Titre: Title:	Reverse-Engineering and Analysis of Access Control Models in Web Applications
Auteur: Author:	François Gauthier
Date:	2014
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Gauthier, F. (2014). Reverse-Engineering and Analysis of Access Control Models in Web Applications [Thèse de doctorat, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/1437/

URL de PolyPublie: PolyPublie URL:	https://publications.polymtl.ca/1437/
Directeurs de recherche: Advisors:	Ettore Merlo, & Michel Gagnon
Programme: Program:	Génie informatique

UNIVERSITÉ DE MONTRÉAL

REVERSE-ENGINEERING AND ANALYSIS OF ACCESS CONTROL MODELS IN WEB APPLICATIONS

FRANÇOIS GAUTHIER
DÉPARTEMENT DE GÉNIE INFORMATIQUE ET GÉNIE LOGICIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(GÉNIE INFORMATIQUE)
MAI 2014

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée:

REVERSE-ENGINEERING AND ANALYSIS OF ACCESS CONTROL MODELS IN WEB APPLICATIONS

présentée par: GAUTHIER François
en vue de l'obtention du diplôme de: Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de:

M. ADAMS Bram, Doct., président
M. MERLO Ettore, Ph.D., membre et directeur de recherche
M. GAGNON Michel, Ph.D., membre et codirecteur de recherche
M. ANTONIOL Giuliano, Ph.D., membre
M. DEAN Thomas R., Ph.D., membre

To Mélanie, my one and only.

ACKNOWLEDGMENTS

This research work could not have happened without the help of several people and funding agencies.

First and foremost I wish to thank Ettore Merlo, my advisor, for believing in me and letting me grow as a researcher and as a person. My Ph.D. was not only about learning new techniques, developing new algorithms and writing papers, it was also, and most notably about learning about myself, pushing my limits, increasing self-confidence, coping with failures and celebrating accomplishments. It was about getting out of my comfort zone and facing new challenges. Thanks again Ettore, it has been a wonderful experience!

I also wish to thank my lab mates: Dominic, Maxime, Marc-André, Théotime and Thierry for all the insights, encouragements and laughter. I will miss you guys. I wish you all the best!

Special thanks to the professors who agreed to serve on my thesis committee: Dr. Bram Adams, Dr. Ettore Merlo, Dr. Michel Gagnon, Dr. Giuliano Antoniol and Dr. Thomas Dean.

I also want to thank the NSERC and the FQRNT for funding my research. Their support allowed me to fully focus on my doctoral studies. It was a great privilege and I am very grateful.

Finally, I wish to thank my parents, my number one fans, for 30 years of love, care, and unshakable support and my sweetheart Mélanie for her love, patience, understanding and for making this crazy long distance adventure possible!

RÉSUMÉ

De nos jours, les applications Web sont omniprésentes et gèrent des quantités toujours plus importantes de données confidentielles. Afin de protéger ces données contre les attaques d'usagers mal intentionnés, des mécanismes de sécurité doivent être mis en place. Toutefois, sécuriser un logiciel est une tâche extrêmement ardue puisqu'une seule brèche est souvent suffisante pour compromettre la sécurité d'un système tout entier. Il n'est donc pas surprenant de constater que jour après jour les nouvelles font état de cyber attaques et de fuites de données confidentielles dans les systèmes informatiques. Afin de donner au lecteur une vague idée de l'ampleur du problème, considérons que différents organismes spécialisés en sécurité informatique rapportent qu'entre 85% et 98% des sites Web contiennent au moins une vulnérabilité *sérieuse*.

Dans le cadre de cette thèse, nous nous concentrerons sur un aspect particulier de la sécurité logicielle, à savoir les modèles de contrôle d'accès. Les modèles de contrôle d'accès définissent les actions qu'un usager peut et ne peut pas faire dans un système. Malheureusement, années après années, les failles dans les modèles de contrôle d'accès trônent au sommet des palmarès des failles les plus communes et les plus critiques dans les applications Web. Toutefois, contrairement à d'autres types de faille de sécurité comme les injections SQL (SQLi) et le cross-site scripting (XSS), les failles de contrôle d'accès ont comparativement reçu peu d'attention de la communauté de recherche scientifique. Par ce travail de recherche, nous espérons renverser cette tendance.

Bien que la sécurité des applications et les modèles de contrôle d'accès constituent les principaux thèmes sous-jacents de cette thèse, notre travail de recherche est aussi fortement teinté par le génie logiciel. Vous observerez en effet que notre travail s'applique toujours à des applications réelles et que les approches que nous développons sont toujours construites de manière à minimiser le fardeau de travail supplémentaire pour les développeurs. En d'autres mots, cette thèse porte sur la sécurité des applications en *pratique*.

Dans le contexte de cette thèse, nous aborderons l'imposant défi d'investiguer des modèles de contrôle d'accès non spécifiés et souvent non documentés, tels que rencontrés dans les application Web en code ouvert. En effet, les failles de contrôle d'accès se manifestent lorsqu'un usager est en mesure de faire des actions qu'il ne *devrait* pas pouvoir faire ou d'accéder à des données auxquelles il ne *devrait* pas avoir accès. En absence de spécifications de sécurité, déterminer qui *devrait* avoir les autorisations pour effectuer certaines actions ou accéder à certaines données n'est pas simple.

Afin de surmonter ce défi, nous avons d'abord développé une nouvelle approche, appellée

analyse de Traversement de Patrons de Sécurité (TPS), afin de faire la rétro-ingénierie de modèles de contrôle d'accès à partir du code source d'applications Web et ce, d'une manière rapide, précise et évolutive. Les résultats de l'analyse TPS donnent un portrait du modèle de contrôle d'accès tel qu'implémenté dans une application et servent de point de départ à des analyses plus poussées.

Par exemple, les applications Web réelles comprennent souvent des centaines de privilèges qui protègent plusieurs centaines de fonctions et modules différents. En conséquence, les modèles de contrôle d'accès, tel qu'extraits par l'analyse TPS, peuvent être difficiles à interpréter du point de vue du développeur, principalement à cause de leurs taille. Afin de surmonter cette limitation, nous avons exploré comment l'analyse formelle de concepts peut faciliter la compréhension des modèles extraits en fournissant un support visuel ainsi qu'un cadre formel de raisonnement. Les résultats ont en effet démontrés que l'analyse formelle de concepts permet de mettre en lumière plusieurs propriétés des modèles de contrôle d'accès qui sont enfouies profondément dans le code des applications, qui sont invisibles aux administrateurs et aux développeurs, et qui peuvent causer des incompréhensions et des failles de sécurité.

Au fil de nos investigations et de nos observations de plusieurs modèles de contrôle d'accès, nous avons aussi identifié des patrons récurrents, problématiques et indépendants des applications qui mènent à des failles de contrôle d'accès. La seconde partie de cette thèse présente les approches que nous avons développées afin de tirer profit des résultats de l'analysis TPS pour identifier automatiquement plusieurs types de failles de contrôle d'accès communes comme les vulnérabilités de navigation forcée, les erreurs sémantiques et les failles basées sur les clones à protection incohérentes. Chacune de ces approches interprète en effet les résultats de l'analyse TPS sous des angles différents afin d'identifier différents types de vulnérabilités dans les modèles de contrôle d'accès.

Les vulnérabilités de navigation forcée se produisent lorsque des ressources sensibles ne sont pas adéquatement protégées contre les accès direct à leur URL. En utilisant les résultats de l'analyse TPS, nous avons montré comment nous sommes en mesure de détecter ces vulnérabilités de manière précise et très rapide (jusqu'à 890 $\times$ plus rapidement que l'état de l'art).

Les erreurs sémantiques se produisent quand des resources sensibles sont protégées par des privilèges qui sont sémantiquement incorrects. Afin d'illustrer notre propos, dans le contexte d'une application Web, protéger l'accès à des ressources administratives avec un privilège destiné à restreindre le téléversement de fichiers est un exemple d'erreur sémantique. À notre connaissance, nous avons été les premiers à nous attaquer à ce problème et à identifier avec succès des erreurs sémantiques dans des modèles de contrôle d'accès. Nous avons obtenu de

tels résultats en interprétant les résultats de l'analyse TPS à la lumière d'une technique de traitement de la langue naturelle appellée Latent Dirichlet Allocation.

Finalement, en investiguant les résultats de l'analyse TPS à la lumière des informations fournies par une analyse de clones logiciels, nous avons été en mesure d'identifier davantage de nouvelles failles de contrôle d'accès. En résumé, nous avons exploré l'intuition selon laquelle il est attendu que les clones logiciels, qui sont des blocs de code syntaxiquement similaires, effectuent des opérations similaires dans un système et, conséquemment, qu'ils soient protégés de manière similaire. En investiguant les clones qui ne sont pas protégés de manière similaire, nous avons effectivement été en mesure de détecter et rapporter plusieurs nouvelles failles de sécurité dans les systèmes étudiés.

En dépit des progrès significatifs que nous avons accomplis dans cette thèse, la recherche sur les modèles de contrôle d'accès et les failles de contrôle d'accès, spécialement d'un point de vue pratique n'en est encore qu'à ses débuts. D'un point de vue de génie logiciel, il reste encore beaucoup de travail à accomplir en ce qui concerne l'extraction, la modélisation, la compréhension et les tests de modèles de contrôle d'accès. Tout au long de cette thèse, nous discuterons comment les travaux présentés peuvent soutenir ces activités et suggérerons plusieurs avenues de recherche à explorer.

ABSTRACT

Nowadays, Web applications are ubiquitous and deal with increasingly large amounts of confidential data. In order to protect these data from malicious users, security mechanisms must be put in place. Securing software, however, is an extremely difficult task since a single breach is often sufficient to compromise the security of a system. Therefore, it is not surprising that day after day, we hear about cyberattacks and confidential data leaks in the news. To give the reader an idea, various reports suggest that between 85% and 98% of websites contain at least one *serious* vulnerability.

In this thesis, we focus on one particular aspect of software security that is access control models. Access control models are critical security components that define the actions a user can and cannot do in a system. Year after year, several security organizations report access control flaws among the most prevalent and critical flaws in Web applications. However, contrary to other types of security flaws such as SQL injection (SQLi) and cross-site scripting (XSS), access control flaws comparatively received little attention from the research community. This research work attempts to reverse this trend.

While application security and access control models are the main underlying themes of this thesis, our research work is also strongly anchored in software engineering. You will observe that our work is always based on real-world Web applications and that the approaches we developed are always built in a such way as to minimize the amount of work on that is required from developers. In other words, this thesis is about *practical* software security.

In the context of this thesis, we tackle the highly challenging problem of investigating unspecified and often undocumented access control models in open source Web applications. Indeed, access control flaws occur when some user is able to perform operations he *should* not be able to do or access data he *should* be denied access to. In the absence of security specifications, determining who *should* have the authorization to perform specific operations or access specific data is not straightforward.

In order to overcome this challenge, we first developed a novel approach, called the Security Pattern Traversal (SPT) analysis, to reverse-engineer access control models from the source code of applications in a fast, precise and scalable manner. Results from SPT analysis give a portrait of the access control model as implemented in an application and serve as a baseline for further analyzes.

For example, real-world Web application, often define several hundred privileges that protect hundreds of different functions and modules. As a consequence, access control models, as reverse-engineered by SPT analysis, can be difficult to interpret from a developer point of

view, due to their size. In order to provide better support to developers, we explored how Formal Concept Analysis (FCA) could facilitate comprehension by providing visual support as well as automated reasoning about the extracted access control models. Results indeed revealed how FCA could highlight properties about implemented access control models that are buried deep into the source code of applications, that are invisible to administrators and developers, and that can cause misunderstandings and vulnerabilities.

Through investigation and observation of several Web applications, we also identified recurring and cross-application error-prone patterns in access control models. The second half of this thesis presents the approaches we developed to leverage SPT results to automatically capture these patterns that lead to access control flaws such as forced browsing vulnerabilities, semantic errors and security-discordant clone based errors. Each of these approaches interpret SPT analysis results from different angles to identify different kinds of access control flaws in Web applications.

Forced browsing vulnerabilities occur when security-sensitive resources are not protected against direct access to their URL. Using results from SPT, we showed how we can detect such vulnerabilities in a precise and very fast (up to 890 $\times$ faster than state of the art) way.

Semantic errors occur when security-sensitive resources are protected by semantically *wrong* privileges. To give the reader an idea, in the context of a Web application, protecting access to administrative resources with a privilege that is designed to restrict file uploads is an example of semantic error. To our knowledge, we were the first to tackle this problem and to successfully detect semantic errors in access control models. We achieved such results by interpreting results from SPT in the light of a natural language processing technique called Latent Dirichlet Allocation.

Finally, by investigating SPT results in the light of software clones, we were able to detect yet other novel access control flaws. Simply put, we explored the intuition that code clones, that are blocks of code that are syntactically similar, are expected to perform similar operations in a system and, consequently, be protected by similar privileges. By investigating clones that are protected in different ways, called security-discordant clones, we were able to report several novel access control flaws in the investigated systems.

Despite the significant advancements that were made through this thesis, research on access control models and access control flaws, especially from a practical, application-centric point of view, is still in the early stages. From a software engineering perspective, a lot of work remains to be done from the extraction, modelling, understanding and testing perspectives. Throughout this thesis we discuss how the presented work can help in these perspectives and suggest further lines of research.

TABLE OF CONTENT

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

ABAC	Attribute-Based Access Control
ACL	Access Control List
CSRF	Cross-Site Request Forgery
DAC	Discretionary Access Control
FCA	Formal Concept Analysis
MAC	Mandatory Access Control
NEC	Necessarily Enabled Capability
OWASP	Open Web Application Security Project
RBAC	Role-Based Access Control Model
SPT	Security Pattern Traversal Analysis
SQLi	SQL Injection
XSS	Cross-Site Scripting

CHAPTER 1

INTRODUCTION

Nowadays, Web applications are an integral part of our daily lives. We use them to shop, bank, communicate and work. Consequently, these applications now manage increasingly large amounts of confidential and sensitive data. Sadly, over the years, a parallel market developed where security flaws in software are exploited for profit or used to blackmail people and industries. Whereas cyber attacks where mainly the fact of isolated hackers in the 90's, we now observe organized communities and even industries whose sole purpose is to exploit software flaws for profit. Governments and industries are now well aware of this problem, as the costs associated to cybercrimes are reaching new highs [13]. To give the reader an idea, various reports suggest that between 85% and 98% of websites contain at least one *serious* vulnerability [147, 149]. While the average number of vulnerabilities per website decreases year after year, a lot of work obviously remains to be done to properly secure modern Web applications.

Securing modern Web applications, however, is a very difficulty task. Network (routers, firewalls and switch), host (operating system, web server, database server, application server) and applications themselves are all potential targets for cyber attacks and any suitable security strategy must take them all into account. Indeed, a vulnerability at any level can compromise the security of a system and a large body of research work has been dedicated to securing each of these components. In the context of this thesis, we will focus on one particular aspect of computer security, that is application security.

Application security encompasses the mechanisms that are enforced inside an application to mitigate security threats. In the context of Web applications, application level threats take several forms. SQL injection (SQLi), cross-site scripting (XSS), cross-site request forgery (CSRF), authentication and authorization bypass all are examples of threats that must be mitigated at the application level [129]. In the context of this thesis, we will mainly focus on authorization flaws, that allow users to perform actions that they *should* not be able to do. Authorization flaws drastically differ from other types of flaws by the fact that they are strongly linked to the application's logic. To clarify the latter affirmation, let's consider some typical flaws in Web applications:

SQL injection attacks aim at making the application execute malicious SQL queries, possibly with the goal of extracting sensitive data. SQL injection vulnerabilities occur when untrusted data flows to the SQL interpreter.

Cross-site scripting attacks aim at making malicious scripts execute in the user's browser, possibly defacing the website or redirecting him to malicious sites. Cross-site scripting vulnerabilities occur when untrusted data flows to the browser. Both SQLi and XSS attacks can be mitigated by filtering untrusted data before it flows to any sensitive section of the application.

Cross-site request forgery attacks aim at forcing the browser of a user that is logged into a vulnerable application to send a forged HTTP request to the vulnerable application. Since the user is already logged in, the attacker can access otherwise restricted areas of the vulnerable application. Mitigating CSRF attacks can be as simple as requesting user specific tokens with any form submission or side-effect URLs.

Authentication bypass allows an attacker to impersonate a legitimate user and perform any action the user is allowed to do. Authentication flaws can occur at several levels, such as logout, password management, timeouts, "remember me" options, secret question, account update, etc. Since authentication schemes are often custom-built, authentication flaws can be hard to detect as each implementation is unique.

Authorization bypass, on the other hand, allows a user to perform actions he *should* not be able to do. Authorization flaws usually occur when security checks are missing or inappropriate.

SQLi, XSS, CSRF and authentication flaws are all of the "black or white" type (e.g. if unfiltered untrusted inputs can flow to a SQL query, SQL injections are possible, if a malicious user can impersonate a legitimate user, there is an authentication flaw, etc.). Authorization flaws, on the other hand, come in shades of grey.

Given a piece of code, developers first have to determine whether it performs security-sensitive operations. While certain operations are obviously security-sensitive (e.g. displaying personal user information), some cases are more subtle (e.g. leaking path information in an uncaught exception). Once established that a piece of code does perform security-sensitive operations, developers must determine which privilege (authorization) check will be enforced. Indeed, modern Web application often define hundreds of application specific privileges and deciding which one to enforce might not be straightforward. Finally, developers must decide where in the code the privilege check will be implemented. Checks that are performed at the beginning of a file quickly redirect unprivileged users to an error page and are appropriate in cases where the whole file must be protected. Checks that protect smaller portions of code allow for fine-grain privilege-based customizations, with the risk of leaving certain security-sensitive operations unprotected. It is not uncommon that issues related to authorization checks are debated for years, as there is no one-size-fits-all solution [84].

