
Phase I (Group A)	*Requirements* *TemplateAG*	`AG security model`	2 weeks
Phase I (Group B)	*Requirements* *TemplateFlask*	`project.py`, (`*.py`)	2 weeks
Phase II (Group A)	*TemplateFlask*	`project.py`, (`*.py`)	2 weeks
Phase II (Group B)	*TemplateAG*	`AG security model`	2 weeks
Phase III (All groups)	*TestsAG* *TestsFlask*	`AG security model` `project.py`, (`*.py`)	1 week
Phase IV (All groups)	*ChangeRequest* *TemplateUpdates*	All files	1 week
Survey (All groups)	*Questionnaire*	Anonymous answers	8 days

Both groups initially received the same *Requirements* document containing the application's functional and security requirements, along with the respective ActionGUI (*TemplateAG*) and Flask (*TemplatePython*) initial code (hereinafter called templates) implementing only the application's functional requirements. The two templates were designed such that the students only need to modify a pre-defined number of files. The students need to modify and submit only the AG security model. The security model provided within *TemplateAG* is trivial as it permits all actions. In the case of Flask, the students only need to modify and submit a file called `project.py`. This file initially contains all the Flask endpoints implementing the application's functional requirements. If the students decide to factor out some of their code into separate `.py` files, they must also submit these files.

The students were tasked with eliciting and analyzing the security requirements from the document and then modifying the respective templates to implement these requirements. In Phase II, the groups received the remaining templates and implemented the other version of the application. After Phase II we can answer RQ5, i.e., we assess the impact of the prior experience in implementing (resp. designing) access control in Phase I on the design (resp. implementation) in Phase II.

In Phase III, we provide test cases designed to exercise the security requirements in both applications (*TestsAG* and *TestsFlask*). Each student was required to study the test cases and fix both of their implementations to pass as many tests as possible. By analyzing which tests failed most frequently across all submissions in each phase, we can determine which aspects of access control were most difficult to implement correctly (RQ6).

Finally, in Phase IV we release the final inputs for the project: new functional and security requirements are given in the *ChangeRequest* document. We also provide the students updates to both the AG and Flask templates (*TemplateUpdates* patch) to implement the functional requirements. The students are again asked to elicit and analyze the security requirements and implement them in both of their applications. The *TemplateUpdates* patch implements functional requirements by only modifying the parts of the templates that the students did not need to submit in earlier phases, so the new changes in Phase IV can just focus on the security requirements.

We analyze difference in the lines of code between the submitted deliverables across the phases to determine the effort needed to implement, fix, and evolve security in applications (RQ3).

All the inputs from Table 1 provided to the students are publicly available in our artifact [7].

3.3 Survey

As the final part of the study, we conducted an anonymous survey to gather data on the experiences and opinions of the participants, as well as the time needed for each project phase. The main goal of this survey is to answer the research questions related to the time efficiency of the two approaches for implementing, fixing, and evolving security requirements (RQ4). The survey consists of 20 questions (Appendix A) covering topics such as the students’ prior experience with AG and Flask development, the hours spent to complete each project phase, and their personal experiences and opinions. The survey was open for eight days and invitations to participate were distributed via email to all enrolled students that participated in at least one project phase.

4 WEB APPLICATION CASE STUDY

Here we describe the functional and security requirements of the web application, called *Event Platform*, to be developed within the project. We designed this application to have a very simple functionality, but complex security requirements. The students have received (almost) identical text as in this section [7].

We also describe the process of conducting the project, in particular how we adjusted the requirements during Phase I as a consequence of the requirements engineering process. This adjustment is in addition to the planned requirements evolution in Phase IV.

4.1 Event Platform

The Event Platform web application is a simplified version of typical event management platforms, like Meetup or Facebook Events. Anyone can visit the platform and register to access limited functionality. These users can also purchase a premium subscription to access additional features, like creating private events.

The application is used to create and maintain a set of events curated by its users: users can create, attend, request to attend, and (help) manage events. Additionally, events can be organized into categories, managed by moderators, to which users can subscribe. Management of the platform’s users is done by administrators.

4.1.1 Data model. Figure 1 shows Event Platform’s data model. The model consists of three classes (`Person`, `Event`, and `Category`) and seven binary associations. All association ends with multiplicity * are considered sets.

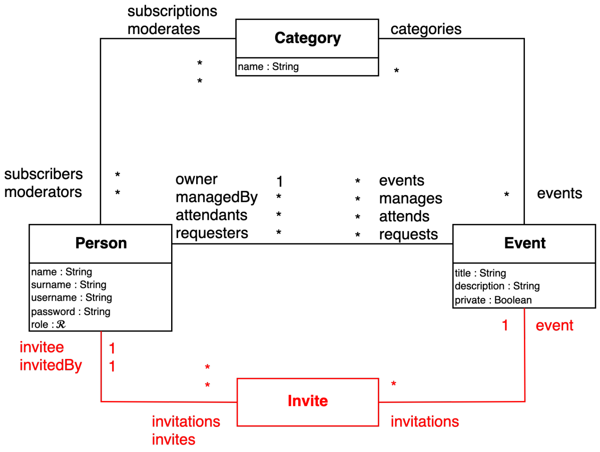

Figure 1: Data model of the Event Platform web application (black) and its evolution (red)

The `Person` class represents platform users and stores their `name`, `surname`, `username`, `password`, and `role`. The `Event` class includes a `title`, `description`, and a Boolean flag `private`. The `Category` class only contains the `name` of the category.

There are four associations between `Person` and `Event` classes. A person can create, manage, attend, or request to join multiple events; hence an event has `owner`, `managedBy`, `attendants` and `requesters` association ends. Dually, an event can be owned by exactly one person, and multiple people can manage, attend, or requested to join it. Respectively, these correspond to the `events`, `manages`, `attends`, and `requests` association ends.

Each event can belong to zero or more categories, which are referred to as the event’s `categories`. Each category contains zero or more events, which are referred to as the category’s `events`. A person can moderate or subscribe to multiple categories, and then they are in the category’s `moderators` and `subscribers`, respectively. Similarly, a category can be moderated or subscribed to by zero or more moderators; these are the user’s `moderates` and `subscriptions`, respectively.

Furthermore, the following invariants should hold on the data model: the `managedBy` and `attendants` association ends of an event always contain the event owner; similarly, the `attendants` association contains all the event managers. Note that the latter invariant corresponds to the one in Example 2.2. Category moderators can contain only persons with the Moderator role.

4.1.2 Security requirements. As stated before, we focus on FGAC policies as security requirements.

The authenticated users (who have previously registered with the application) can have one of the following four roles: Freeuser, Premiumuser, Moderator, or Admin. Since in AG every action must be executed in a context of a role, we explicitly introduce two more roles: the Visitor and the Lib role.

Users that interact with the application without previous authentication have their actions executed in the context of the Visitor role in AG. The Lib role is used to perform authentication actions.

In web applications this is typically done by an authentication library, hence the name of the role. To achieve this in the AG Lib role requires the corresponding permissions for logging in (i.e., reading `username`, `password`, and `role` of a `Person`), and for registering new users (i.e., creating a `Person`, initializing their `username` and `password`, and initializing their `role` to Freeuser). These two roles are not explicitly modeled in the Flask version of the application. In particular, the functionality of the Lib role is carried out by the Flask-User authentication library. The students were not required to modify the library to ensure least privilege. Rather, the library was assumed to be correctly implemented, as is typically done in practice. The permissions of the Visitor role are enforced whenever endpoints are called by a non-authenticated user, which can be detected by calling the Flask-User's `current_user.is_authenticated` property.

In what follows, we define the application's FGAC permissions based on two interrelated dimensions: user-based and role-based. When we refer to *everyone*, we mean both authenticated and non-authenticated users; otherwise we make an explicit distinction.

User-based permissions. We specify access control permissions for the set of attributes and association-ends of each class.

Authenticated users can create public events, and everyone can read a public event's core information such as its title, description, and owner as well as the event's categories. For private events, only attendants can read the core information. Event managers have permission to edit the event's core information, except for ownership, which cannot be changed once it has been initially set. Additionally, only event managers can modify the event's categories by adding or removing them.

For a public event, authenticated users can view the list of attendants and managers. However, for a private event, only its attendants have access to this information. The event owner has the permission to promote and demote users between being attendants and managers. Managers are authorized to remove an attendant from their managed event as long as the attendant is not the event's manager. Additionally, managers can accept new attendant requests to join their managed event. Any authenticated user, except event managers, can opt to withdraw themselves from attending the event. The event owner cannot remove themself from managing the event.

Users can request to join a public event and withdraw their requests. Managers can see all outstanding requests for the respective events they manage and can accept or deny these requests.

Within the `Person` class, authenticated users can view the core information (name, surname, username, and role) of any users. Additionally, they can update their own passwords and core information, with the exception of their role. Each user can access the list of the events they own, manage, attend, or have requested to join, as well as the categories to which they have subscribed. The categories' moderators can be viewed by everyone.

The name, associated events, and moderators of a category can be read by everyone. However, only the moderators of the category can see its subscribers. Users can remove themselves from being a category's moderator.

Role-based permissions. The following access control permissions are defined based on the system roles:

- The Visitor role can perform actions allowed to everyone.
- The Freeuser role can perform actions allowed to any authenticated user.
- The Premiumuser role has all the permissions of the Freeuser role and can also subscribe to and unsubscribe from categories, create private events, and request to join private events.
- The Moderator role has all the permissions of the Premiumuser role and can remove events from the category they moderate.
- The Admin role has all the permissions of the Premiumuser role, as well as to delete persons, and can edit any person's password and role, create and delete categories, edit categories' names, and add or remove users with the role moderator as moderators of a category.

Notice that the role hierarchy induced by the role-based permissions above is exactly the one from Example 2.3.

4.1.3 Security tests. We have designed an extensive set of test cases for both the AG and Flask applications. The tests are used both to assess the correctness of the implemented projects and as inputs for the project's Phase III.

The test cases are organized into groups based on scenarios inspired by the capabilities of different roles. For example, the Lib role can register a user, which means that it can create an instance of a `Person` and initialize its `username` and `password` fields. Moreover, each test can be executed by a caller in a different role. While the above scenario should be allowed when executed by the Lib role, it should not be allowed in the case the caller is in any other role. We summarize all the test cases in Appendix B.

For each test case group, we verify whether the submission correctly implements the access control policies, and whether the data invariants still hold after the actions have been performed.

As AG instruments every action from $\mathcal{A}$, we have defined a set of test cases for each role that covers each of these actions, under different eligible scenarios (i.e., feasible application states). In total, there are 455 test cases, consisting of 52 test cases for Lib, 59 for Visitor, 84 for Freeuser, 86 for Premiumuser, 88 for Moderator, 86 for Admin role.

In the case of the Flask version of the application, we have created one or more test cases for each test case in the AG application. The reason for having more test cases is that the actions from $\mathcal{A}$ are not automatically instrumented in Flask and may be repeated across multiple endpoints. For example, an event request is removed from the appropriate association both when a user is granted (endpoint `accept_request`) and denied (endpoint `reject_request`) attendance. The same removal action is executed in both endpoints, and hence multiple tests are needed to check if the authorization check is implemented correctly and consistently in both endpoints. In general, for each permission (r, a, ϕ) in the security model, we test that every endpoint containing the action a enforces the authorization constraint ϕ for the role r in different states of the application determined by the finitely many scenarios we consider. Similarly, like in AG, each test in Flask is executed by a caller in each role. Overall, this results in a total of 548 test cases, consisting of 70 cases for Visitor, 124 for Freeuser, 117 for Premiumuser, 120 for Moderator, and 117 for Admin.

We have manually ensured that the two sets of tests (for AG and Flask, respectively) are equivalent. The reason for not using a single (common) set of tests is because, unlike Flask, AG does not expose an easily testable API. Due to this technical limitation, we have tested the AG web application by creating special GUI models (one for each caller role), each containing a button with an event handler that runs the test code. The buttons are automatically invoked using a Selenium script.

4.1.4 Evolved requirements. In Phase IV of the project, we added a new *invitation* feature to the Event Platform application, which allows users to send event invitations to other users. To facilitate this, we extended the application's data model to include an `Invite` class and three adjacent associations (Figure 1, the red part). The class is associated with an event and with two users: the user who created the invitation and the user who is invited. The event associated with the invitation is denoted as `event` while the two users are `invitedBy` and `invitee`. Users can send and receive any number of invitations, which are referred to as their `invites` and `invitations`, respectively. Additionally, events can have any number of invitations, which are known as the event's `invitations`.

The new functional requirements impose an additional invariant stating that if a user receives an invitation to an event, then they cannot request to join it, and vice versa. All the above changes were provided to the students as part of the *TemplateUpdates* patch applicable to both the AG and Flask applications.

Finally, we extend our security requirements with FGAC permissions on the new `Invite` class. Namely, authenticated users can view the invitations they have sent and received. Additionally, they can view the core information and the list of attendants and managers of any event (including private one) to which they have been invited. Managers can see every invitation for the event they manage. While anyone can create a blank invitation, only managers are authorized to associate invitations with their managed events and add users as invitees to these invitations.

An invitation can be deleted by the invitation's owner, by the associated event's manager, or if the invitee declines it. If the invitation is accepted by the invitee, it will also be deleted, and the invitee will be added as an attendant of the associated event.

At the end of this phase, we ensure that the new functionality is implemented without breaking the existing security requirements by checking that the old set of test cases still pass. Then, we introduce a new set of test cases to test the new security requirements. In summary, for the AG application, we have 108 new test cases, with 10 for Visitor, 10 for the system role, and 22 for each of the other roles. For the Flask application, we have 96 new test cases, with 8 for Visitor, and 22 for each of the other roles.

4.2 Conducting the Project

Before the project starts, we provided the students with training on relevant topics. This included three exercise sessions and three lab sessions on AG and Flask. We have introduced all relevant authorization Flask libraries mentioned in Section 2 by example. The students were also asked to implement two AG and three Flask applications as part of the lab exercises. Although part of the course, these exercises were not mandatory, and hence we decided to survey the students to understand to what extent did they participate.

The students were not allowed to collaborate and exchange code during the study. They were encouraged to ask questions via a Moodle forum dedicated to the project or directly to the teaching assistants during the exercise and lab sessions. The forum discussions were visible to all students.

The forum was also used to facilitate requirements engineering. During Phase I, the students raised many questions due to the ambiguous nature of the informal project description and some minor technical problems (e.g., problems compiling the templates). To address these concerns, we updated the project description to better reflect the intended requirements and extended the submission deadline for Phase I by four days. Namely, the exception in the requirement that *any user except an event's manager can remove themselves from attending the event* originally applied only to the event's owner. We refined the requirement to exclude all the event's managers to ensure that it remains consistent with the data model invariants. The requirements were frozen once all students submitted their deliverable in Phase I.

In Phase III, we released the test cases (*TestsAG* and *TestsFlask*) and required the students to study them and fix as many implementation errors as possible. Overall, there were more than 1000 deterministic test cases that evaluated the specified security requirements. Unfortunately, there were some errors in the test cases, which were brought to our attention by the students. We quickly fixed the erroneous test cases, and the submission deadline remained unchanged.

5 DATA ANALYSIS

In this section, we present our analysis of the data obtained from the students' submissions and the survey responses. We first present the metrics that we computed from the data. We then use the information they provide to address the research questions posed in Section 3. Finally, we provide some general remarks beyond the scope of the research questions.

For simplicity, in what follows we refer to Phases I and II as the *Implementation Phases*, to Phase III as the *Repair Phase*, and to Phase IV as the *Evolution Phase*. Also, whenever we compare AG and Flask directly, we exclude all the tests that have the Lib role as the caller role. The reason is that the corresponding functionality is assumed to be correct in Flask and hence it is never tested.