Moreover, one must not forget that software evolves, gets patched and refactored, new

features are added and all of these modifications can impact the authorization scheme. To properly prevent authorization flaws, authorization schemes must evolve with the software. From design to testing via implementation, authorization schemes must be synchronized and updated at every step of the software development life cycle.

Preventing and detecting authorization flaws is thus hard and error-prone, even for developers that have a working knowledge of their application [178]. Therefore, it is not surprising that year after year, authorization flaws are reported as one of the most prevalent and critical type of flaws in Web applications [131, 147, 149, 148, 81, 151]. In this thesis, we tackle the problem of detecting authorization flaws in open source Web applications, without *a priori* knowledge and with source code as the only source of reliable information.

1.1 Authorization schemes

Authorization schemes determine the actions a user is allowed to perform in an application. In practice, authorization schemes are implemented using access controls and managed using access control models. Depending on the implementation, authorization can be managed from a very coarse to a very granular level.

1.1.1 Principle of Least Privilege

When designing authorization scheme, it is important to enforce the principle of least privilege. The principle of least privileges states that we should only allow the bare minimum of privileges a user needs to perform the tasks he is allowed to do. For example, people from the sales department need not to access data from the human resources to do their job. While easily understood, the principle of least privilege is also easily violated, especially as access control models grow in complexity.

A classic example of violation of the principle of least privilege can be found in the default configuration of the Windows operating system. In Windows, it is rather common that casual users always run the system with administrator privileges. Hence, a malicious program that run under such an account already has the necessary privileges to perform any action in the system. When the principle of least privilege is respected, malicious programs must exploit other bugs in the system to gain higher privileges *before* performing their malicious actions. The act of exploiting bugs or flaws to gain supplementary or higher privileges in a system is often referred to as a *privilege escalation* attack.

1.2 Access controls

Access controls implement authorization schemes. In other words, access controls are the concrete mechanisms that are put in place to assert whether or not the current user can execute a given operation on a given resource. Typical operations include *read*, *write* and *execute*. Resources are application dependent, but might include things like files, database entries or sessions. Two major strategies, that we present in the next two paragraphs, are typically used to implement access controls: access control lists and capabilities.

1.2.1 ACL-based systems

Access control lists (ACL) are often used to control access in file systems but can also be found in Web applications. ACL are linked to a resource (e.g. a file) and determine which operations each user or group of users are allowed to perform on the resource. For example, in the UNIX file system, each file has its own ACL that determines which operations (*read*, *write* and *execute*) each type of users (*owner*, *group* and *others*) is allowed to perform on the file.

1.2.2 Capability-based systems

Whereas access controls in ACL-based system depend on the resource and the type of user, access controls in capability-based system strictly depend on the possession of a capability. Informally, capabilities can be though of as pair (r,o) where r is a resource (e.g. a file) and o is an operation (e.g. write). In capability-based systems, no matter the identity of the user, if he owns the required capability, he can perform the given operation on the given resource. Originally, capabilities were designed to act as tokens that were meant to be transferable between users and applications. Analogously to a physical key, whoever owned the capability token was granted access to the resource [101]. However, modern applications usually implement restricted capability-based systems where capabilities are not transferable between applications.

1.3 Access control models

Access control lists and capabilities define the low-level mechanisms that are used to grant or deny access to resources. Access control models provide a higher level view and facilitate the management of access controls in a system. Among the most common access control models, we find Discretionary Access Control (DAC), Mandatory Access Control (MAC), Attribute-Based Access Control (ABAC) and Role-Based Access Control (RBAC).

1.3.1 Discretionary Access Control

In Discretionary Access Control (DAC) models, the owner of a resource determines the access to the resource in question. File systems typically implement DAC models. Access controls of the UNIX file system, for example, are implemented with ACL and managed with a DAC model. By default, only the owner of a file can edit the ACL of the file. Hypothetically, if DAC models were to be used in a capability-based system, they would allow the owner of a resource to restrict ownership of all capabilities that are related to the resource in question. DAC models, however, are usually associated to ACL.

1.3.2 MAC

In Mandatory Access Control (MAC) models, access to resources is not determined by owners, but by a group of users that have the authority to set access on resources. Otherwise, DAC and MAC are very similar in nature.

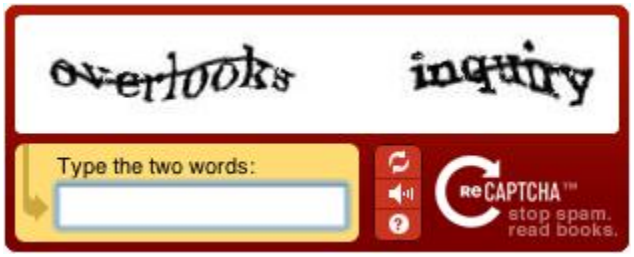


Figure 1.1: Example of a captcha

1.3.3 ABAC

Attribute-based access control models are, as their name implies, based on attributes. Attributes can include the attributes of a person, resource or environment. One of the most widely known example of ABAC that is based on person attributes is the Captcha, as shown in Figure 1.1. Captcha is an acronym that stands for "Completely Automated Public Turing test to tell Computers and Humans Apart" and control access based on "human" attributes, the idea being that only humans can solve Captcha [166]. Resource attributes can also be used to restrict access. Web sites that customize their display based on the type of browser or applications that restrict remote connections to certain protocols are examples of ABACs that are based on resource attributes. Finally, examples of ABAC that are based on environmental conditions can be found in systems that require users to re-authenticate after a

given period of time. In these cases, time is the environmental condition upon which access control is based.

1.3.4 RBAC

Role-based access control (RBAC) models derive from both MAC and ABAC models. Indeed, similarly to MAC, access to resources in RBAC is granted by system administrators and not by owners of resources. In RBAC models, however, access is granted based on the role(s) a user owns. Typically, RBAC models are capability-based systems where capabilities are granted to roles and roles are granted to users. In that sense, RBAC models can be seen as a specialization of ABAC models where only role attributes are supported. In the context of Web applications, RBAC models and their variants are the most widely adopted form of access control models and will be the focal point of this thesis.

RBAC models [143], have been widely adopted by the industry, in part because RBAC models simplify user management. Instead of managing a large number of user-privileges relationships, administrators can grant users pre-defined sets of privileges, called *roles* [35].

In its simplest form, an RBAC model comprises users U, roles R and privileges P and can be defined based on two binary relations:

1. $UA \subseteq U \times R$: User assignments, a *user* is assigned a set of *roles*

2. $PA \subseteq R \times P$: Privilege assignments, a *role* is assigned a set of *privileges*

However, access control models, as their name implies, are only models. Implementation details are left to developers. Consequently, we often observe a gap between theoretical access control models, as found in the literature, and implemented models, as found in real-life applications. In the context of this thesis, we adopted a reverse-engineering approach to the analysis of access control models, as implemented *in practice*.

Indeed, our study of some of the most popular (millions of users) open source Web applications revealed that real-world Web applications are rarely based on formal, well specified and well documented access control models. In several cases, access control models were retro-fitted in the application and implemented in a more or less ad-hoc manner. None of the investigated applications had formal or even informal security specifications and documentation about their access control model was often out of date, erroneous or non-existent. Our goal is to provide support to developers who need to analyze access control models and identify access control flaws in this sub-optimal context.

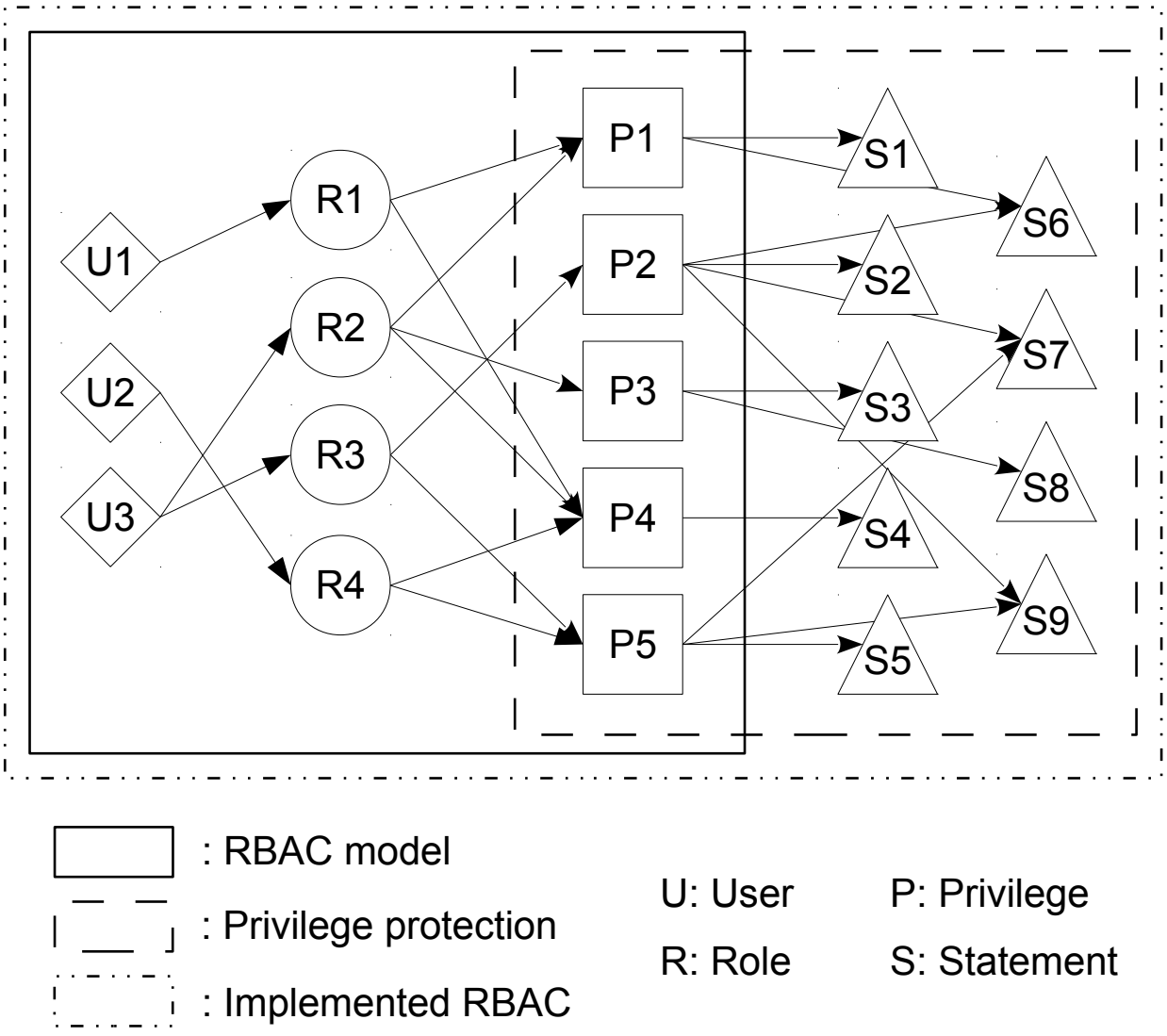


Figure 1.2: A representation of RBAC models

1.4 Role-based access control models in practice

Conceptually, basic RBAC models are very simple. They simply add an abstraction layer between users and privileges, called roles, and simplify privilege management by allowing administrators to grant or revoke privileges to groups of users sharing the same role instead of individual users. In RBAC literature, however, implementation details are often omitted. It is left to developers to implement RBAC infrastructures and enforce appropriate privilege checks in the code.

Figure 1.2 shows a schematic representation of RBAC models, as implemented in practice. Diamonds, circles, squares and triangles represent users, roles, privileges and statements respectively. The solid box encloses users, roles and privileges. It represents basic RBAC models, as described in the literature. In the context of Web applications, the solid box coincides with user administration interfaces, where administrators can create and delete users, assign them roles and manage the privileges that are granted to every role. In the context of this thesis, we will refer to this box as the RBAC model.

The dashed box encloses privileges and statements. It captures the relation between privileges and code statements that is induced by privilege checks. In Figure 1.2, a privilege p maps to a statement s if a privilege check for p is executed prior to executing s. In the context of this thesis, we will refer to this box as $PP \subseteq P \times S$: the privilege protection, where a *privilege* protects a set of *statements*.

Listing 1.1 shows an example of a typical privilege check. The `user_can` function takes a privilege as parameter and returns a Boolean value representing whether or not the current user owns the privilege. In this particular case, the message is only printed if the user owns the `edit_post` privilege and thus statement 2 is *protected* by the `edit_post` privilege.

```
1 if (user_can('edit_post'))
2     echo 'You can edit a post';
```

Listing 1.1: Example of a typical privilege check

RBAC literature gives very few guidelines regarding the implementation of privilege protection. It is left to developers to:

1. Determine which statements are security-sensitive.
2. Design the privileges that should protect security-sensitive statements.
3. Implement privilege check routines.

4. Prevent unauthorized accesses to security-sensitive statements with appropriate privilege checks.

Fortunately, as we shall discuss in further sections, common practices for the implementation of privilege protection emerged over time, facilitating the development of automated privilege protection analysis tools.

Finally the dotted-and-dashed box encloses users, roles, privileges and statements. It represents the statements of the application any given user can access at runtime with respect to his roles and privileges. For the rest of this thesis, we will refer to the graph in this box as the *implemented* RBAC model. Observe that reasoning about the implemented RBAC model requires knowledge about *both* the RBAC model and the privilege protection. In practice, verifying security properties solely at the level of the RBAC model is of little use. Indeed, in practice, any security property that holds on a given RBAC model can be violated by a missing, misplaced or wrong privilege check.

Sadly, while there is a wealth of literature about RBAC models and their variants, comparatively few studies target the reverse-engineering, modelling and testing of privilege protection. The goal of this thesis is thus three-fold: 1. present a novel approach for reverse-engineering of privilege protection, 2. investigate modelling techniques that will help administrators reason about the *implemented* RBAC and privilege protection models, and 3. define novel analyzes for the automatic detection of access control flaws in Web applications all the while keeping the amount of work that is required from developers to a minimum. In other words, we not only ensured that the presented work is novel from a scientific point of view, we also ensured that it is suitable for use by real-world developers and powerful enough to detect unknown security flaws in major and widely popular Web applications.

CHAPTER 2

RESEARCH PROCESS AND ORGANIZATION OF THE THESIS

This chapter briefly presents the papers that constitute the core of this thesis and highlights the underlying long-term research process. In summary, the core chapters of the thesis can be divided in three sections: 1. extraction of the privilege protection model, 2. investigation and comprehension of reverse-engineered RBAC and privilege protection models with FCA and 3. automated identification of access control vulnerabilities.

2.1 Reverse-engineering privilege protection models

Privilege checks restrict access to sensitive parts of an application. In RBAC models, the configuration of privilege checks give rise to the privilege protection model, that maps privileges to source code statements. In open source application, however, privilege protection models are rarely specified or documented and source code often is the only source of reliable information.

Chapter 5 presents our first release of the Security Pattern Traversal analysis (SPT), the original algorithm by which we reverse-engineer privilege protection model (see Figure 1.2) of a system. In the context of this paper, we show how it can reverse-engineer the privilege protection model of Moodle, a course-management system, counting more than a hundred privileges. We further show how privilege protection model can help identify security-sensitive sections of an application and supply developers with an overview of the implemented RBAC.

In chapter 7, we extend SPT with an inter-procedural analysis that tracks the propagation of privilege checks in a system. Indeed, investigation of real world Web applications revealed how the return values of privilege checks are often stored in variables that are propagated in the system only to be verified later. By tracking the propagation of privilege checks, this extended version of SPT is able to extract more precise privilege protection models. For the sake of conservativeness, SPT favors false negatives (statements wrongly reported as unprotected) over false positives (statements wrongly reported as protected). The analysis that is presented in chapter 7 thus aims at reducing the number of false negatives without introducing false positives. Chapter 7 highlights the technical challenges we overcame and compares four variants of the algorithm: two intra-procedural and two inter-procedural versions.

2.2 Investigation and comprehension of reverse-engineered access control models

As mentioned earlier, reasoning about the *implemented* RBAC model requires reasoning about both the RBAC and the privilege protection models. In chapter 9, we explore how Formal Concept Analysis (FCA) can help administrators of Web applications in this task.

In the context of Web applications, RBAC models (user assignment and privilege assignment relations) can usually be extracted from administrator interfaces. Privilege protection models, on the other hand, can be extracted using SPT. In both cases, however, the extracted models might be difficult to interpret due to their size. Indeed, modern Web applications often define hundreds of privileges and count several thousands lines of code. In this context, we explored how FCA can help visualizing and investigating reverse-engineered RBAC and privilege protection models.

On the one hand, FCA extracts concept lattices from formal contexts, representing binary relations between objects and attributes. On the other hand, RBAC models and privilege protection models are defined based on binary relations: users have roles, roles own privileges and privileges protect statements. In the context of this project, we showed how FCA of the RBAC model reveals implicit role hierarchies, that are hidden from administrators and how FCA of the privilege protection reveals implicit constraints between privileges, that can induce misunderstandings and vulnerabilities. Moreover, we explained how FCA highlights impacts of role re-definitions, helps assessing the functionalities a particular user can access or ensuring that certain functionalities cannot be executed by a single user (separation of duties).