5.1 Metrics

Table 2 displays an overview of the students' performance throughout the project. Each project phase is represented by a row containing the total number of submissions for that phase, followed by the number of failed test cases for each caller role executing the test and each tool used, averaged over all the submissions in the phase. The caller roles used to classify the failed test cases are those in the columns in Appendix B.

In Phase IV, we separately show the average number of failed test cases testing the new requirements (in the last column). We combine these new test cases across all roles and calculate the average number of failed test cases as explained above.

The pie chart in Figure 2 represents the ratio between the number of AG and Flask failed test cases summed across all submissions and phases. Since there are different numbers of AG and Flask tests,

Table 2: Average number of failed cases by submission and tool for each caller role

		Visitor		Freeuser		Premiumuser		Moderator		Admin		Evolution (all roles)	
	Submissions	AG	Flask	AG	Flask	AG	Flask	AG	Flask	AG	Flask	AG	Flask
Phase I	75	0.88	11.15	2.31	11.32	4.62	12.37	4.65	13.17	8.71	14.02	-	-
Phase II	74	2.05	12.29	4.35	10.23	7.70	10.94	7.77	11.61	11.87	10.97	-	-
Phase III	69	0.02	1.53	0.36	0.81	0.53	0.82	0.62	1.00	1.02	0.85	-	-
Phase IV	68	0.02	2.11	0.32	1.79	1.07	1.75	1.29	1.89	1.20	1.63	2.05	3.25

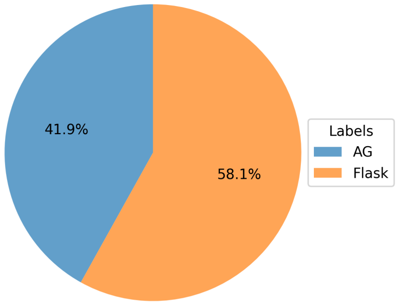

Figure 2: Ratio between AG and Flask failed tests across all submissions and phases

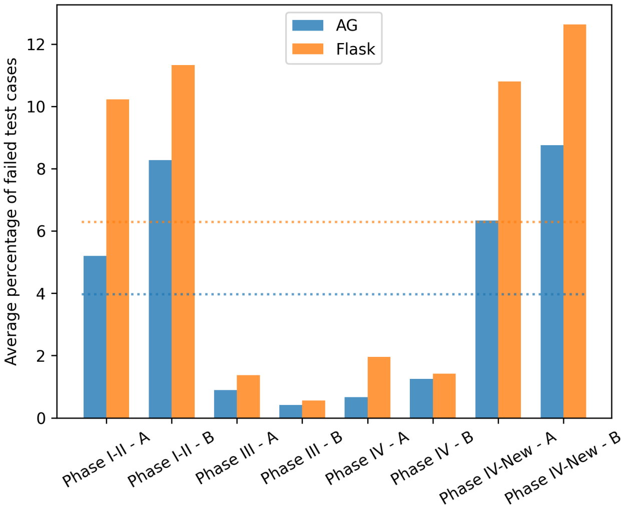

Figure 3: Average percentage of failed tests case by phase

we normalize the ratio by calculating a weight factor based on the number of test cases. More specifically, each AG test case has weight 1, while each Flask test case has weight $\frac{455}{548}$, i.e., the ratio between the number of AG and Flask tests. Then, for each tool, we multiply the total number of failed test cases by the corresponding weight.

Figure 3 presents the average percentage of failed test cases per project phase and student group. Specifically, we compare AG and Flask in three different phases (implementation, repair, and evolution) and for the distinct student groups A and B. Recall that Group A implemented the AG version of the application in Phase I and the Flask version of the application in Phase II, whereas Group B did it in the opposite order. We separately show percentages for the newly added test cases for Phase IV. The average percentage is computed by adding up the percentage of failed test cases per submission (i.e., number of failed test cases divided by the total number of test cases) and dividing it by the total number of submissions. The two dotted lines depict the average percentage for AG and Flask over all the phases.

Figure 4 shows the average percentage of failed test cases per role. Namely, we compare AG and Flask for all roles in each of the four phases. For each phase and caller role, we calculate the average percentage by adding up the percentage of failed test cases for that caller role per submission (i.e., number of failed test cases of the caller role divided by the total number of test cases of the caller role) and dividing it by the total number of submissions. As we show the average percentage of test failures, smaller values are better.

Figure 5 shows the average number of failed test cases grouped by the violated principles of least privilege and complete mediation. This allows us to evaluate the extent to which these principles were violated. As explained in Section 4.1, multiple Flask tests may correspond to a single AG one as we must test multiple endpoints. We call such a group of Flask tests *subtests*. Subtests are not necessary for AG as it achieves complete mediation by design. As a consequence, any AG test failure can be attributed to the violation of least privilege. In the case of Flask, determining which principle was violated is not as clear cut. A student may implement an authorization check correctly as a routine, but then fail to call it before all the relevant actions, which constitutes a complete mediation violation. In contrast, the check itself may be incorrect, or even implemented multiple times inconsistently. To this end, when some, but not all, subtests fail, we classify this as a complete mediation violation. Otherwise, we consider it a least privilege violation when all subtests fail. Using this criteria for each phase, we calculated the number of failed test cases grouped by the principle they violated, averaged over the phase's submissions. We have validated the criteria manually on 10% of the projects sampled randomly. It predicts the complete mediation and least privilege violations with a 3.02% margin of error at a 95% confidence level.

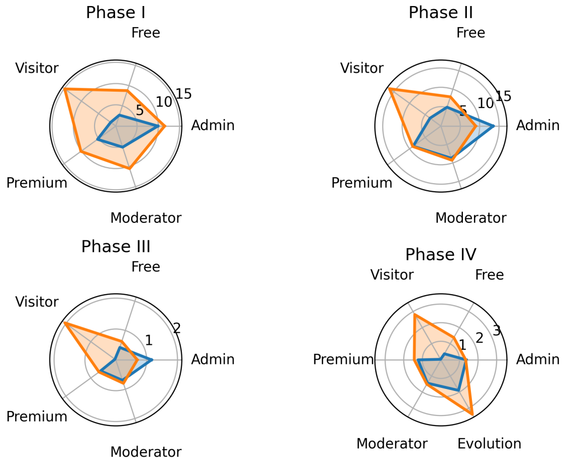

Figure 4: Average percentage of failed test cases per role

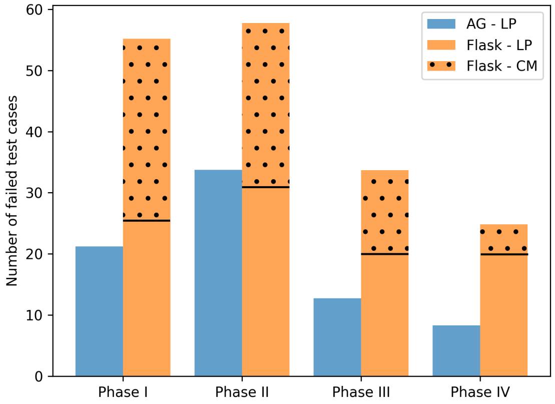

Figure 5: Least privilege and complete mediation violations

Figure 6 illustrates the time needed for completing the project using AG and Flask. The figure contains data collected from the survey responses (Questions 9 – 18 in Appendix A) and it shows the standard five-number summary of the distribution of time that the students required to implement all the project phases. In particular, the boxes delimit the 25th and 75th percentiles, the green line is the median value, with whiskers bounded by 1.5 of the interquartile range. We received 16 survey responses, which represents 23% of the total number of project participants. Among the participants, half of them were assigned to use Flask in Phase I, among which only 25% of them implement security requirements using an existing Flask authorization library.

Figure 7 depicts the time required for requirements engineering based on the survey responses (Question 8 in Appendix A).

Figure 8 shows the amount of effort needed in terms of lines of code (LoC) in each project phase grouped by the tool. It is computed as the ratio between the number of changed lines of code between two consecutive phases and the corresponding difference in the passed test cases, averaged over the submissions. For example, in the implementation phase (Phases I and II) for each submission, we consider the number of its LoC that are different than the provided template and divide that by the difference in the number of passed test cases. We focus on the difference in the number of passed test cases as a proxy for the relative security achieved with the LoC change. Note that a change may make an application less secure, and hence this metric may have a negative value. Finally we report the average and standard deviation of this metric across all submissions.

Figure 9 provides additional information on the amount of effort needed in terms of lines of code to achieve a perfectly secure application (i.e., an application that passes all test cases). To measure this accurately, only perfect submissions in Phases III and IV are considered. In Phase III, there were 55 perfect submissions in AG and 54 in Flask, while in Phase IV, there were 19 and 12, respectively. We computed the average number of changed lines of code as the difference between these submissions and the provided template (in Phase III), and similarly, between the submissions in Phases IV and III. In this, case we can guarantee that the metric does not assume a negative value for any of the submissions.

5.2 Answers to Research Questions

RQ1: Can, and to what extent, modeling access control help achieve least privilege? We use the metric shown in Figure 5 as a proxy for the achieved degree of least privilege. It shows that the least privilege principle is equally well enforced in both AG and Flask. However, AG preserves this property better than Flask during the repair and evolution phase. We hypothesize that with the model analysis tools that are able to automatically check for least privilege, the benefit of using models would be significantly greater.

Figure 3 also demonstrates the benefits of modeling access control. Here a lower percentage of failed tests is a proxy for improved least privilege. We see that AG outperforms Flask in all of the phases. Modeling access control is particularly beneficial in the implementation and evolution phases where access control is developed only based on the requirements (not based on tests).

RQ2: Can, and to what extent, modeling access control help achieve complete mediation? The metric in Figure 5 serves as a proxy for the degree of complete mediation achieved. As AG provides complete mediation by design and Flask lacks this property, AG is superior to Flask in this respect as demonstrated in Figure 5. Generally speaking, as long as access control models have a formal semantics and are accompanied with semantic-preserving model transformations, MDS can be used to achieve complete mediation by design.

RQ3: What is the effort needed to implement, fix, and evolve security in terms of lines of code? Figure 8 shows the metric that approximates the effort (LoC difference) needed to secure the application normalized by the degree of the achieved security (difference in the number of passed tests). Similarly, Figure 9 shows the effort needed, in terms of the number of changed LoC, to produce a perfectly secure implementation. In both figures, the metric is consistently better for AG than for Flask across the phases. This suggests implementation, repair, and evolution of security requirements in AG

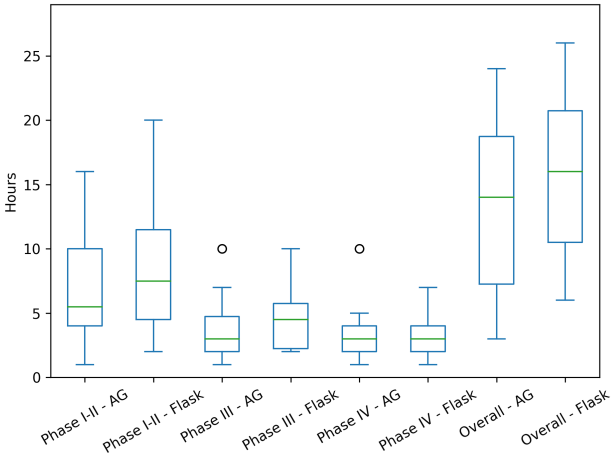

Figure 6: Time distribution for implementation

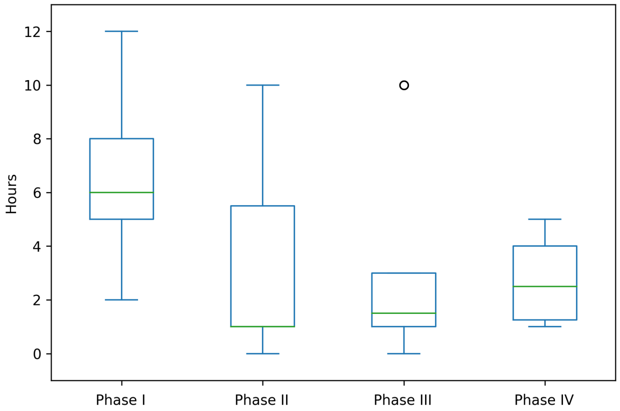

Figure 7: Time distribution for requirements engineering

requires less code than in Flask. We hypothesize that the conciseness of models compared to code improves system maintainability, as it requires less time to read. Nevertheless, number of LoC is only an approximation of the effort, as writing a line of code in different tools may require different effort. We therefore also consider RQ4.

RQ4: What is the effort needed to implement, fix, and evolve security in terms of time? Figure 6 shows the time needed for each phase as well as the overall time for each of the tools. In general, AG required on average 3 hours less time. The time required for the students to perform Phase II is less than in Phase I. This can be attributed to the fact that the requirements were clarified and became sufficiently clear after the requirements engineering step in Phase I. Furthermore, in the repair and evolution phases, it takes less time for the students to refine their AG security models compared to (re-)implementing the security checks in Flask, indicating that adaptation to changes in AG is less time-consuming than in Flask.

RQ5: Is implementing (resp., designing) access control more effective given an existing design (resp., reference implementation)? The difference between the percentage of test failures in Phase I and II in Figure 3 can be used as a proxy for determining the effect of the existing artifacts on the effectiveness of the implementation and design. The question of *why* AG implementations in Phase II are, on average, worse than those in Phase I cannot be answered conclusively using data from the project submissions alone. Nevertheless, based on survey responses, we can observe that students in Group A are more proficient with AG based on prior experience with this tool than students in Group B. A potential disparity in the student proficiency between the two groups may affect the outcome. The survey data reveals that all but one respondent has no prior experience with AG, whereas 75% of the students have used Flask before. Additionally, more than half of the respondents did not participate in the relevant training before the project.

RQ6: What are the most challenging aspects of access control to specify (resp., implement)? Figure 4 indicates that Flask tests executed with the caller role Visitor had the highest failure ratio in all project phases. Further analysis of the students' submissions reveals that a likely cause is Flask-User API misuse. Specifically, in all these tests the `current_user.is_authenticated` property always returns false and the `current_user` object is not initialized to any user. Inappropriate attribute lookup of the uninitialized object accounted for 46% of all unsuccessful test cases for the Visitor caller role. Although this is predominantly an error for the Visitor role, participants made the similar implementation mistake multiple times for other roles. In total, attribute errors accounted for 17% of all failed test cases in Flask.

AG tests executed with the caller role Admin had a higher failure ratio compared to Flask in Phases II and III. This is mainly due to insufficiently restrictive authorization constraints in permissions for updating roles and for maintaining data model invariants (e.g., preventing moderators from changing roles if they are currently moderating). In particular, the Admin role may update any user role to one of the authenticated user roles, but not to a Visitor, Lib, or even the `null` value. A typical pitfall was that the GUI model only offered authenticated user roles as possible values, which does not excuse the developer from performing the appropriate checks.

After an object is created, all its collection attributes and association ends are initially empty and non-collection fields are initially `null`. This fact is often used in update permissions to distinguish between attribute initialization and its subsequent modification. As a consequence, null values must never be assigned, which caused a significant number of test failures in AG. Nevertheless, MDS tools in general may consider initialization actions separately to alleviate this burden on the designer.

Overall, AG consistently outperforms Flask across all phases for the security implementation of all other user roles.