2.3 Identification of security flaws in access control models

FCA of reverse-engineered RBAC and privilege protection models gives an overview of the protection in a system and is useful to reason about high level security properties. Such high level properties are typically more relevant to administrators of a system. Developers, on the other hand, are usually more concerned by lower-level, code-centric security properties, such as missing or erroneous privilege checks that lead to security flaws. Chapters 11 to 15 are dedicated to automated identification of security flaws in implemented access control models.

Contrary to other types of security vulnerabilities, determining the presence or the absence of access control vulnerabilities is often a matter of interpretation and judgement. For example, one might have to answer questions like: “should role X have access to data Y?” In theory, such questions should be answered with the help of RBAC model specifications, which are almost never available in practice. We therefore developed heuristics to automati-

cally detect access violations in the absence of security specifications. To this day, three main strategies have been investigated:

1. **Privileged resource identification using privileged hyperlinks.** Web applications usually hide hyperlinks to privileged resources from unprivileged user. Forced browsing vulnerabilities occur when hidden privileged resources are directly accessible through their URL. In order to detect forced browsing vulnerabilities, we developed a static analysis approach that contrasts pages that are accessible through visible hyperlinks with pages that can be accessed directly through their URLs. This project is presented in chapter 11.

2. **Identification of semantic errors.** In access control models, privileges are expected to be semantically related to the action they protect. For example, it is expected that a *download* privilege protects the action of downloading a file. When this semantic link is broken, semantic errors occur. In this project, we used Latent Dirichlet Allocation to model fragments of PHP applications as a distribution of latent topics. Using a logistic regression model, we were able to associate latent topics to privileges and to further identify fragments for which the latent topic distribution did not match the enforced privilege. Doing so, we were able to report semantically wrong privilege checks in the system under study. This project is the subject of chapter 13.

3. **Uncovering access control flaws with security-discordant clones.** Software clones are syntactically similar fragments of code. In the context of this study, we hypothesized that clones usually perform similar operations and that they should be protected by similar privileges. Investigation of security-discordant clones, that are similar fragments of code that are protected by different privileges, revealed several novel security flaws in the investigated applications. Chapter 15 is dedicated to this project.

CHAPTER 3

SECURITY PATTERN TRAVERSAL ANALYSIS

This chapter introduces Security Pattern Traversal (SPT) analysis, the cornerstone of this thesis. SPT is the analysis that we use to extract privilege protection (see Figure 1.2) from systems. The first release of SPT was presented by Letarte et al. [100] and could only process simple admin/user access control models. Chapters 5 and 7 of this thesis extend this approach to support arbitrary large numbers of privileges and track the inter-procedural propagation of privilege checks through variables and parameters. This chapter introduces the fundamental concepts of SPT analysis that are needed to fully understand this research work.

As its name suggests, Security Pattern Traversal analysis tracks the propagation of particular security properties, that are the result of the traversal of a security pattern. Given any program point, SPT tracks the results of all security patterns that have been traversed at the time this particular program point is reached. In the context of RBAC models, security patterns are designed to capture privilege checks and SPT will thus track the results of all privilege checks that were traversed before a statement is executed.

For the sake of simplicity, the following section introduces a running example that illustrates how SPT works.

3.1 Running example

SPT is a static analysis that models the traversal of security patterns. In the context of this thesis, security patterns capture the privilege checks in a system. In all of the investigated applications, privilege checks are boolean: they verify a privilege and succeed if the current

```
1 echo 'Hello';
2 if (user_can('edit_post'))
3     echo 'You can edit a post';
4 else
5     echo 'You cannot edit a post';
6 echo 'Goodbye';
```

Listing 3.1: Running example for security pattern traversal analysis.

user has the privilege and fail otherwise. Consequently, whenever an execution path traverses a security pattern, the result is either true or false and represents whether or not the current user has the verified privilege.

For example, Listing 3.1 shows a simple snippet of PHP code with a privilege check (a call to the `user_can` function) at line 2. In this case, the privilege check returns a Boolean value that indicates whether the user owns the `edit_post` privilege. If the check succeeds, the user has the privilege and the statement at line 3 is executed. Otherwise, the statement at line 5 is executed. In both cases, the statement at line 6 is finally executed and the program terminates. In this particular case, SPT would report that *all* execution paths that reach line 3 succeeded a check for the `edit_post` privilege, *all* execution paths that reach line 5 failed a check for the `edit_post` privilege and that *some* execution paths that reach line 6 succeeded the check while *some* did not.

In order to compute such results, SPT: 1. models privilege checks as security patterns, 2. models the program as model checking automatons and 3. performs reachability analysis on the extracted automatons. In the following sections, we will cover these steps in more details. Finally, section 3.5 highlights how SPT differs from and outperforms related approaches such as slicing or branch dominators.

3.2 Modeling privilege checks as security patterns

Privilege checks are application-specific constructs that assert whether the current user owns sufficient privileges to access security-sensitive parts of the application. If the check succeeds, security-sensitive statements are executed. If the check fails, the execution might stop, an error might be signaled to the user or the execution might continue. Listings 3.2 to 3.6 show examples of security checks in different Web applications.

It is interesting to observe that while privilege checks are application specific, they share several similarities:

1. They are syntactically stereotyped constructs.

2. The permission to check is a string that is passed as a key or a parameter.

3. They are all boolean (e.g. succeed or fail).

Figure 3.1 shows the annotated CFG resulting from our running example, as presented in Listing 3.1. Nodes are annotated with statement numbers and there is one *grant_edit_post* edge and one *revoke_edit_post* edge to represent the two possible outcomes of the privilege check at line 2.

```
if ($_SESSION["admin"] == "yes")
    //Code protected by the admin privilege
```

Listing 3.2: Security check in PHP Calendars

```
if (getUser()->authorise('core.manage', 'com_plugins'))
    // Code protected by the core.manage privilege of the com_plugin module
```

Listing 3.3: Security check in Joomla!

```
if (has_capability('mod/workshop:submit'))
    // Code protected by the mod/workshop:submit privilege

require_capability('mod/workshop:submit');
// Exits if the user does not have the mod/workshop:submit privilege
```

Listing 3.4: Security checks in Moodle

```
if($wgUser->isAllowed('createpage'))
    // Code protected by the createpage privilege

if($wgUser->isAllowedAll('createpage', 'edit'))
    // Code protected by both the createpage and edit privileges

if($wgUser->isAllowedAny('createpage', 'edit'))
    // Code protected by the createpage or edit privilege
```

Listing 3.5: Security checks in Mediawiki

```
if(current_user_can('manage_options'))
    // Code protected by the manage_options privilege

if(has_cap('manage_options'))
    // Code protected by the manage_options privilege
```

Listing 3.6: Security checks in Wordpress

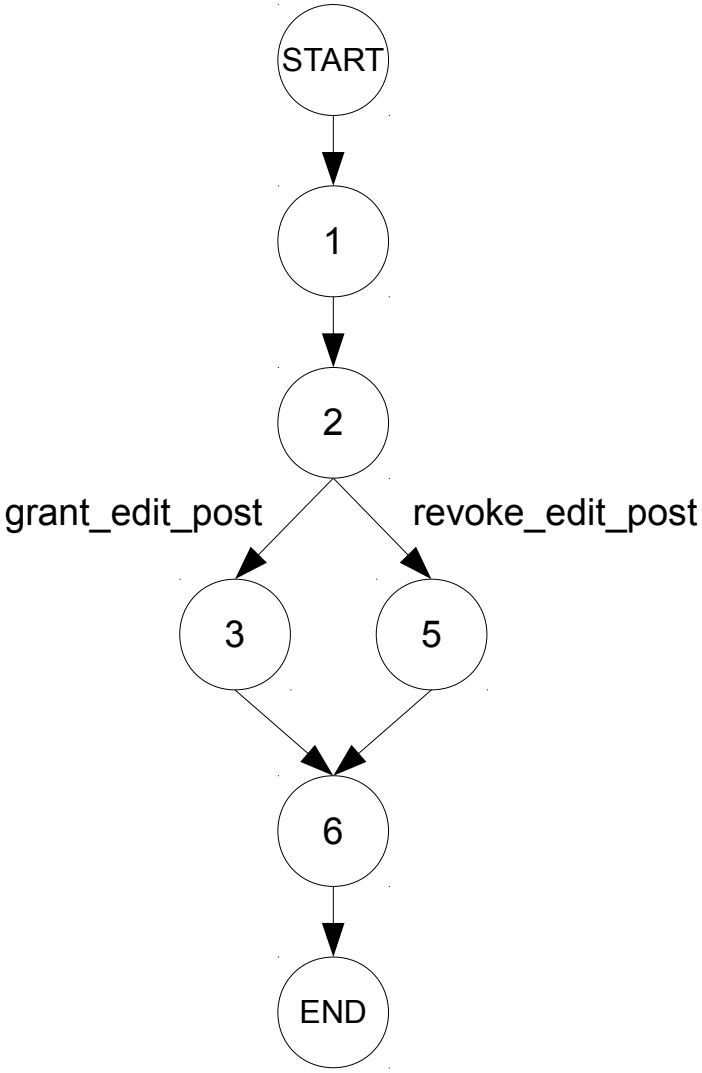


Figure 3.1: The annotated CFG that represents the snippet of code in Listing 3.1

The fact that privilege checks are syntactically stereotyped not only facilitates code reviews, it also greatly simplifies their modeling as security patterns. From an implementation point of view, this allows us to model privilege checks as Abstract Syntax Tree (AST) patterns, that can easily be detected using AST visitors [56]. Given the AST of an application, application-specific visitors are designed to detect specific subtrees in the AST that represent the application's privilege checks. To give the reader an idea, approximately 20 to 30 lines of Java code are usually sufficient to implement the visitors that detect privilege checks in an application, given that the privilege check routines are known.

Security pattern traversal analysis, similarly to other program analysis approaches, does not operate directly on the AST of a program. Further abstraction steps are required. SPT operates on model checking automatons, that represent the control-flow, together with the security patterns of a program.

3.3 Modeling programs as security automatons

The first step towards the extraction of security automatons is to model programs as a Control Flow Graphs (CFG), annotated with security properties:

$$CFG = (V_{CFG},\ E_{CFG}) \tag{3.1}$$

where the CFG has a single *entry* node $v_{in} \in V_{CFG}$ and a single *exit* nodes $v_{out} \in V_{CFG}$. Nodes in V_{CFG} can be of type *generic*, *entry*, *exit*, *call_begin* and *call_end*. Nodes of type *generic* represent any kind of statement and are involved in intra-procedural control flow; nodes of type *call_begin*, *call_end*, *entry*, and *exit* are used in inter-procedural control flow. Conceptually, a *call_begin* node represents the point in a *caller* right before the flow of control is transferred to a *callee*. Similarly, a *call_end* node represents the point in a *caller* right after the flow of control is transferred back from the *callee* to the *caller*. Thus, nodes of type *call_begin* and *call_end* are paired: to each *call_begin* corresponds exactly one *call_end* and vice-versa. The same goes for *entry* and *exit* nodes that correspond to the entry and exit points of a function.

Edges in E_{CFG} can be of type *generic*, $grant\_p_i$, $revoke\_p_i$, *call*, or *return*. Edges of type *generic* represent intra-procedural transfers of control. Edges of type $grant\_p_i$ represent intra-procedural transfers of control that grant the privilege p_i (e.g. the check for the privilege p_i succeeded). Likewise, $revoke\_p_i$ edges revoke the privilege p_i (e.g. the check for the privilege p_i failed). Finally, edges of type *call* and *return* represent inter-procedural control flow links. Edges of type *call* link *call_begin* to *entry* nodes while edges of type *return* link *exit* to *call_end* nodes. Edges of type *grant* and *revoke* are produced based on the information that

is returned by security pattern visitors.

Starting from this annotated CFG, SPT performs a last abstraction step by rewriting the CFG to multiple model checking automaton, each representing one security property. For example, Figure 3.2 shows the automaton resulting from the rewriting of the CFG in Figure 3.1 for the *granted_edit_post* property. Each state is now labeled with a pair (s, p) representing the statement id and the value of the security property (0 stands for false, 1 stands for true) respectively. At the beginning of the program, represented by the start state, no privilege has been checked so $(1, 0)$ is the only state that can be reached. Statement 1 has no influence over the security property so $(1, 0)$ transitions to $(2, 0)$ and $(1, 1)$ transitions to $(2, 1)$. Statement 2 is a privilege check. If the check succeeds, the *granted_edit_post* property becomes true and statement 3 is executed. Hence, all transitions from statement 2 to

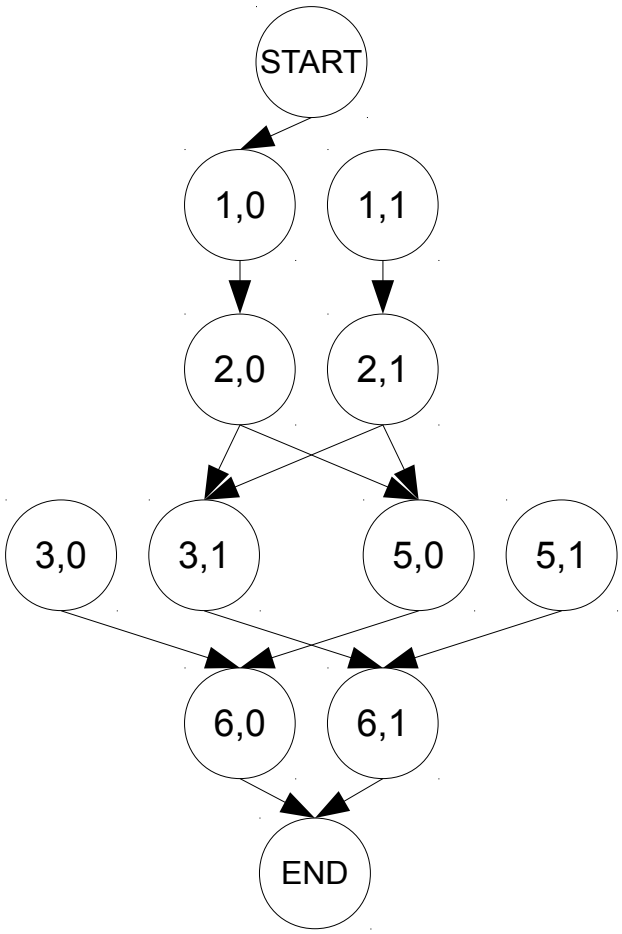


Figure 3.2: The snippet of code in Listing 3.1 represented as a model checking automaton that is suitable for SPT analysis

statement 3 go to state $(3, 1)$. If the check fails, the privilege is denied, the *granted_edit_post* property remains false, and statement 5 is executed. Hence, all transitions from statement 2 to statement 5 go to state $(5, 0)$. Neither statements 3, 5 or 6 are security checks and the remaining transitions do not alter the security property. Once the automaton is built, a reachability analysis from the start state will reveal the statements that are executable together with the possible values of the security property. In the context of Figure 3.2, it is easy to observe that the $(3, 1)$ state is the only reachable state for statement 3 and thus that all execution paths that reach statement 3 have the *granted_edit_post* property set to true.

```
1 function edit() {
2     // Body of the function
3 }
4
5 echo 'Hello';
6 if (user_can('edit_post'))
7     edit();
8 else
9     echo 'You cannot edit a post';
10 echo 'Goodbye';
```

Listing 3.7: Running example, augmented with a function call.

3.4 From simple to complex security patterns

Up to now, for comprehension considerations, our running example consisted of a very simple snippet of code without function calls. At its core, however, SPT is an inter-procedural, *security-sensitive* analysis. Security pattern traversal analysis is security-sensitive in the sense that, contrary to context-sensitive analysis that distinguishes *every* calling contexts, it only distinguishes calling contexts with different values for the security property. Listing 3.7 shows our running example, augmented with a function call. Since the `edit` function is only called at statement 7, from a context where the *granted_edit_post* property is true, SPT would report that the `edit` function is only reachable with the *granted_edit_post* property set to true.