5.3 General Remarks

We summarize some remarks beyond the posed research questions:

- The majority of the requirements engineering effort was unsurprisingly spent during the project Phase I (Figure 7), when the students analyzed the project description (*Requirements*) and the templates (*TemplateAG* and *TemplateFlask*). According to the survey, during the first phase, requirements took

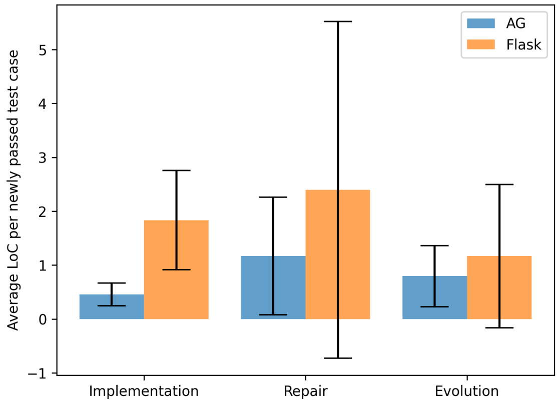

Figure 8: Average LoC with standard deviation per passed test case

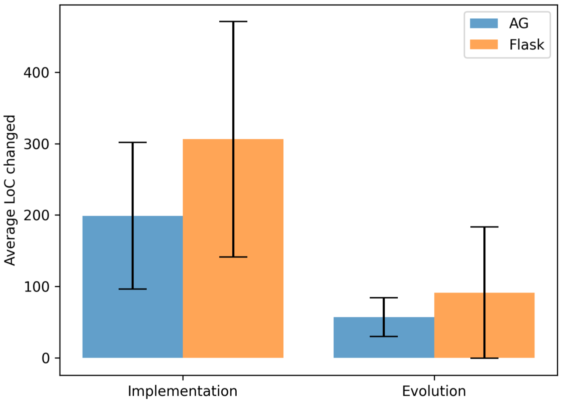

Figure 9: Average LoC with standard deviation for perfect implementation and evolution

between two to twelve hours to complete, with a median of six hours. As the requirements became clearer in the second phase, less time was needed, with a median of one hour of work. During the repair phase, some students needed to revisit the requirements to fix incorrect constraints/checks after the tests were released, resulting in a slight increase in requirements effort. Finally, in the evolution phase, the effort increased again to account for the new requirements.

- In the implementation phases, none of the submissions passed all the test cases. In particular, students had the most difficulty in correctly implementing the FGAC permissions for the Admin, Premiumuser, and Moderator roles Table 2). This is because these roles inherit permissions from the Freeuser role, which adds complexity to both security model design and manual implementation.
- In the repair phase, there was a significant decrease in the average number of failed test cases across all roles. This indicates that the students effectively fixed the enforcement errors in the implementation phases, regardless of the technology used. It also shows that it is more difficult to interpret requirements phrased in natural language than tests.
- In the evolution phase, introducing new requirements resulted in a slight increase in the average number of failures in the old set of test cases. In addition, the average number of failed test cases for the new requirements is consistent with the data from the previous phases.
- According to the results of our survey, only 6% of students had prior experience with AG, while 75% had prior experience with Flask. Conversely, only 6% of students submitted the course assignments related to AG and Flask. Due to the high proportion of individuals with prior experience, one might imagine that the students would perform better carrying out manual implementations using Flask. However, regardless of their lack of experience prior to the course, the students required less time and effort to complete the assigned tasks in AG in terms of overall hours.

6 RELATED WORK

There are several research areas related to our work that include the empirical comparison of model-driven and code-centric approaches. Particularly relevant is research making such comparisons together with evolution, maintenance, and access control. In the following, we organize this section along these different themes.

Empirical evidence on model-driven development benefits. A recent paper by Domingo et al. [11] surveys work on empirically measuring the benefits of model-driven development against code-centric approaches and propose a new experiment where a domain-specific language for video game development is compared to a code-centric approach. This work shows that in a model-driven approach, aside from advantages in efficiency previously reported in the literature, there are also advantages in terms of correctness with respect to a specification. In contrast to our work, which addresses web applications and access control, Domingo et al. focuses on a domain-specific language specific for developing videogames and general functional correctness.

Empirical studies on access control. Parkinson et al. [24] recently survey the literature for empirical assessments of the security of access control mechanisms. Among their key findings is that only relatively little research in access control focuses on empirical evaluations (key finding number 3, which lists only 30 works in the literature on the empirical analysis of access control). Many of these works use synthetically generated datasets and when using real-world datasets ground-truth is often unknown (finding number 7). The survey does not consider the comparison of code-centric and model-driven approaches for access control policies.

Empirical studies on access control and evolution. The research by Hwang et al. [18] and Han et al. [16] studies the evolution of access control policies over time using industrial data and shows that such policies tend to increase in complexity over time, which often results in errors and less security. They do not however compare the level of security or code complexity resulting from a model-driven approach versus a code-centric approach, as we do.

Model-inference applied to web-applications for maintenance of security policies. Gauthier et al. [14] have studied model-inference based on an existing web application to illustrate how reasoning at the model level can help with evolution and maintenance tasks. Although this study argues in favor of model-driven approaches over code-centric approaches, the discussion is not based on empirical measurements but on experts' experience when carrying out maintenance and evolution tasks. In contrast, we provide a direct comparison between the two approaches on the same development project.

Academic competitions on secure coding. The line of work started with "Build It, Break It, Fix It" [13, 28, 36] studies the result of a student competition where participants are asked to implement a software project and iteratively test and fix security vulnerabilities. These works shows that mistakes in implementing access control policies are common and they attempt to understand why certain vulnerabilities are introduced or go undetected. Interestingly, [13] found that misunderstandings are common when coding access control requirements, and that teams with a more detailed system design made fewer errors. However, this line of work does not compare a code-centric approach against a model-driven approach.

Evaluations of model-driven development in academic settings. Roussev [27] studies model-driven approaches to software development versus code-centric approaches in an academic settings with the goal of measuring impact on learning outcomes. Although it is interesting that students develop better overall programming skills when confronted with model-driven approaches, we focus our work on the comparison of both approaches in terms of security and maintenance.

Model-driven security in industrial settings Clavel et al. [10] report on the experience of using model-driven security in industrial settings. They use a transformation function from SecureUML [21] and Component UML to C++ code. They report on lessons learned from this experience, including the potential for re-usability and evolution of the models. They do not however provide empirical evidence of the advantage of this approach compared to a code-centric one.

To the best of our knowledge, our work is the first to empirically compare a model-driven approach with a code-centric approach when developing secure web-applications.

7 CONCLUSIONS

We have compared the outcome of implementing the same security requirements multiple times, with groups of developers using a model-driven approach and with other groups using a code-centric approach. This experiment was carried out in a graduate course in security engineering. We show that, on average, model-driven implementations were both more secure (with respect to a set of predefined test cases) and more concise (with respect to code-centric implementations). These results support a longstanding claim about the benefits of model-driven security for which empirical support in the literature was previously lacking.

As future work, we plan to collect more data from additional courses to further strengthen this claim. We also plan to test a version of ActionGUI that can be integrated with Python and Flask and supports enforcing privacy requirements.

ACKNOWLEDGMENTS

Hoang Nguyen is supported by the Swiss National Science Foundation grant 204796 ("Model-driven Security & Privacy"). Felix Linker contributed to the design of the previous versions of the Event Platform web application. Andris Suter-Dörig helped design the code templates and the training material for the students. Over the past decade, security engineering course staff at ETH Zürich have contributed to the ActionGUI training material. Finally, we thank the students who participated in our study and the anonymous reviewers for their helpful comments.

REFERENCES

[1] Amazon. 2023. Cedar Authorization Language. https://www.cedarpolicy.com/en.

[2] Colin Atkinson and Thomas Kühne. 2003. Model-Driven Development: A Metamodeling Foundation. *IEEE Softw.* 20, 5 (2003), 36–41. https://doi.org/10.1109/MS.2003.1231149

[3] Authorize. 2023. Flask Authorize Authorization library. https://flask-authorize.readthedocs.io/en/latest/.