In order to perform inter-procedural, security-sensitive analysis, SPT augments the labels of states in the automaton with context information and introduces call and return transitions. Figure 3.3 shows the automaton that corresponds to the code of Listing 3.7. Notice that labels are now triples (s, c, p) representing the statement id, the value of the security

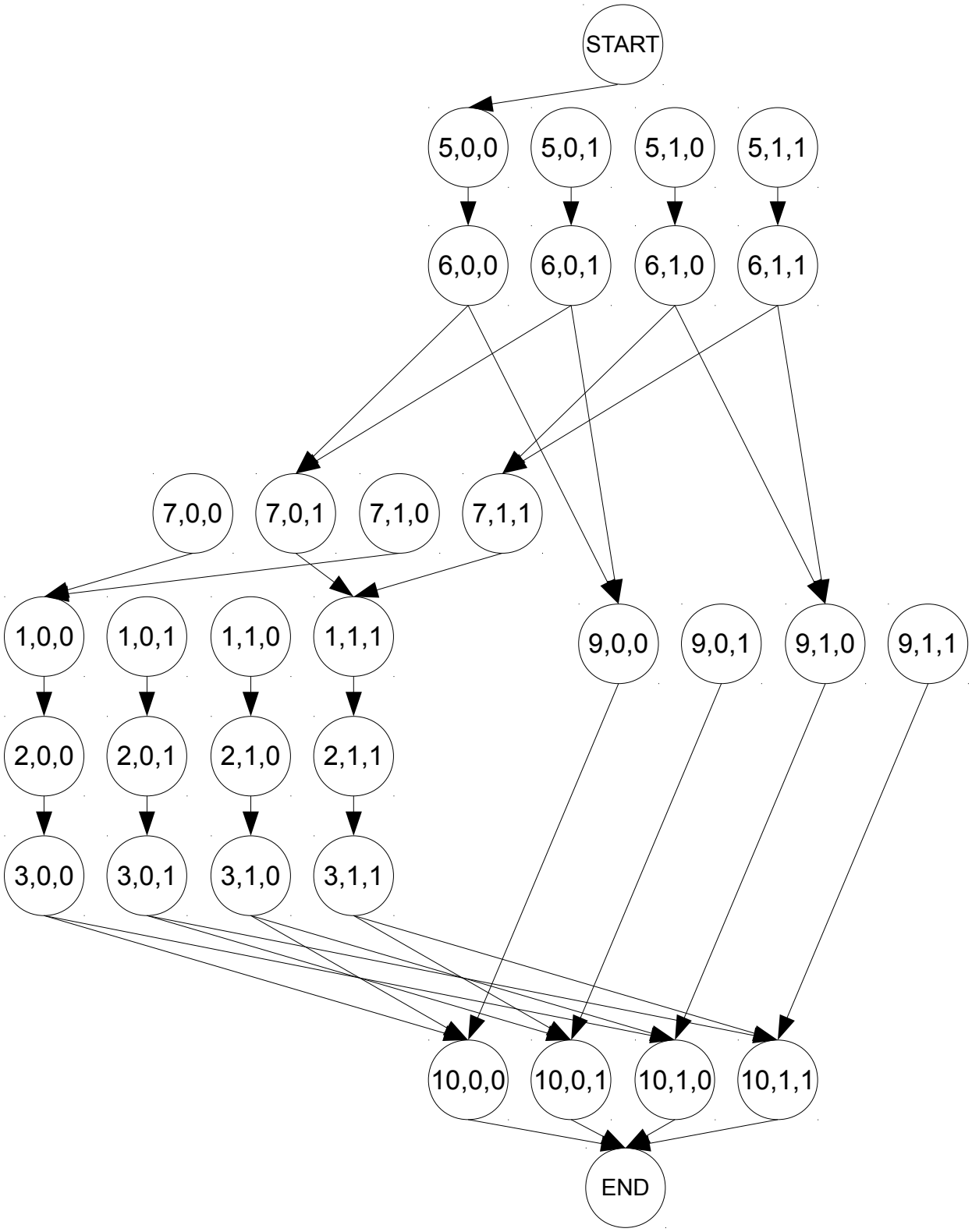


Figure 3.3: The snippet of code in Listing 3.7 represented as an inter-procedural model checking automaton

property of the context and the value of the security property of the statement respectively. Apart from the augmented labels, the inter-procedural automaton of Figure 3.3 also introduces call and return transitions, as illustrated in transitions from states 7 to 1 and states 3 to 10 respectively. Call transitions link a *call_begin* state to an *entry* state. When a call occurs, the value of the security property at the *call_begin* state becomes the value of the security property of both the context and the statement of the *entry* state. Hence, call transitions from states $(7, 0, 0)$ and $(7, 1, 0)$ both lead to $(1, 0, 0)$. Similarly, call transitions from states $(7, 0, 1)$ and $(7, 1, 1)$ both lead to $(1, 1, 1)$.

Return transitions are less straightforward to model. When an *exit* state is reached, SPT must return the flow of control to the *call_end* state with: 1. a security value for the context that matches that of the *call_begin* state that induced the call and 2. a security value for the statement that matches that of the *exit* state. Since this information is not available at the time the automaton is being built, SPT conservatively creates every possible return transitions between *exit* and *call_end* states, as illustrated by transitions between states 3 and 10 in Figure 3.3. Spurious return transitions are eliminated during reachability analysis through the use of assignments on call transitions and guards on return transitions. We refer the interested reader to [100] for the complete reachability algorithm. Note that the reachability algorithm has a complexity that is $O(n)$. In summary, this approach is linear over the number of statements without loss of precision nor security information [100], whereas typical context-sensitive approaches achieve similar results with combinatorial complexities.

Up to now, our examples only comprised one simple privilege check. Privilege checks, however, can be used in arbitrarily complex predicates:

```
1 if (user_can('read') || ext_func() && user_can('read_own'))
2   read_file();
3 else
4   error('Sorry, you cannot read that file.');
```

Listing 3.8: A complex predicate containing security checks

In the case of Listing 3.8, if the predicate at line 1 is true, the user either has the `read` privilege, the `read_own` privilege or both. Conversely, if the predicate is false, the user certainly does not own the `read` privilege and does possibly not own the `read_own` privilege. At the automaton level, such uncertainty is translated as *possibly_grant* and *possibly_revoke* transitions. In our experimental setup, AST visitors detect application-specific privilege checks and annotate the CFG with *grant*, *possibly_grant*, *revoke* and *possibly_revoke* edges in accordance with De Morgan's laws. Specifically, our implementation of SPT can process

Table 3.1: Syntax directed definitions to compute the privileges that are definitely and possibly activated after the traversal of a Boolean predicate.

PRODUCTIONS	SEMANTIC RULES
$S \rightarrow if(E)$	$S.true.def = E.true.def$ $S.true.pos = E.true.pos$ $S.false.def = E.false.def$ $S.false.pos = E.false.pos$
$E \rightarrow E_1 \mid\mid E_2$	$E.true.def = \emptyset$ $E.true.pos = E_1.true.def \cup E_1.true.pos \cup E_2.true.def \cup E_2.true.pos$ $E.false.def = E_1.false.def \cup E_2.false.def$ $E.false.pos = E_1.false.pos \cup E_2.false.pos$
$E \rightarrow E_1 \;\&\&\; E_2$	$E.true.def = E_1.true.def \cup E_2.true.def$ $E.true.pos = E_1.true.pos \cup E_2.true.pos$ $E.false.def = \emptyset$ $E.false.pos = E_1.false.def \cup E_1.false.pos \cup E_2.false.def \cup E_2.false.pos$
$E \rightarrow \neg E_1$	$E.true.def = E_1.false.def$ $E.true.pos = E_1.false.pos$ $E.false.def = E_1.true.def$ $E.false.pos = E_1.true.pos$
$E \rightarrow$ **check**	$E.true.def =$ **check**$.privilege$ $E.true.pos = \emptyset$ $E.false.def = \neg$**check**$.privilege$ $E.false.pos = \emptyset$

arbitrarily complex Boolean predicates with the help of the syntax directed definitions (SDD) that are presented in Table 3.1. The production rules in Table 3.1 define a simple grammar for conditional expressions, boolean predicates and privilege checks. In the SDD, nonterminals have four attributes: $true.def$, $true.pos$, $false.def$, $false.pos$ representing the set of security properties that are definitely and possibly (de)activated when the predicate evaluates to true or false respectively.

Privilege checks, however, can also appear outside predicates. For example, the result of a privilege check can be stored in a variable that gets propagated through the system. Consider the example of Listing 3.9.

In this case, when the `read_file` function is executed at line 9, line 2 verifies the `$privilege` variable, that contains the result of the `user_can('read_own')` privilege check. Thus, if the call to `read_special_file` at line 3 is executed, the current user owns the `read_own` or the `read_all` privilege. Our implementation of SPT handles such complex cases. The algorithms that are employed to inter-procedurally track privilege checks through PHP applications are presented in chapter 7.

3.5 Differences with related approaches

Security pattern traversal analysis is very similar in nature to other static analysis approaches, especially slicing and dominator analysis. In this section we will compare SPT to these two approaches and highlight the major differences between them.

3.5.1 SPT vs slicing

Given a point of interest in a program, called the slicing criterion, program slicing aims at identifying all the instructions that can affect the slicing criterion. The slicing criterion C

```
1 function read_file ($privilege) {
2   if($privilege || user_can('read_all'))
3     read_special_file ();
4   else
5     error('Sorry, you cannot read that file.');
6 }
7
8 $read_own = user_can('read_own');
9 read_file ($read_own);
```

Listing 3.9: Security checks can be stored in variables that are passed as parameters

typically consists of a pair (p, V) where p is a program point and V is a set of variables. Specifically, program slicing reports the statements upon which the slicing criterion is *control* or *data* dependent. Given a program dependence graph (PDG), where vertices represent statements and edges represent control and data dependencies, a slice comprises all the vertices that can be reached starting from the vertex that represents the slicing criterion and traveling backward in the PDG. This form of program slicing is also called *backward* slicing since it starts from a slicing criterion and travels backward in the PDG. Conversely, if one starts from the slicing criterion and travels *forward* in the PDG, a forward slice, that represents all the statements that the slicing criterion may affect, is computed [159].

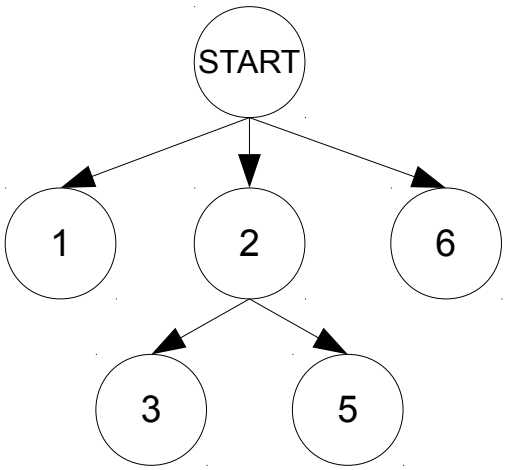


Figure 3.4: The program dependence graph (PDG) that is associated to the snippet of code in Listing 3.10.

```
1 echo 'Hello';
2 if (user_can('edit_post'))
3     echo 'You can edit a post';
4 else
5     echo 'You cannot edit a post';
6 echo 'Goodbye';
```

Listing 3.10: Slicing vs SPT. The forward slice of the privilege check at line 2 is (3,5). SPT reports that the edit_post privilege is always granted when statement 3 is executed, never granted when statement 5 is executed and sometimes granted when statement 6 is executed.

Security pattern traversal analysis and forward program slicing share some level of similarity. Indeed, in both SPT and slicing, there is a notion of control dependency. As we shown earlier, privilege checks can alter the control-flow of a program with respect to the privileges a user owns. Furthermore, SPT also propagates properties in a forward manner, starting from privilege checks and propagating properties along execution paths. SPT, however, goes a step further than forward slicing: it analyses privilege checks and propagates the *result* of the check. For simplicity purposes, we re-copy a previous example below. In Listing 3.10, statement 2 is a privilege check that verifies if the user has the `edit_post` privilege. Figure 3.4 shows the program dependence graph that is associated to the code in Listing 3.10. Edges in the graph represent control dependencies only since Listing 3.10 contains no data dependencies. Considering the PDG in Figure 3.4, if statement 2 was selected as the slicing criterion, its forward slice would be $(3, 5)$. Hence, slicing would reveal that the privilege check at line 2 affects the execution of statements 3 and 5. Similar conclusions could be achieved by investigating the backward slices of statements 3 and 5 respectively. On the other hand, SPT propagates the *result* of the privilege check. Figure 3.5 shows the model checking automaton that represents the code in Listing 3.10, together with the reachable states. Thus, SPT would report that the `edit_post` privilege is always granted when statement 3 is executed, never granted when statement 5 is executed and sometimes granted when statement 6 is executed. In that sense, SPT provides more precise information about the security of a program than slicing.

SPT vs path conditions

Path conditions were developed to enhance the precision of slicing results. As we already explained, slicing reports the statements that may influence (backward slicing) or that may be influenced (forward slicing) by the slicing criterion. Path conditions are defined over program variables and refine slicing results by defining necessary conditions for an influence relation to exist. In other words, given a path condition $PC(x, y)$ where x and y are statements, if the path condition cannot be satisfied, there is definitely no influence from x to y [140, 75]. Path condition approaches typically use constraint solvers to determine whether or not there exists inputs that can satisfy the path condition. While both SPT and path condition approaches perform some form of predicate evaluation (see Table 3.1), the aim is totally different. SPT aims to identify the privileges a user must own to execute a given statement while path conditions aim at pruning slicing results by eliminating spurious control and data dependencies. Some interesting work by Hammer et al. explored how path conditions approaches could be adapted for security assesment [74]. However, they report a complexity of $O(n^3)$ for the context-sensitive version of their algorithm. SPT, on the other hand, performs

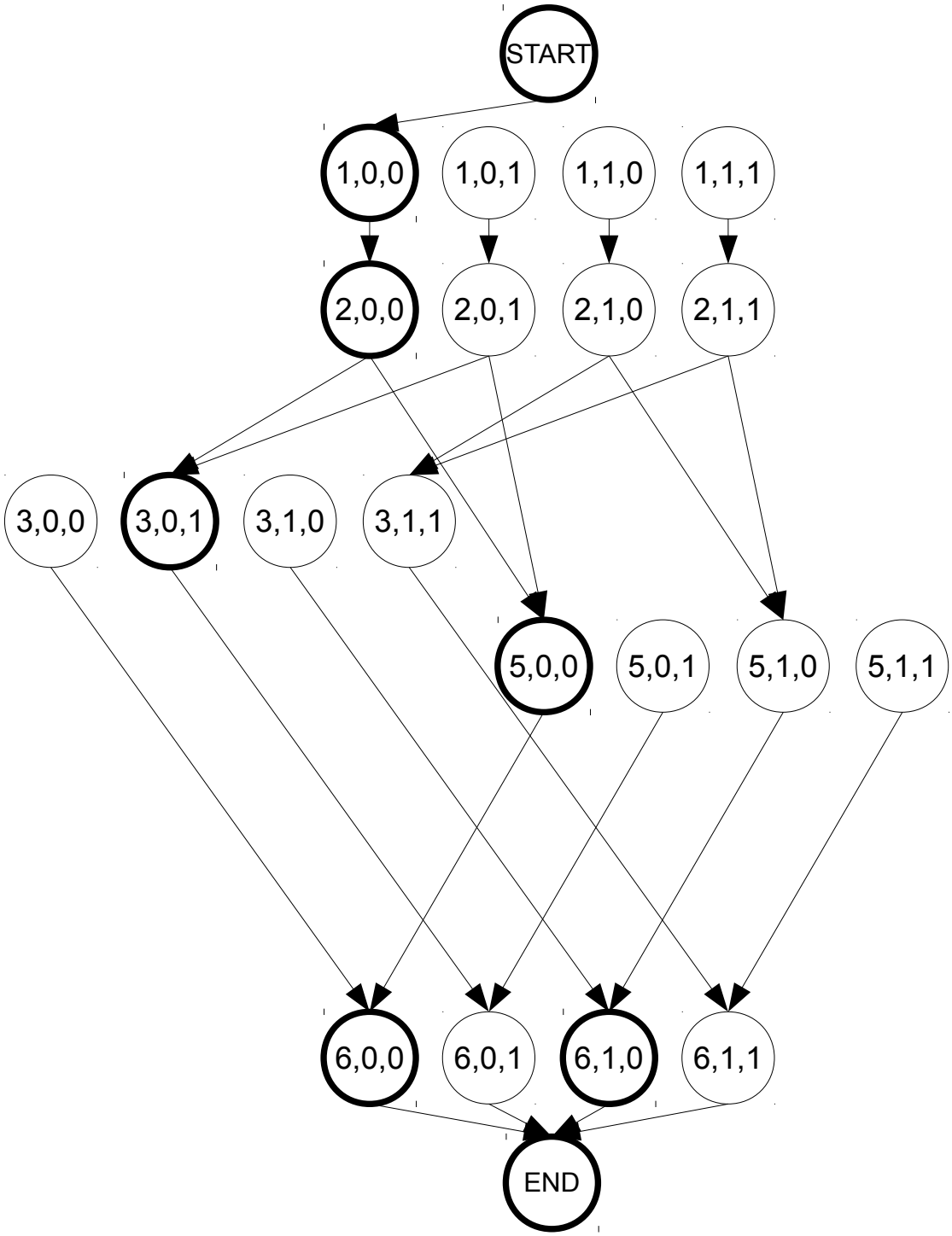


Figure 3.5: The snippet of code in Listing 3.10 represented as a model checking automaton. States in bold represent the reachable states in the automaton.

a security-sensitive analysis that has a complexity of $O(n)$, allowing it to scale to large applications.

3.5.2 SPT vs branch dominators

When a program is represented as a control-flow graph (CFG), we say that node d dominates node n if every path from the entry node of the CFG to n goes through d. Similarly, a branch br *dominates* node n if every path from the entry node of the CFG to n goes through br. Given that security checks are used in control-flow predicates and that the result of security checks are reflected in the branches that are followed, branch dominator analysis might seem a good alternative to SPT analysis. However, consider the snippet of code of Listing 3.11

```
1 switch ($action)
2 {
3 case read:
4   if(user_can('read_post'))
5     ...
6   else
7     error(' Insufficient  privileges ');
8   break;
9 case edit:
10  if(user_can('read_post') && user_can('edit_post'))
11    ...
12  else
13    error(' Insufficient  privileges ');
14  break;
15 default:
16  error('Unsupported action')
17 }
18 echo 'Done with processing';
```

Listing 3.11: Line 18 can only be executed if the user owns the read_post privilege.

It is clear that line 18 is only executed if the user owns the `read_post` privilege. SPT analysis would report this result without problem. Now, consider the control-flow graph that is associated to the snippet of code of Listing 3.11, as represented in Figure 3.6. Neither the $(4, 5)$ or the $(10, 11)$ branch dominates node 18. Consequently, a branch dominator analysis would fail to report that the statement at line 18 can only be executed by users who own the `read_post` privilege. At the inter-procedural level, however, SPT significantly outperforms branch dominators. Consider the following snippet of code:

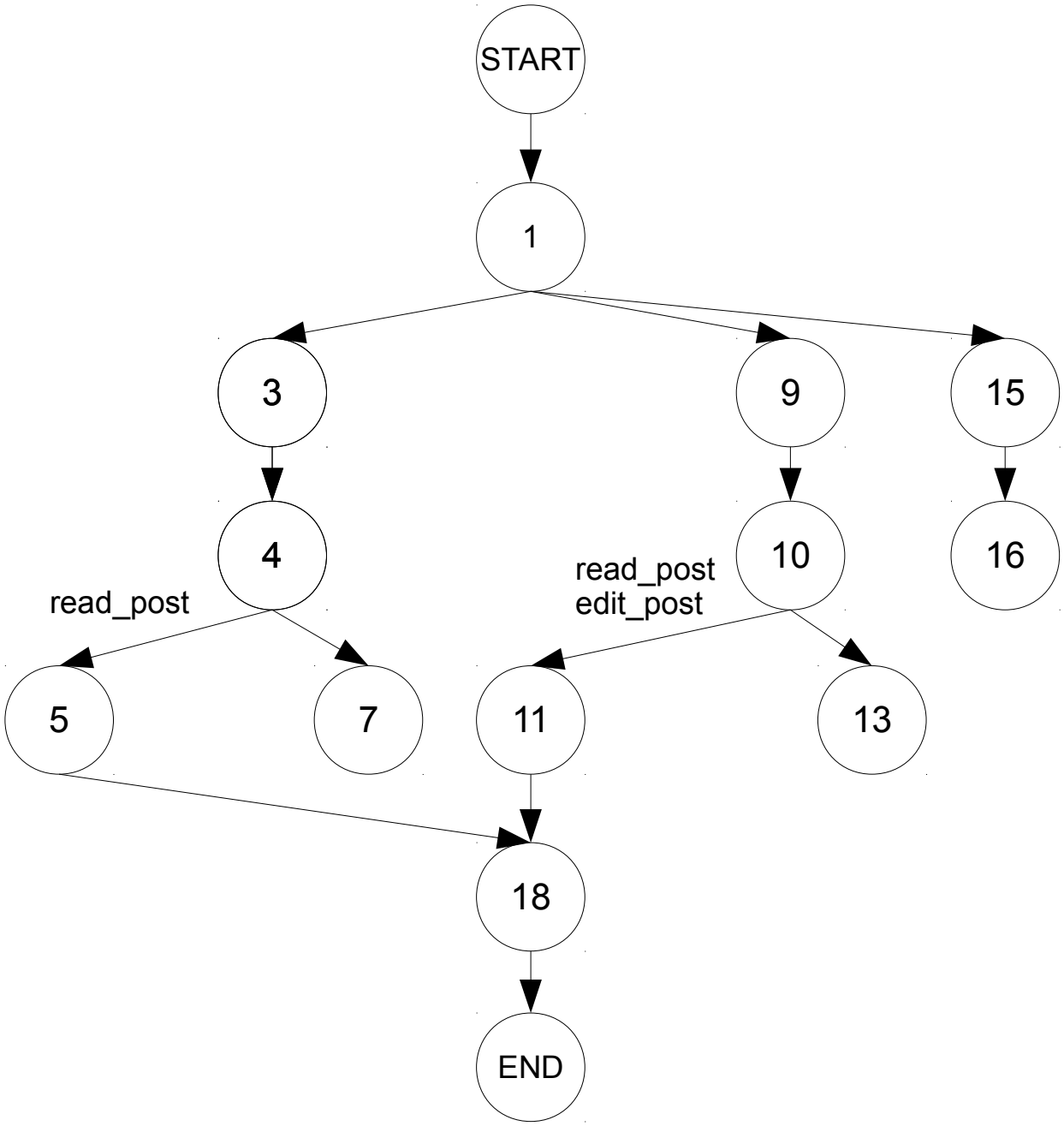


Figure 3.6: Control-flow graph of the snippet of code of Listing 3.11

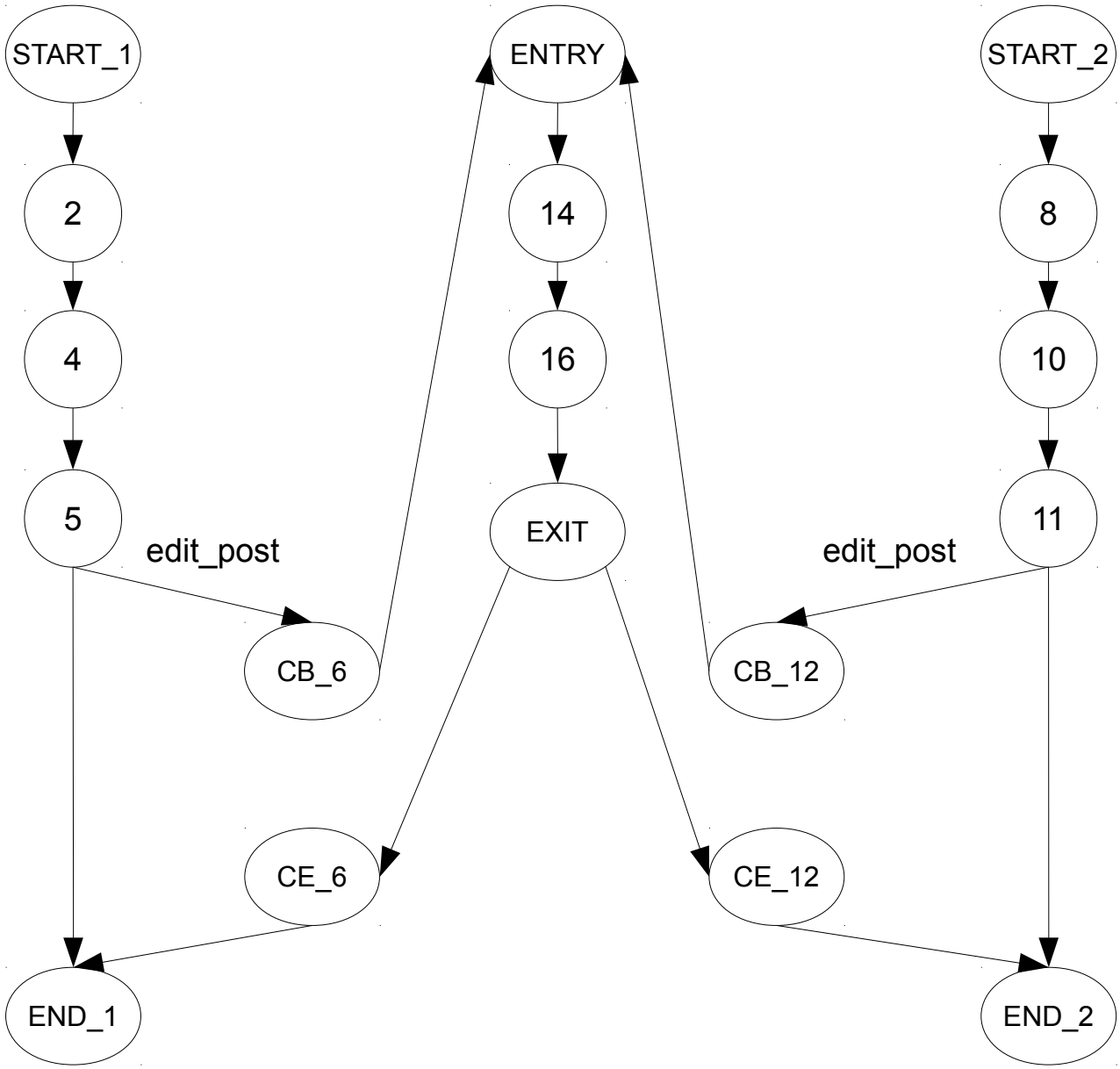


Figure 3.7: Control-flow graph of the snippet of code of Listing 3.12

```
1 // file1.php
2 include 'library.php'
3
4 echo 'This is file 1';
5 if (user_can('edit_post'))
6     edit();
```

```
7 // file2.php
8 include 'library.php'
9
10 echo 'This is file 2';
11 if (user_can('edit_post'))
12     edit();
```

```
13 //library.php
14 function edit() {
15     // Body of the function
16 }
```

Listing: SPT vs branch dominators. At the inter-procedural level, branch dominators might fail to report that a privilege is always granted before a function is called. Here, branch dominators would fail to report that the edit function is protected by the edit_post privilege

Figure 3.7 shows the annotated control-flow graph that corresponds to the previous snippet of code. In the figure, CB and CE stand for call_begin and call_end nodes respectively and represent the points in code right before function call and right after function return. Observe that the `edit` function is called from both `file1.php` and `file2.php`, as represented by edges $(CB\_6, ENTRY)$ and $(CB\_12, ENTRY)$. In both cases, the function is only called if a check for the `edit_post` privilege succeeds. However, no branch of neither checks dominates the entry point of the function. A branch dominator analysis would thus be unable to report that the `edit_post` privilege is always granted when the `edit` function is called. SPT, on the other hand, naturally detects and reports such cases. Overall, SPT extracts more relevant security information than branch dominators, especially at the inter-procedural level.

CHAPTER 4

RELATED WORK

The work that is presented in this thesis spans several fields of computer science. This literature review reflects the diversity of topics covered and is organized around three main themes: 1. program analysis, 2. formal concept analysis and 3. vulnerability detection in Web applications. While relevant literature is reviewed in each of the paper that constitute the core chapters of this thesis, this section also covers topics that we did not have the chance to address in chapters 5 to 15. Several of the reviews presented here are adapted from "related work" sections of chapters 5 to 15.

4.1 Program analysis

In chapter 3, we presented the Security Pattern Traversal (SPT) analysis, a static analysis that propagates the results of security checks throughout programs. We also briefly showed how SPT differs from program slicing and branch dominators. In this section, we discuss these techniques as well as other program analysis approaches that are related to SPT.

4.1.1 Program slicing

In subsection 3.5.1, we described slicing as an analysis that reports the statements upon which a slicing criterion (p, V) is *control* or *data* dependent. We also showed how SPT differs from slicing in the way it analyzes the control dependencies that are induced by security checks. In this section, we briefly review existing slicing approaches.

Program slicing was originally proposed by Weiser [171] who suggested that slicing could be used for debugging tasks. In Weiser's paper, a slice S is defined as a reduced, executable program obtained by removing statements from the original program P such that S replicates parts of the behavior of P, including the faulty behavior. More recent definitions of slicing left out the requirement that a slice must be *executable* and instead focused on reporting the statements that can *influence* the slicing criterion.

Another important distinction must be made between static and dynamic slicing. Original slicing approaches computed slices statically, making no assumptions about the inputs to the program and including all the statements that *may* influence the slicing criterion. Dynamic slicing approaches, that compute slices at run-time based on specific test inputs, were later proposed.

Static slicing

In Weiser's approach, slices are calculated by computing consecutive sets of transitively relevant statements, according to data flow and control flow dependences [159]. In subsequent work, Ottenstein and Ottenstein [127] restated the problem of computing static slices as a reachability problem in a program dependence graph (PDG) representing data and control dependencies, as we described in 3.5.1. Nowadays, the PDG-based approach to computing static slices is the most commonly used [80, 54, 79, 25].

Dynamic slicing

The term "dynamic program slicing" was first coined by Korel et al. in [94]. Contrary to static slicing, in dynamic slicing, the only data and control dependences that are taken into account are those that occur in a specific execution of the program. Dynamic slicing approaches work based on execution traces. In that context, the slicing criterion usually is a triple (*input*, *statement*, *variable*) where *statement* corresponds to a particular occurrence of a statement in the execution trace and *input* defines the inputs that generated the trace [2, 95, 68].

4.1.2 Dominator analysis

In subsection 3.5.2, we discussed how SPT is related to branch dominators algorithms. While dominators find applications in a variety of domains, they are most notably used in optimizing compilers [3]. While the problem of finding dominators can be expressed quite simply using standard data flow equations, they are so widely used in compilers that a vast body of work have been dedicated to the elaboration of more efficient algorithms [98, 78, 12, 64].

4.1.3 Model checking

Remind from chapter 3 that SPT models programs and security checks as model checking automaton. More general model checking approaches for program verification have been presented in [17, 18, 49, 165]. These approaches are arguably more powerful, but also more complex and costly to execute. SPT can be considered as a special case of general model checking approaches that is suitable for problems that can expressed as a reachability problem over a Boolean property.

In [31], authors present a model checking approach, named MOPS, to verify security properties on C programs. In MOPS, security properties are expressed as finite state automatons (FSA), and programs are represented as push-down automatons (PDA). Using a process known as *composition*, MOPS merges the security FSA with the program PDA and produces a *composite* PDA. Using model checking techniques, MOPS determines whether there is a

reachable risky state in the composite PDA. If so, it reports a vulnerability together with the execution path that led to the risky state.
MOPS and SPT mainly differ on two aspects. First, contrary to SPT, MOPS does not evaluate predicates. For example, MOPS would not be able to determine that the following predicate: `user_can(read) && user_can(write)` grants both the `read` and `write` privileges. Second, MOPS models security properties as FSA and programs as PDA. On the one hand, the use of PDAs allows MOPS to be path-sensitive and to supply counter-examples when a security property is violated. On the other hand, the use of FSAs allows for the expression of more complex security properties than SPT. However, these implementation choices come at a price. The complexity of MOPS ranges between $O(SP^2)$ and $O(S^3P^2)$ where S is the number of states in the FSA and P is the number of statements in the control-flow graph (CFG) of the program. SPT, on the other hand, has a complexity that is $O(P)$.

4.1.4 Points-to analysis

Chapter 7 presents algorithms to track the propagation of security patterns trough variables and parameters. These algorithms are strongly inspired from points-to analysis, that associates pointers or heap references to variables and storage locations. In the context of SPT, the goal is to determine to which security patterns a variable can directly or indirectly (through aliasing) points to.
Points-to analysis is not a new subject. However, it has attracted and continues to attract the attention of many researchers. Over the years, several studies have been devoted to the area of alias analysis [24, 172, 44, 154, 133, 77]. However, the focus of most of these studies was static languages like C, C++ and Java. The dynamic nature of the PHP language made the existing analyses difficult to reuse and initiated the work presented in chapter 7. Some recent research also targets points-to analysis for the highly dynamic JavaScript language [50, 155, 85].

4.2 Formal concept analysis

In chapter 9, we show how Formal Concept Analysis (FCA) can be used to investigate reverse-engineered RBAC and privilege protection models. On the one hand, we show, in chapter 1, how RBAC and privilege protection models are defined based on binary relations. On the other hand, FCA takes a binary relation, abstracted as $Object \times Attribute$, as input and identifies clusters of objects that share identical attributes before ordering them in a Galois lattice [59]. Therefore, FCA naturally lends itself to the analysis of RBAC and privilege protection models.

Unsurprisingly, many researchers indeed suggested FCA-based approaches to support the design [117, 46, 126] of RBAC models. The idea behind these approaches is to empower the role engineer with the formalism that is provided by FCA to design better and more robust RBAC models. Using a process that is called *attribute exploration*, role engineers can iteratively define constraints on their RBAC model while the underlying FCA engine takes care of producing an appropriate RBAC configuration.
In the context of the work presented in chapter 9, however, we used FCA to investigate *reverse-engineered* RBAC and privilege protection models. The only work, that we are aware of, that is related to this study used a combination of static analysis and formal concept analysis to mine security-sensitive operations from legacy code [58]. In this work, authors manually identify application specific, security-sensitive, code patterns that are further statically detected with the help of ASTs. Formal concept analysis is then used to identify candidate *fingerprints*, that are sets of security-sensitive code patterns that tend to cluster together in the system. A final step of manual analysis by domain experts is required to select relevant candidate fingerprints and refine them to build a final set of fingerprints. Fingerprints are then matched against the code base to identify security-sensitive operations that are further protected with specific authorization hooks.
Apart from the investigation of access control models, formal concept analysis also finds use in several other areas of software engineering, such as program testing, refactoring, comprehension, maintenance and others [158, 161, 137, 160, 67, 91, 114].

4.3 Vulnerability detection in Web applications

The studies that are presented in chapters 11 to 15 specifically target the detection of access control vulnerabilities in PHP applications. In the following paragraphs, we review fundamental studies about both the detection of vulnerabilities in Web applications in general and in access control models in particular.

4.3.1 Detection of SQLi and XSS

Year after year, SQLi and XSS vulnerabilities continue to sit at the top of the OWASP Top 10 Web application vulnerabilities [131] list. While SQLi and XSS vulnerabilities are not the focus of this thesis, several approaches that were developed for the detection of SQLi and XSS present interesting similarities with SPT.

Taint analysis

In some respects, SPT, when applied to a single privilege, relates to taint analysis. Generally speaking, one refers to *tainted* variables do designate untrusted data that can flow to security-sensitive code. Untrusted variables are therefore tagged as *tainted* until some sanitizing routine sanitizes them. Classic taint analysis is thus a data-flow analysis. Our approach, on the other hand, implements a form of static control-flow taint analysis where privilege checks are analogous to sanitizing routines that sanitize *tainted* execution paths. Taint analysis, both in its static and dynamic forms, has been successfully employed to detect SQLi and XSS vulnerabilities in Web applications.

Classic taint analysis, however, does not make any distinction between sanitizing routines. For example, in the context of SQLi vulnerability detection, a taint analyzer would only report whether or not the untrusted variables have been processed by *some* sanitizing routines before flowing to the SQL interpreter. SPT, on the other hand, differentiates the privilege checks that have been traversed before executing security-sensitive code.

Tripp et al. [163] implemented a static taint analysis for Java (TAJ). Their approach aims at identifying XSS and injection vulnerabilities in Web applications and is able to handle reflective calls, flow through containers and nested taint.