[4] David Basin, Manuel Clavel, Jürgen Doser, and Marina Egea. 2009. Automated analysis of security-design models. *Inf. Softw. Technol.* 51, 5 (2009), 815–831. https://doi.org/10.1016/j.infsof.2008.05.011

[5] David Basin, Manuel Clavel, Marina Egea, Miguel Angel García de Dios, and Carolina Dania. 2014. A Model-Driven Methodology for Developing Secure Data-Management Applications. *IEEE Trans. Software Eng.* 40, 4 (2014), 324–337. https://doi.org/10.1109/TSE.2013.2297116

[6] David Basin, Jürgen Doser, and Torsten Lodderstedt. 2006. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.* 15, 1 (2006), 39–91. https://doi.org/10.1145/1125808.1125810

[7] David Basin, Juan Guarnizo, Srđan Krstić, Hoang Nguyen, and Martín Ochoa. 2023. Case study web application for access control implementation. https://anonymous.4open.science/r/MDS-study-7546.

[8] Bastian Best, Jan Jürjens, and Bashar Nuseibeh. 2007. Model-Based Security Engineering of Distributed Information Systems Using UMLsec. In *29th International Conference on Software Engineering (ICSE)*. IEEE, USA, 581–590. https://doi.org/10.1109/ICSE.2007.55

[9] Alan W. Brown. 2004. Model driven architecture: Principles and practice. *Softw. Syst. Model.* 3, 4 (2004), 314–327. https://doi.org/10.1007/s10270-004-0061-2

[10] Manuel Clavel, Viviane Torres da Silva, Christiano Braga, and Marina Egea. 2008. Model-Driven Security in Practice: An Industrial Experience. In *4th European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA) (LNCS, Vol. 5095)*, Ina Schieferdecker and Alan Hartman (Eds.). Springer, Berlin, Heidelberg, 326–337. https://doi.org/10.1007/978-3-540-69100-6_22

[11] África Domingo, Jorge Echeverría, Oscar Pastor, and Carlos Cetina. 2020. Evaluating the Benefits of Model-Driven Development - Empirical Evaluation Paper. In *32nd International Conference on Advanced Information Systems Engineering (CAiSE) (LNCS, Vol. 12127)*, Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant (Eds.). Springer, Berlin, Heidelberg, 353–367. https://doi.org/10.1007/978-3-030-49435-3_22

[12] Flask. 2023. Flask Web Framework. https://flask.palletsprojects.com/en/2.3.x/.

[13] Kelsey R. Fulton, Daniel Votipka, Desiree Abrokwa, Michelle L. Mazurek, Michael Hicks, and James Parker. 2022. Understanding the How and the Why: Exploring Secure Development Practices through a Course Competition. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi (Eds.). ACM, New York, NY, USA, 1141–1155. https://doi.org/10.1145/3548606.3560569

[14] François Gauthier, Ettore Merlo, Eleni Stroulia, and David Turner. 2014. Supporting Maintenance and Evolution of Access Control Models in Web Applications. In *30th IEEE International Conference on Software Maintenance and Evolution (ICSM)*. IEEE, USA, 506–510. https://doi.org/10.1109/ICSME.2014.83

[15] Object Management Group. 2014. Object Constraint Language (OCL) 2.4 Specification. https://www.omg.org/spec/OCL/2.4/.

[16] Zhuobing Han, Xiaohong Li, Guangquan Xu, Naixue Xiong, Ettore Merlo, and Eleni Stroulia. 2020. An Effective Evolutionary Analysis Scheme for Industrial Software Access Control Models. *IEEE Trans. Ind. Informatics* 16, 2 (2020), 1024–1034. https://doi.org/10.1109/TII.2019.2925422

[17] Martin Höst, Björn Regnell, and Claes Wohlin. 2000. Using Students as Subjects-A Comparative Study of Students and Professionals in Lead-Time Impact Assessment. *Empir. Softw. Eng.* 5, 3 (2000), 201–214.

[18] JeeHyun Hwang, Laurie A. Williams, and Mladen A. Vouk. 2014. Access control policy evolution: an empirical study. In *Symposium and Bootcamp on the Science of Security, (HotSoS)*, Laurie A. Williams, David M. Nicol, and Munindar P. Singh (Eds.). ACM, New York, NY, USA, 28. https://doi.org/10.1145/2600176.2600204

[19] Anneke Kleppe, Jos Warmer, and Wim Bast. 2003. *MDA explained - the Model Driven Architecture: practice and promise.* Addison-Wesley, USA. http://www.informit.com/store/mda-explained-the-model-driven-architecture-practice-9780321194428

[20] Charlie Lai, Li Gong, Larry Koved, Anthony J. Nadalin, and Roland Schemers. 1999. User Authentication and Authorization in the Java(tm) Platform. In *15th Conference on Computer Security Applications (ACSAC)*. IEEE, USA, 285–290. https://doi.org/10.1109/CSAC.1999.816038

[21] Torsten Lodderstedt, David Basin, and Jürgen Doser. 2002. SecureUML: A UML-Based Modeling Language for Model-Driven Security. In *5th International Conference on The Unified Modeling Language (UML) (LNCS, Vol. 2460)*, Jean-Marc Jézéquel, Heinrich Hußmann, and Stephen Cook (Eds.). Springer, Berlin, Heidelberg, 426–441. https://doi.org/10.1007/3-540-45800-X_33

[22] Microsoft. 2023. Policy-based authorization in ASP.NET Core. https://learn.microsoft.com/en-us/aspnet/core/security/authorization/policies.

[23] OSO. 2023. Flask OSO Authorization library. https://docs.osohq.com/python/reference/frameworks/flask.html.

[24] Simon Parkinson and Saad Khan. 2023. A Survey on Empirical Security Analysis of Access-control Systems: A Real-world Perspective. *ACM Comput. Surv.* 55, 6 (2023), 123:1–123:28. https://doi.org/10.1145/3533703

[25] Principal. 2023. Flask Principal Authorization library. https://pythonhosted.org/Flask-Principal/.

[26] Open Worldwide Application Security Project. 2021. OWASP Top 10. https://owasp.org/Top10/.

[27] Borislav Roussev. 2003. Empirical Evidence Justifying the Adoption of a Model-Based Approach in the Course Web Applications Development. *J. Inf. Technol. Educ.* 2 (2003), 73–90. https://doi.org/10.28945/314

[28] Andrew Ruef, Michael Hicks, James Parker, Dave Levin, Michelle L. Mazurek, and Piotr Mardziel. 2016. Build It, Break It, Fix It: Contesting Secure Development. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, New York, NY, USA, 690–703. https://doi.org/10.1145/2976749.2978382

[29] Iflaah Salman, Ayse Tosun Misirli, and Natalia Juristo Juzgado. 2015. Are Students Representatives of Professionals in Software Engineering Experiments?. In *37th IEEE/ACM International Conference on Software Engineering (ICSE)*, Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum (Eds.). IEEE, USA, 666–676. https://doi.org/10.1109/ICSE.2015.82

[30] Jerome H. Saltzer and Michael D. Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975), 1278–1308. https://doi.org/10.1109/PROC.1975.9939

[31] Richard E. Smith. 2012. A Contemporary Look at Saltzer and Schroeder's 1975 Design Principles. *IEEE Secur. Priv.* 10, 6 (2012), 20–25. https://doi.org/10.1109/MSP.2012.85

[32] SQLAlchemy. 2023. Flask SQLAlchemy SQL toolkit and Object Relational Mapper. https://flask-sqlalchemy.palletsprojects.com/en/2.x/.

[33] Mikael Svahnberg, Aybüke Aurum, and Claes Wohlin. 2008. Using students as subjects - an empirical evaluation. In *2nd International Symposium on Empirical Software Engineering and Measurement (ESEM)*, H. Dieter Rombach, Sebastian G. Elbaum, and Jürgen Münch (Eds.). ACM, New York, NY, USA, 288–290. https://doi.org/10.1145/1414004.1414055

[34] User. 2023. Flask User Authentication Library. https://flask-user.readthedocs.io.

[35] Vaadin. 2023. Vaadin Web Framework. https://vaadin.com/.

[36] Daniel Votipka, Kelsey R. Fulton, James Parker, Matthew Hou, Michelle L. Mazurek, and Michael Hicks. 2020. Understanding Security Mistakes Developers Make: Qualitative Analysis from Build It, Break It, Fix It. In *29th USENIX Conference on Security Symposium (SEC)*. USENIX Association, USA, Article 7, 18 pages.

[37] WIRED. 2019. Hack Brief: 885 Million Sensitive Financial Records Exposed Online. https://www.wired.com/story/first-american-data-exposed/.

A SURVEY QUESTIONS

Below we provide the complete list of questions intended for the survey participants. The questions are organized into different categories: background knowledge, requirements engineering, implementation effort, and approach preference. Each question includes an answer format (shown in parentheses).

Background knowledge.

(1) Have you done the Object Constraint Language assignment before the project? (Yes/No)
(2) Have you done the Model-driven security assignment before the project? (Yes/No)
(3) Have you done the Flask authorization assignment before the project? (Yes/No)
(4) Have you had any experience with AG before the course? (Yes/No)
(5) Have you had any experience with Python and Flask before the course? (Yes/No)
(6) In Phase I of the project, which approach were you assigned to? (AG/Flask)

Requirements engineering.

(7) Project requirements were sufficiently clear. (Strongly disagree/Disagree/Neither agree nor disagree/Agree/Strongly agree)
(8) In Phase IV of the project, which of the two approaches did you choose to evolve first? (AG/Flask)

Implementation effort.

(9) How many hours did you effectively spend on requirements engineering in Phase I of the project? (in hours)
(10) How many hours did you effectively spend to implement Phase I of the project? (in hours)
(11) How many hours did you effectively spend on requirements engineering in Phase II of the project? (in hours)
(12) How many hours did you effectively spend to implement Phase II of the project? (in hours)
(13) How many hours did you effectively spend on requirements engineering in Phase III of the project? (in hours)
(14) How many hours did you effectively spend to implement Phase III of the project in AG? (in hours)
(15) How many hours did you effectively spend to implement Phase III of the project in Flask? (in hours)
(16) How many hours did you effectively spend on requirements engineering in Phase IV of the project? (in hours)
(17) How many hours did you effectively spend to implement Phase IV of the project in AG? (in hours)
(18) How many hours did you effectively spend to implement Phase III of the project in Flask? (in hours)

Approach preference.

(19) For the Flask implementation, did you use an existing authorization library, or implement the security manually? (Library/Manual)
(20) If you would have to code a similar project in the future using one technology of your choice, which approach would you choose? Justify your choice in the previous answer. (AG/Flask with short text)

B TEST CASES

In Table 3, we summarize the test cases used to systematically evaluate the participants' submissions. The test cases are organized into groups based on scenarios inspired by the capabilities of different roles. Each test case is associated with a unique identifier, a textual description, a list of Flask endpoints, and the expected authorization decisions for each caller role: a checkmark (✓) means that the scenario is authorized, a cross (×) means that it is unauthorized, and a hyphen (–) means that the scenario is not applicable. Caller roles are abbreviated in the rightmost six columns. An example of latter is test 34 when executed with a Visitor caller role (abbreviated as V): a non-authenticated user does not have a representation in the data model, and hence they cannot be an event attendant.

Table 3: Test cases

ID	Textual description	`Flask endpoints`	L	V	F	P	M	A
Lib inspired								
1	Create a `Person` instance	`update_user`	✓	×	×	×	×	×
2	Update a `Person`'s `username` from `null` to something	`update_user`	✓	×	×	×	×	×
3	Update a `Person`'s `username` from something to something else	`update_user`	×	×	×	×	×	×
4	Update a `Person`'s `password` from `null` to something	`update_user`	✓	×	×	×	×	✓
5	Update a `Person`'s `password` from something to something else	`update_user`	×	×	×	×	×	✓
6	Update a `Person`'s `role` from `null` to `null`	`update_user`	×	×	×	×	×	×
7	Update a `Person`'s `role` from `null` to Freeuser	`update_user`	✓	×	×	×	×	✓
8	Update a `Person`'s `role` from `null` to Lib	`update_user`	×	×	×	×	×	×
9	Update a `Person`'s `role` from `null` to Admin	`update_user`	×	×	×	×	×	✓
10	Update a `Person`'s `role` from Freeuser to `null`	`update_user`	×	×	×	×	×	×
11	Update a `Person`'s `role` from Freeuser to Lib	`update_user`	×	×	×	×	×	×
12	Update a `Person`'s `role` from Freeuser to Premiumuser	`update_user`	×	×	×	×	×	✓
13	Update a `Person`'s `role` from Freeuser to Admin	`update_user`	×	×	×	×	×	✓
14	Read a `Person`'s `username` field	`user`	✓	×	✓	✓	✓	✓
15	Read a `Person`'s `password` field	`user`	✓	×	×	×	×	×
16	Read a `Person`'s `role` field	`user`	✓	×	✓	✓	✓	✓
Visitor inspired								
17	Read a `Person`'s `moderates` field	`view_event,edit_event`	×	✓	✓	✓	✓	✓
18	Read a public `Event`'s `private` flag	`view_event,edit_event`	×	✓	✓	✓	✓	✓
19	Read an `Event`'s `categories` field	`view_event`	×	✓	✓	✓	✓	✓
20	Read a public `Event`'s `title` field	`events,profile,view_event,edit_event,view_category`	×	✓	✓	✓	✓	✓
21	Read a private `Event`'s `title` field	`events,profile,view_event,edit_event,view_category`	×	×	×	×	×	×
22	Read a public `Event`'s `description` field	`view_event,edit_event`	×	✓	✓	✓	✓	✓
23	Read a private `Event`'s `description` field	`view_event,edit_event`	×	×	×	×	×	×
24	Read a public `Event`'s `owner` field	`events,view_category`	×	✓	✓	✓	✓	✓
25	Read a private `Event`'s `owner` field	`events,view_category`	×	×	×	×	×	×
26	Read a `Category`'s `name` field	`view_category,categories,view_event,events, edit_event,edit_category,user`	×	✓	✓	✓	✓	✓
27	Read a `Category`'s `moderators` field	`edit_category`	×	✓	✓	✓	✓	✓
28	Read a `Category`'s `events` field	`categories,view_category`	×	✓	✓	✓	✓	✓
Freeuser inspired								
29	Create a public `Event`	`create_event`	−	×	✓	✓	✓	✓
30	Read a public `Event`'s core information owned by someone else	`view_category,view_event,edit_event,events`	×	✓	✓	✓	✓	✓
31	Read a private `Event`'s core information owned by someone else without attending it	`view_category,view_event,edit_event,events`	×	×	×	×	×	×
32	Read the attendants of a public `Event` owned by someone else	`view_event,manage_event`	×	×	✓	✓	✓	✓
33	Read the attendants of a private `Event` owned by someone else without attending it	`view_event,manage_event`	×	×	×	×	×	×
34	Read the attendants of a private `Event` owned by someone else while attending it	`view_event,manage_event`	−	−	−	✓	✓	✓
35	Edit an `Event` owned by someone else while managing it	`update_event`	−	−	✓	✓	✓	✓
36	Edit an `Event` owned by someone else without managing it	`update_event`	×	×	×	×	×	×
37	Edit the owner of an `Event`	`update_event`	×	×	×	×	×	×
38	Promote an attendant to a manager in an `Event` that one owns	`promote_manager`	−	−	✓	✓	✓	✓
39	Demote a manager in an `Event` that one owns	`demote_manager`	−	−	✓	✓	✓	✓
40	Demote a manager in an `Event` that one does not own	`demote_manager`	×	×	×	×	×	×
41	Accept an attendance request for an `Event` that one owns	`accept_request`	−	−	✓	✓	✓	✓
42	Accept an attendance request for an `Event` that one manages	`accept_request`	−	−	✓	✓	✓	✓
43	Accept one's own attendance request, without being the `Event`'s owner or manager	`accept_request`	−	−	×	×	×	×
44	Promote oneself as a manager of an `Event` that one does not own	`promote_manager`	−	−	×	×	×	×
45	Reject an attendance request to an `Event` that one owns	`reject_request`	−	−	✓	✓	✓	✓
46	Reject an attendance request to an `Event` that one manages	`reject_request`	−	−	✓	✓	✓	✓
47	Remove oneself from attending an `Event` that one does not own	`remove_attendee,leave`	−	−	✓	✓	✓	✓
48	Remove others from attending an `Event` that one manages	`remove_attendee`	−	−	✓	✓	✓	✓
49	Remove oneself from attending an `Event` that one owns	`remove_attendee`	−	−	×	×	×	×
50	Remove others from attending an `Event` that one does not manage	`remove_attendee`	×	×	×	×	×	×
51	Request to join a public `Event`	`join`	−	×	✓	✓	✓	✓
52	Request to join a private `Event`	`join`	−	×	×	✓	✓	✓
53	Reject other's attendance request without being the `Event`'s manager nor owner	`reject_request`	×	×	×	×	×	×
54	Cancel one's own attendance request	`reject_request`	−	−	✓	✓	✓	✓
55	Request to join an `Event` on behalf of someone else	`events,view_event,manage_event,users,user`	×	×	×	×	×	×
56	Read core information of other users	`view_category,edit_category,`	×	×	✓	✓	✓	✓
57	Read one's own core information	`view_category,users,edit_category, view_event,user,manage_event,events`	−	−	✓	✓	✓	✓
58	Edit one's own core information	`update_user`	−	−	✓	✓	✓	✓
59	Edit others core information	`update_user`	×	×	×	×	×	×
60	View the events one owns	`profile`	−	−	✓	✓	✓	✓
61	View the events one manages	`profile`	−	−	✓	✓	✓	✓
62	View the events one attends	`profile`	−	−	✓	✓	✓	✓
63	Change one's own role	`update_user`	−	×	×	×	×	✓
64	View one's own attendance requests	`manage_event`	−	−	✓	✓	✓	✓
65	View one's own subscribed categories	`profile`	−	−	✓	✓	✓	✓