Clause et al. [34] presented a framework (DYTAN) for conservative dynamic tainting. Their framework takes as input a user supplied configuration file, describing the taint analysis to be performed, and instruments x86 binaries accordingly. Execution of the instrumented binaries generates reports according to user's specifications. Interestingly, to our knowledge, they are among the first authors to explicitly address the problem of control flow tainting.

Jovanovic et al. [88, 89] developed Pixy, a static taint analysis for identification of cross-site scripting (XSS) vulnerabilities in PHP Web applications. Pixy uses an inter-procedural and context-sensitive data flow analysis enhanced with literal analysis to detect potential XSS vulnerabilities in PHP scripts. Literal analysis allows Pixy to statically approximate the target of dynamic includes and enhance the precision of their analysis. Chapter 7 discusses how we adapted and implemented this idea in the context of SPT.

Several other data-flow based approaches have been dedicated to the detection of SQLi and XSS vulnerabilities [106, 105, 173, 83, 72, 92].

String analysis

String analysis has also been explored as an alternative to data-flow based approaches for the detection of SQLi and XSS vulnerabilities. String analysis is a technique that tracks string values in a program. For example, advanced string analysis techniques are typically able to

approximate, in the form of regular expressions, all the strings a variable can hold at a given point in the program. Since PHP programs typically output HTML strings, string analysis techniques can be used to approximate all the HTML pages a script can output.
Minamide [116] implemented a PHP string analyzer, based on Christensen et al. [32] implementation for Java, to validate dynamically generated web pages. Wassermann and Su further extended Minamide's approach to detect SQLi [168] and XSS [169] vulnerabilities. While very powerful, string analysis approaches typically suffer from high complexity and execution times.

Model checking

Model checking approaches have also been used for the detection of XSS attacks. The analysis presented in [104], aims at identifying confidential information leaks and XSS attacks using source code model checking. Programmers are first requested to manually add special comments to identify each confidential strings in the application. Their tool then extracts these specifications from source code as well as XSS sanitizing routines and produces model checking automatons that are compatible with *Bandera* [48], a model-checking tool. It is then left to *Bandera* to determine whether there exists XSS vulnerabilities or if confidential strings can flow to untrusted parts of the application.

4.3.2 Detection of access control vulnerabilities

Compared to SQLi and XSS vulnerabilities, detecting access control vulnerabilities often poses an additional challenge: the lack of security specifications. Whereas it is usually accepted that SQLi and XSS vulnerabilities occur when untrusted input data can flow to the SQL interpreter or the JavaScript engine, access control violations do not admit such a clear and generic definition. For example, there is no clear and general rule as to what an administrator *should* be allowed to do in a system. Therefore, access control violations must be interpreted in the light of application-specific security specifications, that define the actions any user and/or role is allowed to perform in the system. However, such specifications are rarely available, especially in open source systems. To circumvent this problem, researchers have adopted two main strategies.
The first strategy is to extract formal access control models from source code and to perform security verifications on the extracted models. On the one hand, extracting formal access control models enables users to formulate and verify potentially complex security properties. On the other hand, users are usually required to manually specify the security properties they wish to verify.

The second strategy is to build heuristics that detect access control violations based on a set of assumptions that are expected to hold for a large range of applications. These approaches are usually fully automated but are limited by their heuristic nature and the assumptions that are made.
The work that is presented in chapters 5 to 9 mostly follow the first strategy while the work in chapters 11 to 15 is mostly related to the second strategy.

Access control model extraction and verification

Alalfi et al. developed a framework for the extraction and verification of access control models in Web applications. In summary, their framework consists in a suite of tools that reverse-engineer access control models to UML representations and perform security verifications over the extracted UML models. In the following paragraphs, we review each of the tools that compose their framework in more details.
The SQL2XMI [6] tool automatically transforms a SQL schema into a UML ER data model. This UML ER model serves as a reference model for subsequent tools.
The PHP2XMI [7] tool instruments PHP applications using the TXL [36] technology to recover dynamic behavior models from dynamic execution traces. The behavior model that is recovered by PHP2XMI is a sequence diagram where users and dynamic pages are represented as lifelines and transitions between pages are represented as messages.
The WAFA [9] tool extends PHP2XMI by including interactions with the database into the sequence diagram that is extracted by PHP2XMI. The database interaction information that is collected by WAFA can be mapped onto the UML ER model of SQL2XMI.
The DWASTIC [10] tool instruments PHP applications to gather coverage metrics that are specifically tailored for Web applications. Metrics include page access, SQL statement and server environment variables coverage ratios.
The PHP2SecureUML framework [11] uses SQL2XMI, PHP2XMI and WAFA to extract secureUML models from dynamic executions of PHP applications. In this context, DWASTIC is used to assess the quality of reverse-engineered secureUML models: the higher the coverage metrics, the higher the quality of extracted models.
Finally, in [5], the authors presented the SecureUML2Prolog tool that transforms a secureUML model to a Prolog-based formal model, that is suitable for verification of access control security properties, such as unauthorized access to privileged resources.
In a similar perspective of access control model extraction and verification, Koved et al. [96] proposed an approach to compute access rights requirements in Java applications. The authors used their tool to construct an Access Right Invocation Graph (ARIG), representing the privilege protection model, as implemented in the code. In [136] they further extended

their approach with taint analysis to detect and automatically protect code that should have privileged access. Their technique also identifies unnecessary and redundant privileged code and tag tainted variables as benign or malicious depending on whether or not they are used in privileged functions.
Wang et al. [167] presented a technique for the automatic reverse engineering of access control models from access control configuration files.
In [73], Hallé et al. proposed to model web applications as state machine and dynamically verify that runtime operations don't violate some pre-defined temporal properties. Similarly, Dalton et al. proposed a tool, called Nemesis [41] that takes a user specified access control model as input and dynamically detects access control violations at runtime.

Heuristics for the detection of access control vulnerabilities

As we previously mentioned, in order to circumvent the lack access control specifications, some researchers developed heuristic approaches to infer specification and automatically detect specific types of access control flaws.
Swaddler [38] automatically learns the relationships between an application's critical execution points and internal states to infer a workflow. It then detects anomalous behaviors by reporting executions that violate the workflow. In subsequent work [51] they enhanced their tool to infer invariants of an application from its execution traces. Invariant violations are then reported at runtime.
In 2011, Son et al. introduced RoleCast [152], a tool that is specifically designed for access control vulnerabilities detection. RoleCast statically infers the privilege checks of a program by analyzing the variables that are usually verified before the execution of security-sensitive operations. In the context of their study, security-sensitive operations are limited to `INSERT`, `UPDATE` and `DELETE` database operations. To mitigate some difficulties that are specifically related to the static analysis of PHP, they convert PHP applications to Java programs and perform several post-processing operations to produce well-formed Java files. RoleCast then processes the Java version of the program and reports a vulnerability if an execution path that lead to a sensitive operation misses a privilege check.
Some other approaches detect access control violations based on the protection level of hyperlinks [156]. The assumption behind these studies is the following: if all the hyperlinks that point to a page are privileged, direct URL access to the page should be denied. Otherwise, a *forced browsing* vulnerability is reported. Work by Sun et al. [156] relies on string analysis techniques [168, 169] to identify forced browsing vulnerabilities in PHP applications. Similarly to SPT, their analysis relies on application-dependent patterns to identify security checks. However, contrary to SPT, their approach requires manual specification of infor-

mation such as: session values, cookie values, request parameter values, database records, variable values and function return values. Chapter 11 shows how we adapted SPT to detect forced browsing vulnerabilities with equivalent precision, but much less manual work and much higher execution speeds (up to 890× faster).

In [176], the authors present Chucky, a tool to identify missing security checks based on the *neighbors* of a function. The idea behind their approach is to first identify, with the help of information retrieval techniques, neighbour functions that share a similar vocabulary with a target function. Chucky then compares security checks in the target function with those of its neighboring functions to identify discrepancies, the idea being that if neighboring functions do perform a check while the target function does not, there might be a missing security check. Interestingly, Chucky is strongly related to the work presented in chapters 13 and 15 that respectively use information retrieval techniques and clone analysis (another form of neighborhood) to identify wrong or missing security checks.

Other researchers also developed approaches to detect access control violations based on dynamic analysis [19, 20]. Applying data-mining techniques to access logs that are collected over a certain period of time, their approach identifies access control misconfigurations, that are small differences between the privileges of a group of users.

On a similar line of thoughts, Das et al. developed the Baaz tool [45] to automatically detect misconfigurations in access control models. Contrary to the approach by Bauer et al. that is based on dynamic analysis, Baaz detects misconfigurations based on static access control policies.

In [57], authors proposed an approach to mine security-sensitive operations from legacy code using a mix of static and dynamic analysis. They first extract static code patterns and dynamic side-effects of operations that are known to be security-sensitive. Both static code patterns and dynamic side-effect constitute the fingerprint of an operation. Then, their tool automatically mitigate security flaws by protecting operations that share similar fingerprints with security-sensitive operations.

Conformance testing and verification

For cases where formal security policies are available, conformance testing and verification approaches can prove very powerful. Conformance testing and verification aims at certifying that the *implemented* access control model is conformant with a given security policy. On the one hand, testing approaches uses test-case generation techniques to derive security test suites from security policies [111, 123, 82, 110, 162, 108]. Some of these approaches guarantee the conformance of the security policy with the implemented program if all tests succeed. On the other hand, verification methods express security policies and their implementations in

formal or logical languages to verify security properties. Examples of verification approaches can be found in [21, 76, 138, 93, 70, 52].

The only work we are aware of that uses static analysis to check conformance of RBAC security policies with their implementations were presented in [134, 124]. In [134] authors presented a method to test conformance between policies and implementations based on Abstract Syntax Trees (AST). However, while the approach is related to ours, they only presented a case study on a very simple policy and a snippet of five lines of code.

The static analysis presented in [124] leverages the J2EE framework specifications to detect inconsistencies between access control policies and their implementations in J2EE applications.

CHAPTER 5

PAPER 1: EXTRACTION AND COMPREHENSION OF MOODLE'S ACCESS CONTROL MODEL: A CASE STUDY

ABSTRACT

Whether for development, maintenance or refactoring, multiple steps in software development cycle require comprehension of a program's access control model (AC model). In this paper, we present a novel approach to reverse-engineer AC model structure from PHP source code. Using an hybrid approach combining static analysis and model checking techniques, we are able to extract AC model structure in a fast and precise way.
An experimental tool was developed to evaluate the presented approach and report AC models using source code coloring. For this case study, Moodle, a medium-scale (approx. 625K lines of code), open-source PHP application with a rich AC model was investigated. Results revealed that, although very complex by design, implemented AC models may comparatively be very simple, suggesting that developers tend to maintain a low complexity level when implementing ACs. Detailed figures and distributions are reported.
We believe the presented tool and approach may help in understanding and evaluating the implemented AC models in Web systems. Discussion of findings, limitations, and further research are presented.

5.1 Introduction

Web applications are present in every area of daily life; we use them to communicate, buy and sell merchandise, gather information, etc.
Among all the web applications we use, many deal with privacy or security sensitive information such as: credit card numbers, addresses or financial data. These applications must therefore implement mechanisms to control access to privacy or security sensitive content. Some good examples of applications that need to implement such mechanisms are: e-commerce applications, e-learning applications, content management systems, forums, or online banking applications.
Access control (AC) models simplify administration of user privileges and are therefore well suited for Web applications, which often have to deal with a large number of users that may

change in time. In fact, AC models are widely used in Web applications nowadays.
Sandhu et al. [143] introduced a terminology to describe Role-based Access Control models (RBAC). As its name suggest, RBAC defines an AC model based on user roles. For the sake of clarity, we borrowed some relevant terms from their terminology to describe AC models:

- "User" refers to any person who directly interacts with a Web application through its user interface.
- "Object" refers to a passive entity that contains or receives information. In the context of this paper, "objects" will refer to sections of code and statements.
- "Capability" is the name of an interactive action a "user" may perform on an "object". For example, *moodle/blog:view* is a "capability" in Moodle.
- "Permission" represents the allowed level of interaction for a given "capability"
- "Access control" refers to the process of determining whether a "user" can perform the action described by a "capability" according to their "permission".

Several software engineering objectives may benefit from extracting and understanding the AC model of an application.

- Identification and correction of AC model defects and vulnerabilities. Indeed, understanding which "permissions" are required to access an "object" helps to detect vulnerable AC.
- Modification of existing AC model structures. Full understanding of an AC model is effectively required to assess that the model correctly implements the AC policy.
- Addition of new "objects" and "capabilities". As an application is developed, developers may need to define new "capabilities" in relation to some of the "objects" they created. Comprehension of the implemented AC model may help determine whether interactions with the new "objects" need to be described with new "capabilities" or if existing "capabilities" are sufficient. Introduction of unnecessary "capabilities" adds complexity to AC models and decreases long term maintainability.
- Impact analysis of AC model related modifications. Extracting the implemented AC model can help showing the impact of conferring a new "permission" to a "user" by highlighting newly accessible sections of code.

- Regression testing. For example, comparing extracted AC models along the evolution of a software may help target "objects" for which related AC were modified and that require re-testing.

- Software documentation for auditing and training. AC models can serve as a solid base for writing documentation, can facilitate communication among customers, designers, and developers and thus can simplify training.

In this paper, we are interested in extraction and reporting of AC models from source code using reverse-engineering techniques. Though we studied a Web based PHP application, the presented technique is not strictly restricted to PHP nor to Web applications and can be customized for other languages and paradigms.
In the context of this paper, our objective is to identify the *necessarily enabled* "capabilities" required to access "objects" in an application. The *necessary* perspective is important since *enabled* "capabilities" may not be *sufficient* to restrict access to "objects".
For the rest of this paper, we will refer to a *Necessarily Enabled Capability* under the abbreviation NEC. Moreover, according to the definition of "objects" presented above, we may now refer to "objects" as statements. The detailed definition of a NEC will be presented in Section 5.3.3.

5.1.1 Case study: AC model extraction in Moodle

For this paper, we studied the AC model of an open-source course management system, written in PHP, known as Moodle [118]. The first version was released in 2002 and the application is still under active development.
The recovery of the AC model in Moodle currently poses serious challenges. Indeed, several difficulties arise when trying to extract an AC model. Among them, we can list:

- Sparse documentation

- Inter-procedural flow issues

- Dealing with programming language idiosyncrasies

- Discrepancies between documentation (when available) and implementation

In this paper, we implemented a tool to automate the identification of NECs related to statements in an application. Specifically, we applied our method on a recent version of Moodle, a medium-scale software application that presents a rich AC model. As we will detail in section 5.3, our tool makes use of a PHP parser, a model extractor, and a model checker to generate reports and highlighted source code.

5.1.2 Objective and contributions

We present a novel approach, based on model checking, to extract the AC model from source code and to communicate the extracted model to developers. NECs are reported using source code coloring, that should help developers to explore and understand the implemented AC model model in a visual way.
The contribution of this paper is two-fold. First, our approach is able to extract and report rich and complex AC models from medium-scale software systems. Second, our technique runs in linear time and benefits from model checking formalism.

5.1.3 Outline

The rest of the paper is organized as follows: section 5.2 presents the AC model in Moodle, as described in its documentation. Section 5.3 presents our AC model extraction framework and describes each of its components. Section 5.4 first shows results and statistics about the extracted AC model from Moodle and then shows samples of code coloring reports. Section 5.5 discusses our results and threats to validity. Section 5.6 concludes the paper.

5.2 Access controls in Moodle

AC in Moodle is enforced through capabilities, as described in Moodle documentation [118]. Capabilities in Moodle correspond to the "capability" definition we provided in section 5.1.

5.2.1 Access control patterns in Moodle

As mentioned earlier, the goal of this paper is to identify NECs associated with each statement in Moodle. Therefore we had to detect AC patterns in Moodle source code. This section presents the patterns we identified and investigated.
The first pattern consists of a single Moodle function: *require_capability*. The pattern detects calls to *require_capability* with two parameters: a constant string representing a "capability" name and a variable representing the current context. *Require_capability* function controls access of a "user" to subsequent statements by interrupting the execution of the PHP script if access to the "object" is denied.
An occurrence of this pattern is illustrated in Listing 5.1, at line 4. In this case, execution is interrupted, if the "user" is denied the right to solicit *moodle/site:uploadusers* "capability" in the *system* context.

```
1 /// File header
2
3 admin_externalpage_setup('uploadusers');
4 require_capability('moodle/site:uploadusers', get_context_instance(CONTEXT_SYSTEM));
5
6 /// Do the job
```

Listing 5.1: Example of require_capability usage

The second pattern consists of a Moodle function: *has_capability*, which is called inside a conditional statement. The pattern detects calls to *has_capability* function with two parameters: a constant string representing a "capability" name and a variable representing the current context. *Has_capability* function returns a Boolean variable representing whether or not the "user" has sufficient "permissions" to access the "object" targeted by the "capability". "Access control" in this case is performed by the conditional statement.
Listing 5.2 illustrates an occurrence of this pattern at line 3. In this case, appropriate hyperlinks are added to the page that will be displayed, if the "user" is granted the right to solicit *moodle/course:viewparticipants* "capability" in the *course* context.

```
1 $navlinks = array();
2
3 if (has_capability('moodle/course:viewparticipants', get_context_instance(CONTEXT_COURSE, $course
      ->id)))
4 {
5     $navlinks[] = array('name' => $strparticipants,
6                         'link' => "$CFG->wwwroot/user/index.php?id=$course->id",
7                         'type' => 'misc');
8 }
9
10 ///Do the job
```

Listing 5.2: Example of has_capability usage

Moodle version 1.0 was released on August 2002. However, the AC model described in this paper was introduced in version 1.7, on November 2006. Prior to version 1.7, Moodle used fixed "roles". AC were then performed once, at the top of the file, using functions like *isteacher*, *isstudent*, or *isguestuser*. Before version 1.7 was released, a large portion of Moodle's code therefore had to be updated to make it compliant with the new AC model. The need for backward compatibility has sometimes led to implementation of mixed AC, as illustrated in Listing 5.3. Both *isloggedin* and *isguestuser*, on line 1, implement legacy AC while *has_capability* implements AC for the new AC model.

```
1 if (isguestuser() or !isloggedin() or has_capability('moodle/legacy:guest', $modcontext)) {
2     $canreply = ($forum->type != 'news'); // no reply in news forums
3 }
4 else {
5     $canreply = forum_user_can_post($forum, $discussion, $USER, $cm, $course, $modcontext);
6 }
```

Listing 5.3: Intermixing new style and legacy permission checks

Moreover, we observed there exists other "access controls" patterns in the code. For example, multiple calls to *has_capability* are sometimes joined in a complex Boolean predicate inside an *if* statement. In the context of this paper, we only have investigated the first two patterns we presented: calls to *require_capability* and calls to *has_capability* inside a conditional statement and have left remaining patterns for further research. Section 5.5 will discuss related consequences and issues.

5.3 Model extraction framework

This section aims to provide the reader with an overview of our methodology. The goal of the framework is to extract and report an application's AC model.

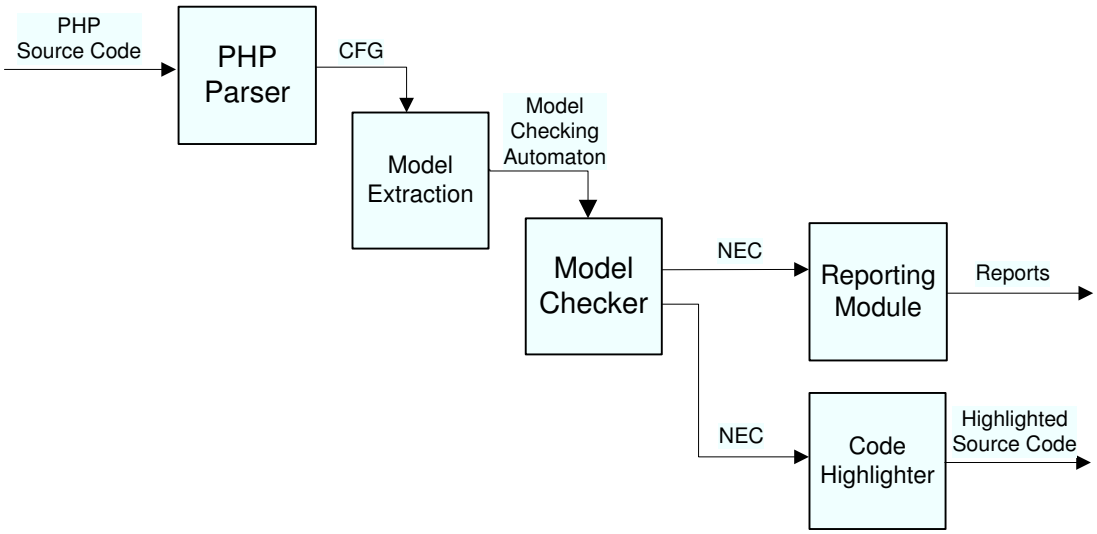


Figure 5.1: Process of extracting AC model from source code and reporting results.

The whole process of extracting an AC model a from source code is illustrated in Figure 5.1. The process starts with PHP source code and finishes with the production of reports and of highlighted source code.

5.3.1 PHP parser

Our experimental setup uses a PHP parser generated by JavaCC, a common parser generator tool for LL grammars. The PHP grammar used with JavaCC was inspired by [144] although it was heavily modified to fit our needs. Also, the grammar was augmented with JJTree instructions to generate Abstract Syntax Tree (AST) for each source file of Moodle.
The output of the parser is a program Control Flow Graph (CFG):

$$CFG = (V_{CFG},\ E_{CFG}) \tag{5.1}$$

with a unique *entry* node $v_{in} \in V_{CFG}$ and a unique *exit* node $v_{out} \in V_{CFG}$. Nodes in V_{CFG} can be of type *generic*, *call_begin*, *call_end*, *entry*, or *exit*. Nodes of type *generic* are involved in intra-procedural control flow; nodes of type *call_begin*, *call_end*, *entry*, and *exit* are used in inter-procedural control flow.
Edges in E_{CFG} can be of type *generic*, *allow_*c_i, *prevent_*c_i, *call*, or *return*. Edges of type *generic* represent intra-procedural transfers of control that do not affect capability enablement; edges of type *allow_*c_i represent intra-procedural transfers of control that enable a capability c_i ; likewise *prevent_*c_i are edges that disable a capability c_i; finally, edge of type *call* and *return* represent inter-procedural control flow links.
Finding AC patterns in Moodle involves finding some syntactic patterns in the AST. As we mentioned in subsection 5.2.1 the primitive *has_capability*(c_i, u) is a function determining if a "user" u can solicit a "capability" c_i.
Execution of code based on privileges can thus be accomplished by the simple use of a conditional statement using *has_capability* as its condition. If the predicate returns true, control is transferred to a block in which "capability" c_i is enabled for "user" u until block end. Conversely, if the predicate returns false, control is transferred to another block in which "capability" c_i is disabled for "user" u until block end. Those transfers of control are represented in the CFG with *allow_*c_i and *prevent_*c_i edges.
In a similar manner, calls to *require_capability*(c_i, u) are also represented with *allow_*c_i and *prevent_*c_i edges except that *prevent_*c_i edges always point to the *exit* node, since the program terminates.

5.3.2 Model extraction and inter-procedural aspects

The model extraction uses a CFG from the PHP parser and transforms it into an automaton $\mathcal{A}$ suitable for model checking:

$$\mathcal{A} \ = \ (Q_{\mathcal{A}},\ L_{\mathcal{A}},\ T_{\mathcal{A}},\ q_0,\ V_{\mathcal{A}},\ G_{\mathcal{A}},\ A_{\mathcal{A}}) \tag{5.2}$$

where $Q_{\mathcal{A}}$ is a finite set of states; $L_{\mathcal{A}}$ is a finite set of labels applied on the states; $T_{\mathcal{A}} \subseteq Q_{\mathcal{A}} \times Q_{\mathcal{A}}$ is a set of transitions; q_0 is the initial state; $V_{\mathcal{A}}$ is a set of variables used in "guards" and "assignments"; $G_{\mathcal{A}}$ is a set of "guards" that are logical propositions over $V_{\mathcal{A}}$ and are associated with transitions; and $A_{\mathcal{A}}$ is a set of assignments that modify the value of variables and are also associated with transitions.

The model extraction is performed by operations that include the rewriting of intra-procedural and inter-procedural nodes, and the identification of capability granting functions through AC patterns. The intra-procedural nodes V_{CFG} and edges E_{CFG} are directly rewritten in the automaton $\mathcal{A}$ into the corresponding states $Q_{\mathcal{A}}$ and transitions $T_{\mathcal{A}}$. A label $stmt_x$ is applied on each state to indicate to which statement in the source code this state corresponds. The label is formed as $stmt_x$ with x as a unique identifier.

In more of states and transitions, inter-procedural nodes and edges will also produce variables $V_{\mathcal{A}}$, guards $G_{\mathcal{A}}$, and assignments $A_{\mathcal{A}}$. The variables with guards and assignments are used to reproduce the logic of inter-procedural analysis as explained in [100].

Inter-procedural representations in the automaton add time and complexity; nevertheless, we found they are essential. Intra-procedural analysis only deals with events that occur inside the scope of a function, but a NEC affects all statements executed after it even if they are located in other functions.

Thus, some functions do not produce NEC but are still affected by NECs produced by their calling function. Therefore, intra-procedural analysis alone is not precise enough, because it limits NECs' scope to statements inside a function.

Although we could have, in principle, done the same analysis with static analyzes by using an algorithm that operates directly on the CFG, as demonstrated by [145], we preferred to use model checking because of the formal reasoning it offers. Optimized and specialized inter-procedural static analyzes are hard to devise and it may be difficult to assess their soundness and complexity. Making an inter-procedural automaton for model checking is at least as difficult as doing an inter-procedural static analysis, but we found reasoning about the formally specified automaton easier than reasoning about an algorithm that operates on a CFG.

5.3.3 Model checking

Software model checking [33] is the algorithmic analysis of programs to prove properties of their executions. While originating from logic and theorem proving fields, it has now evolved as a hybrid technique, simultaneously making use of analyzes traditionally classified as theorem proving, model checking, or dataflow analysis [87].

A well-known limitation of model checking techniques is known as the combinatorial "state

explosion problem". Various techniques have been developed over the years to circumvent this problem and analyze increasingly larger software. Among them, we find bounded, symbolic and abstract model checking as well as a large variety of state-space exploration and graph refinement algorithms.

The capability c_i is defined as a NEC for the statement $stmt_j$ if the following temporal logic formula is verified:

$$\Diamond stmt_j \wedge \Box(\ stmt_j \wedge c_i) \tag{5.3}$$

meaning that the automaton can possibly reach a statement labeled $stmt_j$ ($\Diamond stmt_j$) and always when $stmt_j$ is reachable, "capability" c_i is enabled ($\Box(\ stmt_j \wedge c_i)$). States satisfying Equation 5.3 correspond to statements that are reachable by an execution for which a capability c_i is enabled, but that cannot be reached by an execution for which c_i is disabled. Indeed, capability c_i represents a NEC for statement $stmt_j$.

The model checker in Figure 5.1 solves the reachability problem of states in the automaton and therefore solves the corresponding problem of identifying NECs for statements.

For ease of implementation, we produce many automata, each one representing the whole system for one of Moodle's 217 capabilities. Building an automaton representing all capabilities would require model checking the effect of all 2^{217} combinations of capabilities; this would lead to a combinatorial state explosion. We therefore analyzed the effect of each 217 capability by producing one automaton for each capability.

Introduced in [66, 150], may and must models are among the latest approximation techniques for model checking. For those more familiar with program analysis techniques, may and must models are conceptually related to static and dynamic analysis respectively. May models provide information about a whole system, covering *all* of its possible executions. For performance and scalability reasons may models often provide an over-approximation of reality. Thus, an erroneous execution reported by a may model may not be triggered by any real execution path. Conversely, must models, as their name suggest, report information that are guaranteed to hold on those particular executions that were reported. Must models typically under-approximate the whole system as covering all possible executions of a program is often infeasible.

In this paper, we present a scalable summary-based approach to perform formal verification of multi-feature security properties on large software. To achieve scalability, we combined a may model with an inter-procedural analysis [18, 139] on a restricted number of contexts. While this restricts the kind of property that can be verified, it can perform analysis on arbitrarily large systems efficiently without any loss of precision due to the specially conceived inter-procedural model.

5.3.4 Property satisfaction profiles

Property satisfaction profiles (PSP) were introduced in [99] as Boolean vectors representing Boolean satisfaction values of a set of properties defined on CFG nodes. In the context of this study, we defined PSP for each node in the security CFG as: $psp(v) = (p_0,\ p_1,\ ...,\ p_i,\ ...,\ p_n)$, where each property represents the Boolean satisfaction of a specific capability in Moodle. More specifically, every property is a Boolean pair, indicating if the state is reachable without and with the capability, see Equation 5.4. This is an important precision since strong conclusions may be drawn when a node is reachable *only* in one of the two states. A node that is only reachable with a specific capability is guaranteed, under the limitations of the model, to be secured with this capability under any executions of the system.

$$\begin{aligned} psp(v) \ &= (p_{v_0},\ p_{v_1}) \ = \\ &((reachable(q_{v,0,0}) \vee reachable(q_{v,1,0})), \\ &(reachable(q_{v,0,1}) \vee reachable(q_{v,1,1}))) \end{aligned} \tag{5.4}$$

PSP are produced through the calculation and conjunction of state reachability properties [90], on each security models. $Psp(v)$ vector therefore represents the Boolean values of reachability for potentially every available capabilities in Moodle on a single CFG node.
Mapping model states to CFG nodes is trivial as graph rewriting rules create four model states for every nodes in the CFG [99, 100]. Mapping back CFG nodes to specific lines of code in PHP files is also a trivial operation as the parser can instruments the CFG with pertinent information. Using these information, we are able to highlight every lines of code in original PHP files according to their corresponding $PSPs$.

5.4 Experiments and results

Moodle is a medium-scale application, totaling 625 473 LOC across 2331 PHP files. The Moodle website reports a total of 39 412 496 users across 211 countries [119]. In the context of this paper, we analyzed Moodle 1.9.5, a recent version although not the latest.
The goal of our analysis is to extract NECs for each statement and communicate results to developers in a comprehensive manner.
Table 5.1 shows execution times for the different steps involved in our technique.
Results presented in this paper are derived from model checking 217 automata, each representing a single "capability" in Moodle. Hence, we report the average times to extract the model checking automata and perform model checking.
Moodle 1.9.5 has a total of 217 "capabilities". Out of these, we were surprised to observe that only 146 "capabilities" were actually identified as NECs. We will discuss, in section 5.5

Table 5.1: Execution times for AC model analysis of Moodle

Step	Time
Parsing Moodle and building CFG	13m 55s
Model extraction	(average) 20s
Computing NECs	(average) 8s
Total time on 217 capabilities	114m 44s

consequences and issues of this observation.
Furthermore, out of a total of 2^{217} possible combinations of NECs, results revealed that only 166 combinations of NECs were actually detected by our approach. Out of these, 135 consist of a single NEC, 28 of two NECs and 3 of three NECs.
In the context of this paper, we define NEC coverage as $NEC_i \rightarrow \{stmt_1, \ldots, stmt_n\}$; the set of statements a NEC is associated to. Interesting results may be drawn from calculating NEC combinations coverages. Results are reported in Figure 5.2.
We can observe that cardinality of NEC combinations coverages vary widely from one combination to another. Out of the 166 observed NEC combinations, cardinality of coverage varied from 1 to 5347 with an average cardinality of 373.84 and a standard deviation of 725.15.
Crossing coverage information with Moodle directory structure provides some interesting insights about Moodle AC model. In Table 5.2 we present base directories of Moodle with their number of files and average percentage of covered statements. This shows, for example, that files in the *admin* directory are less covered, on average, compared to directories with a similar number of files like *grade* and *course*.
Coloring statements according to their NEC combinations is an effective way to communicate the implemented AC model to developers. Figure 5.3 shows an example of a colored section of PHP code. A call to *require_capability* at the head of the file colors the whole code in light gray. Sections of code that are also covered by *moodle/site:doanything* through a call to *has_capability* are colored in dark gray.

5.5 Discussion

Since our algorithm has linear time and memory complexity [100], it scales well to analysis of medium/large applications. Furthermore, since we independantly analyzed "capabilities" of Moodle, linearity is preserved. In practice, Table 5.1 shows we are able to generate model checking automata and extract NECs from Moodle, in about 30 seconds on average.
Results revealed that out of 217 documented capabilities, 71 are never identified as NECs by our technique. Different reasons may explain this matter of fact:

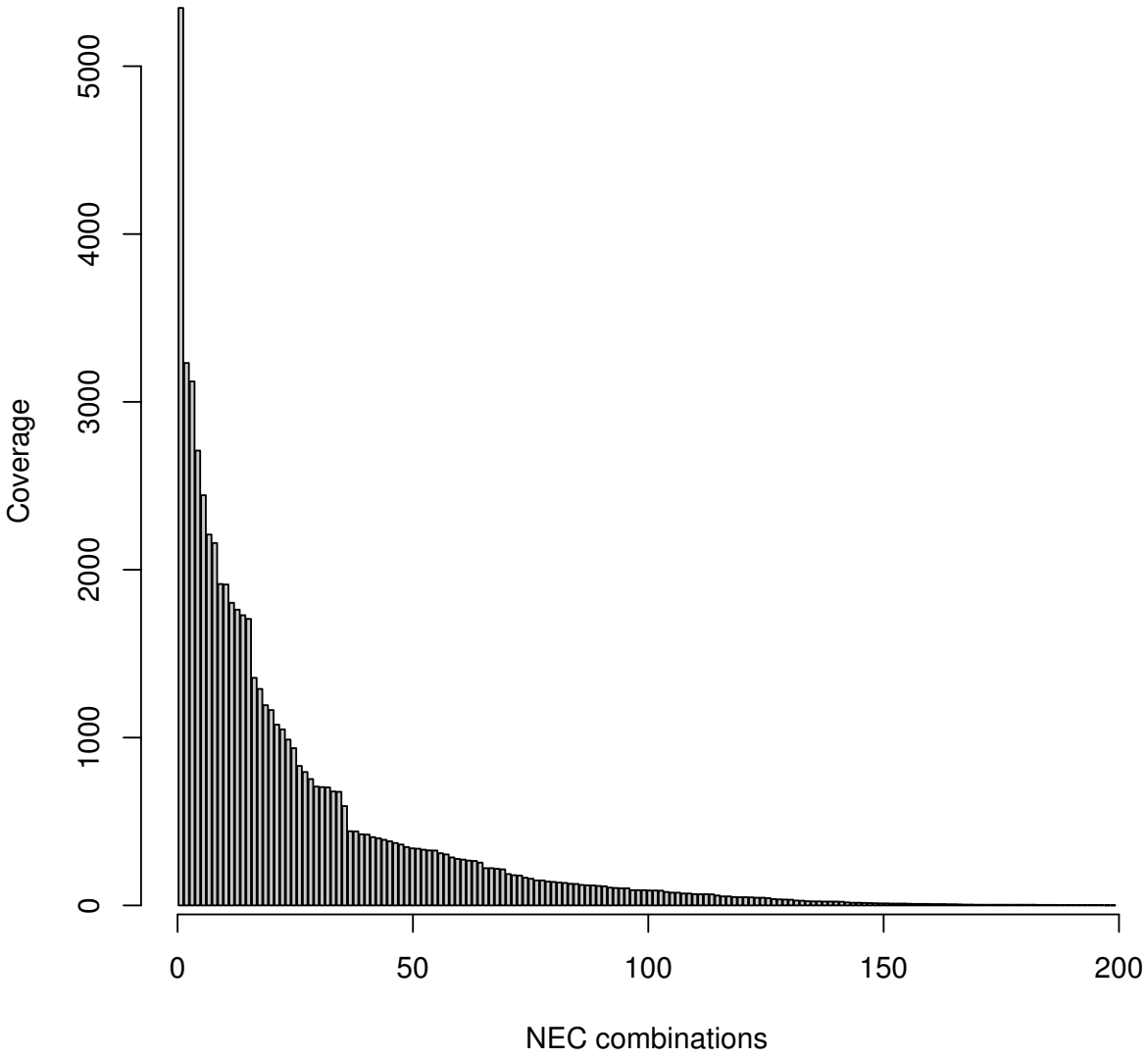


Figure 5.2: Histogram reporting NEC combinations coverages

1. AC patterns involving *require_capability* and *has_capability* functions have been investigated to obtain the results presented in this paper. Additional AC patterns in Moodle are described in subsection 5.2.1 and can be investigated using the same approach.