ID	Textual description	`Flask endpoints`	L	V	F	P	M	A
Premiumuser inspired								
66	Unsubscribe from a `Category`	`unsubscribe`	–	–	–	✓	✓	✓
67	Subscribe to a `Category`	`subscribe`	–	×	×	✓	✓	✓
68	Add a `Category` to one's own `subscriptions`	`subscribe`	–	×	×	✓	✓	✓
69	Create a private `Event`	`create_event`	–	×	×	✓	✓	✓
Moderator inspired								
70	Remove a `Category` that one does not moderate from an `Event`	`remove_category,update_event`	×	×	×	×	×	×
71	Remove someone else from being a moderator of a `Category`	`remove_moderator`	×	×	×	×	×	✓
72	Remove oneself as a moderator of a `Category`	`remove_moderator`	–	–	–	–	✓	–
73	Remove a `Category` that one moderates from an `Event`	`remove_category`	–	–	–	–	✓	–
74	Read a `Category`'s `subscribers`	`view_category,profile`	×	×	×	×	✓	×
Admin inspired								
75	Delete a `Person` instance	`user`	×	×	×	×	×	✓
76	Edit the `password` of another user	`user`	×	×	×	×	×	✓
77	Edit the `role` of another user	`user`	×	×	×	×	×	✓
78	Add a user as a moderator of a `Category`	`add_moderator`	×	×	×	×	×	✓
79	Remove a user as a moderator of a `Category`	`remove_moderator`	×	×	×	×	×	✓
80	Create a `Category`	`create_category`	×	×	×	×	×	✓
81	Delete a `Category`	`update_category`	×	×	×	×	×	✓
82	Change a `Category`'s `name`	`update_category`	×	×	×	×	×	✓
Miscellaneous								
83	An `Event`'s manager adds an attendant who has not requested attendance	`accept_request`	–	–	×	×	×	×
84	An `Event`'s owner removes oneself from managing the `Event`	`demote_manager`	–	–	×	×	×	×
85	An `Event`'s owner removes oneself from owning the `Event`	`remove_attendee`	–	–	×	×	×	×
86	An `Event`'s manager removes oneself from attending the `Event`	`remove_attendee`	–	–	×	×	×	×
87	An `Event`'s manager removes oneself from managing the `Event`	`demote_manager`	–	–	×	×	×	×
88	Modify the `role` of a moderator while they moderate a `Category`	`update_user`	–	–	–	–	–	×
89	Update a `Person`'s `role` from Freeuser to Visitor	`update_user`	×	×	×	×	×	×
90	Accept an attendance request for an `Event` that one does not own or manage	`accept_request`	×	×	×	×	×	×
91	Edit the `categories` of an `Event` that one manages	`update_event`	–	–	✓	✓	✓	✓
92	Edit the `categories` of an `Event` that one does not manage	`update_event`	×	×	×	×	×	×
93	Read a private `Event`'s `private` flag	`view_event,edit_event`	×	✓	✓	✓	✓	✓
94	Read a private `Event`'s `categories`	`view_event`	×	✓	✓	✓	✓	✓
Invitation inspired								
1	Read the attendants of a private `Event` that one does not own while not invited	`view_event,manage_event`	–	–	✓	✓	✓	✓
2	Read the attendants of a private `Event` that one does not own while neither invited nor attending	`view_event,manage_event`	×	×	×	×	×	×
3	Read the core information of a private `Event` that one does not own while invited	`events,view_event,edit_event,view_category`	×	×	✓	✓	✓	✓
4	Read the core information of a private `Event` that one does not own while neither invited nor attending	`events,view_event,edit_event,view_category`	×	×	×	×	×	×
5	View the sent invitations of an `Event` that one owns	`profile`	–	–	✓	✓	✓	✓
6	View the sent invitations of an `Event` that one does not own	`profile`	×	×	×	×	×	×
7	View the received invitations of an `Event` that one owns	`profile`			✓	✓	✓	✓
8	View the received invitations of an `Event` that one does not own	`profile`	×	×	×	×	×	×
9	Invite a user to join an `Event` that one owns	`send_invitation`	–	–	✓	✓	✓	✓
10	Invite a user to join an `Event` that one does not own without being the event's manager	`send_invitation`	–	–	×	×	×	×
11	Invite user to join an `Event` that one manages	`send_invitation`	–	–	✓	✓	✓	✓
12	Cancel an invitation to join an `Event` that one owns	–	–	–	✓	✓	✓	✓
13	Cancel an invitation for someone else	–	×	×	×	×	×	×
14	Cancel an invitation to join an Event that one manages	–	–	–	✓	✓	✓	✓
15	Accept a received invitation	`accept_invitation`	–	–	✓	✓	✓	✓
16	Accept an invitation received by someone else	`accept_invitation`	×	×	×	×	×	×
17	Decline a received invitation	`decline_invitation`	–	–	✓	✓	✓	✓
18	Change an `Invite`'s `invitee` field	–	×	×	×	×	×	×
19	Change an `Invite`'s `invitedBy` field	–	×	×	×	×	×	×
20	Change an `Invite`'s `event` field	–	×	×	×	×	×	×
21	Invite a user who already requested to join an `Event`, while being the event's manager	`send_invitation`	–	–	×	×	×	×
22	Request to join an `Event` to which one is already invited	`join`	–	–	×	×	×	×

Abbreviated caller roles: L = Lib role; V = Visitor role; F = Freeuser role; P = Premiumuser role; M = Moderator role; A = Admin role.