2. As presented in subsection 5.3.3, our approach identifies NECs through model checking of the extracted automata. AC patterns located in dead code are undetected since they are unreachable.

Table 5.2: Moodle directories with their number of files and average NEC coverage

Moodle directory	Number of files	Average percentage of covered statements (%)
/	7	0.15
/admin	170	8.10
/auth	40	0.0
/backup	13	6.03
/blocks	98	0.34
/blog	10	4.22
/calendar	9	0.12
/course	77	15.52
/enrol	32	4.67
/error	1	0.0
/files	1	41.62
/filter	17	2.70
/grade	103	14.47
/group	14	26.48
/install	86	0.0
/iplookup	1	0.0
/lang	118	0.0
/lib	711	0.31
/login	10	0.93
/message	8	7.98
/mnet	10	7.59
/mod	453	5.73
/my	2	0.0
/notes	7	5.63
/pix	2	0.0
/question	107	1.21
/rss	1	0.0
/search	100	0.28
/sso	3	0.0
/tag	10	12.80
/theme	47	0.99
/user	41	7.32
/userpix	2	46.62

We also observed that out of 2^{217} possible combinations of NECs, 166 were actually detected; the longest one being of length 3. This observation supports the assumption that while AC models can be very rich and complex, implemented models may prove to be simpler. It also suggests that developers tend to maintain a relatively low level of complexity when it comes

```
306:         // delete user
307:         if (!empty($user->deleted)) {
308:             if (!$allowdeletes) {
309:                 $usersskipped++;
310:                 $upt->track('status', $strusernotdeletedoff, 'warning');
311:                 continue;
312:             }
313:             if ($existinguser) {
314:                 if (has_capability('moodle/site:doanything', $systemcontext, $existinguser->id)) {
315:                     $upt->track('status', $strusernotdeletedadmin, 'error');
316:                     $deleteerrors++;
317:                     continue;
318:                 }
319:                 if (delete_user($existinguser)) {
320:                     $upt->track('status', $struserdeleted);
321:                     $deletes++;
322:                 } else {
323:                     $upt->track('status', $strusernotdeletederror, 'error');
324:                     $deleteerrors++;
325:                 }
326:             } else {
327:                 $upt->track('status', $strusernotdeletedmissing, 'error');
328:                 $deleteerrors++;
329:             }
330:             continue;
331:         }
```

Figure 5.3: Example HTML output of a PHP file colored according to combinations of NECs

to protecting their code.

The histogram of Figure 5.2 revealed major differences between NEC coverages, which, we think, may reflect different coding intentions.

Calls to *require_capability* usually appear at the top of file since, in general, developers want the execution of the script to be interrupted quickly if a "user" does not have appropriate "permissions". Therefore, NECs induced by calls to *require_capability* functions tend to have higher coverage. We think that such NECs are used for "protection" purposes. For example, in Moodle, the script governing bulk registration of users from a comma separated file is covered at 97% by the *moodle/site:uploadusers* NEC by a call to *require_capability* at the head of the file.

Alternatively, *has_capability* functions tend to have a more limited scope and we think that NECs induced by this function may represent options. For example, Listing 5.2 shows that *moodle/course:viewparticipants* NEC covers the single statement that governs the addition of hyperlinks to a Web page.

While not a hard rule, manual inspection tends to support the hypothesis that higher coverage relates to protection and lower coverage to option control. Further investigation would be required to fully assess this hypothesis.

NEC coverage reports are provided to ease reasoning about capability enforcement by developers. For instance, it may help to:

1. Identify vulnerabilities. Given a coverage report, a developer may identify and investigate pieces of code that are not covered with the expected NEC combination.

2. Understand the AC model, as implemented, when documentation is sparse or unavailable.

3. Document existing code. Since our analysis is precise and conservative [115, 100, 99], coverage results can be used for documenting the underlying AC model.

5.5.1 Threats to validity

Since we applied our technique to a single system, conclusions from our analysis may not generalize to all software systems or AC models. Furthermore, to provide a scalable, linear time analysis, we had to introduce the following approximations:
First, we only considered *require_capability* and *has_capability* functions. Legacy functions, such as the ones described in subsection 5.2.1, were ignored since they don't belong to the new AC model and are tagged for deletion. Moreover, boolean predicates including one or more calls to *has_capability* function were not treated in this preliminary version. Enhancement of our technique with treatment of boolean predicates is planned for a future release.
Furthermore, although our technique is largely application independent, AC patterns remain application dependent. However, we may argue that access control patterns are usually quite stereotyped, since they often test the value of an AC property. Therefore, new access control patterns can often be implemented as simple substructure matches in an AST.
On the other hand, implementation of complex patterns that are expressed as complex Boolean predicates would require more advanced analyses and more programming effort for tree matches. Nevertheless, we think that complex security patterns may be hard to maintain and should be replaced by a call to a specialized security function that is maintained by a security team, rather than by general developers.
Moreover, PHP supports string evaluation through its *eval* function. It takes an arbitrary string and evaluates it as if it was native PHP code. Our technique cannot correctly infer its behavior and two control flow approximations were considered for this paper:

1. Make every *eval* function a potential call to every function in Moodle.

2. Ignore *eval* functions.

Manual inspection of the 235 calls to *eval* in Moodle revealed that none of them induces inter-procedural calls. *Eval* calls were therefore treated as *generic* edges in the CFG. In the context of this study, this approximation has no consequence since calls to *eval* can rightly be represented with a simple edge in the CFG.

Finally, during model extraction, our conservative inter-procedural analysis may also keep infeasible execution paths in the model checking automaton. For instance, consider a conditional statement with a predicate that can never be true. Our technique currently cannot detect such cases and would therefore report NECs on unexecutable statement. In practice, our technique remains conservative since it does not report false information; if the statement was to be executed, it would be covered by the NEC.

5.5.2 Related Work

Taint analysis

In some respects, our approach, when applied to a single "capability", relates to taint analysis. Generally speaking, one refers to *tainted* variables do designate untrusted data that can flow to security-sensitive code. Untrusted data are therefore tagged as *tainted* until some sanitizing routine sanitizes them. Our approach implements a form of control flow taint analysis where AC are analogous to sanitizing routines that sanitize *tainted* executions.
However, in contrast with classic taint analyzes, in which no distinction is made between sanitizing routines, the presented approach is able to tag executions with precise sanitation information, in the form of multiple and independant NECs.
Tripp et al. [163] implemented a static taint analysis for Java (TAJ). Their approach aims at identifying security vulnerabilities in Web applications and is able to handle reflective calls, flow through containers and nested taint. Interestingly, the authors share our willingness to provide a scalable approach, suited for industrial size applications. Their approach mainly differs from ours from the language (Java vs. PHP) point of view.
Clause et al. [34] presented a framework (DYTAN) for conservative dynamic tainting. Their framework takes in entry a user supplied configuration file, describing the taint analysis to be performed, and instruments x86 binaries accordingly. Execution of the instrumented binaries generates reports according to user's specifications. Interestingly, to our knowledge, they are among the first authors to explicitly address the problem of control flow tainting. While their approach could theoretically be used to implement an analysis similar to ours, their framework can currently treat x86 binaries only and is therefore unsuitable for source code of applications written in PHP.
Jovanovic et al. [88] developed a static taint analysis for identification of cross-site scripting (XSS) vulnerabilities in PHP Web applications. They used an inter-procedural and context-sensitive data flow analysis enhanced with literal analysis to detect potential XSS vulnerabilities in PHP scripts. They claim a false positive rate of 50%. Their work differs from ours in the implemented analysis, which is solely static and aimed at XSS flaws detection. Our

analysis uses model checking on top of an inter-procedural analysis to provide precise results.

Access control model extraction

In [4] Alalfi et al. proposed a framework to reverse-engineer and verify secureUML diagrams for PHP Web applications. They suggest their framework could be used to detect access privilege violations in Web applications. Our technique differs from theirs in the chosen approach; they model an application AC model with a secureUML diagram that can further be converted to a model checking automaton while we extract AC model using static analysis techniques and convert the resulting CFG in a model checking automaton.
Koved et al. [96] proposed an approach to compute access rights requirements in Java applications. The authors used their tool to construct an Access Right Invocation Graph (ARIG), representing authorization model as implemented in the code. While conceptually related to our analysis, our approach was designed for analysis of PHP Web applications instead of Java applications. Furthermore, since Java is a typed language, it generally eases the challenges faced by static analysis.
Pistoia et al. [136] extended the preceding approach with taint analysis for detecting and automatically protecting code that should have privileged access. Their technique also identifies unnecessary and redundant privileged code and tag tainted variables as benign or malicious depending on whether or not it is used in privileged functions.
Wang et al. [167] presented a technique for automatic reverse engineering of an application's access-control configurations. When available, access rights in configuration files may prove a fast and efficient way to retrieve security specifications. However, several software applications, such as Moodle, lack such information, thus requiring a different approach.
A survey in [8] presents 24 different modeling methods used with websites. It is interesting to observe that, among the 24 reported modeling methods, none is specifically targeted at AC modeling, underlying the relevance of our approach.

5.5.3 Future work

In this paper, we have presented a subset of all the research questions that our technique may help solving.
Considering the reasonable execution times we obtained on Moodle, it would be interesting to extend our analysis to monitor evolution of AC models along multiple versions of a software application. Comparison of AC models from one version to another would help testers to focus on modified blocks of code and refine regression testing strategies.
If scalability became an issue on larger systems, we could consider implementation of incre-

mental, or bounded model checking techniques [87].
In the long term, we would like to enhance our technique with dynamic analysis. Dynamic analysis could help eliminate infeasible execution paths that were introduced by static approximations, improving overall precision of our technique. An interesting approach that combines model checking with static and dynamic analyzes to test Web applications is presented in [14].

5.6 Conclusion

In this paper, we have presented a novel approach that extracts the implemented AC model from source code based on NECs. We evaluated our technique on Moodle, a medium-scale, course management system written in PHP.
Using a PHP parser, Moodle's *AST* was computed and AC patterns were identified and extracted. An inter-procedural CFG with capability enabling edges was further derived from Moodle's *AST*. Model checking automata, each representing one capability of Moodle, were then extracted from the CFG. Model checking results were interpreted as NECs on each statement. NEC data were finally mapped back to source code using a coloring strategy.
Results shown that our technique is able to extract NECs on an a medium scale application with acceptable practical performances, supporting the assumption that it can be used for daily development and evolution monitoring.
Moreover, results revealed how rich and complex AC models may have comparatively simple implementations; out of every possible NEC combinations in Moodle, a very small proportion is actually used.
Furthermore, NECs were reported using simple code coloring, providing developers with a practical tool to understand and analyze implemented AC models. Reported NEC coverage also revealed the potential existence of two distinct AC usage: protection and option.
Globally, we think our approach may help in comprehension, documentation and refactoring of medium to large-scale Web applications.

Acknowledgements

This research has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program.

CHAPTER 6

CORRECTIONS FOR PAPER 1

The previous chapter contains the published version of the paper entitled: "Extraction and Comprehension of Moodle's Access Control Model: A Case Study". The current chapter addresses comments and corrections by the jury regarding this paper.
In subsection 5.3.3, we mentioned that we analyzed the effect of each of Moodle's capabilities independently. Consequently, the presented approach cannot directly take into account interactions between different capabilities. However, a simple post-processing step is sufficient to merge back the results and draw conclusions about interactions between capabilities. Suppose, for example, the following capability check:

```
1 if (has_capability('edit_post') && has_capability('create_post'))
2     echo 'Success';
3 else
4     echo 'Fail';
```

Listing 6.1: Capturing interactions with independent analysis of capabilities

Independent analysis of the `edit_post` and `create_post` capabilities would reveal that line 2 is definitely protected by the `edit_post` capability and also definitely protected by the `create_post` capability. Merging these results would lead to the conclusion that line 2 is definitely protected by both capabilities. This approach is however limited in certain contexts. Consider the following example:

```
1 if (has_capability('edit_post') || has_capability('create_post'))
2     echo 'Success';
3 else
4     echo 'Fail';
```

Listing 6.2: Limitations of independent analysis of capabilities

In this case, independent analysis of each capability would reveal that line 2 is possibly protected by the `edit_post` capability and also possibly protected by the `create_post` capability. However, we would not have enough information to conclude that line 2 is always protected by *at least* one of the two capabilities. If such level of precision is required, however, the model checking automaton can be augmented with more states that should be labelled in

such a way to represent interactions between capabilities. One must be aware, however, that the number of states will increase in an exponential manner with the number of interactions that will be modelled.

In subsection 5.5.1, we mentioned that after manual inspection, calls to *eval* were ignored. A member of the jury rightly observed that *eval* can contain access checks and impact the results of SPT. While it was not clearly stated in the original paper, we manually ensured that this what not the case in the investigated version of Moodle.

On another line of thoughts, in section 5.6, we reported that: "out of every possible NEC combinations in Moodle, a very small proportion is actually used". What we meant was that given a set of capabilities S of cardinality n, any given combination of capabilities in $\mathcal{P}(S)$ can be enforced. In other words, in theory, a statement can be protected by 2^n different combinations of capabilities. In practice, however, we observed that the number of combinations of capabilties that are enforced is much smaller.

CHAPTER 7

PAPER 2: ALIAS-AWARE PROPAGATION OF SIMPLE PATTERN-BASED PROPERTIES IN PHP APPLICATIONS

ABSTRACT

In this paper, we present novel algorithms for the propagation of *pattern-based* properties in PHP applications. Intuitively, pattern-based properties designate those properties that are intrinsically associated to syntactic patterns in the source code. Security checks in access control models are an example of pattern-based properties. At the source code level, permissions are typically verified with stereotyped constructs, called security checks, that can be detected with syntactic patterns.

Depending on the program, pattern-based properties can be aliased to variables that are propagated through the application. In that context, support from data-flow approaches is needed to track the propagation of patterns through the application. In the context of this paper, we focus on the alias-aware propagation of security checks through PHP applications. Specifically, we investigated the propagation of security checks in 8 PHP applications that implement access control models.

We show how, using the Datalog language, one can implement conceptually complex data-flow algorithms in an incremental, intuitive and compact manner. From the results perspective, we show how our algorithm identifies security checks and security check aliased variables in a precise way. The reported false positive rate varies between 0% and 4% for the investigated applications.

7.1 Introduction

In recent years, a significant body of work have been devoted to the design of static analyses for PHP applications, mainly in the field of security. On one hand, we find data-flow based approaches [125, 173, 83, 92] for the detection of *taint-style* vulnerabilities. On the other hand, we find string analysis based approaches [116, 168, 169] that approximate the output of a PHP program in order to determine if it correctly filters out malicious input data.

In this paper, we propose a novel static analysis algorithm for the propagation of pattern-based properties in PHP applications. Informally, pattern-based properties designate those

properties that are intrinsically associated to syntactic patterns in the source code. For example, permissions in access control models usually are pattern-based properties. At the source code level, permissions are typically verified with stereotyped constructs, called security checks, that can be detected with syntactic patterns.
In a previous study, we investigated the extraction of access control models in PHP applications [60]. Using syntactic patterns and model checking techniques, we extracted, for each statement, some of the permissions a user must own to execute the statement. In that study we did not, however, investigate security check aliased variables. Suppose for example that the result of a security check on permission p is stored in a variable v and that v is further checked in a conditional statement. Without data-flow support, we cannot deduce that verifying the value of v is conceptually equivalent to performing a security check on p.
The main goal of this paper is to design an inter-procedural, alias-aware pattern propagation algorithm, tailored for PHP applications. We show how this algorithm increases the precision of security checks detection and how, with minor adjustments, the same algorithm can be used to approximate the target of dynamic include statements. For the sake of simplicity, we use the term "pattern propagation" in place of pattern-based properties propagation. All the presented algorithms are complete, reusable Datalog programs.
Section 7.2 introduces the Datalog language. In sections 7.3 to 7.6, we guide the reader through algorithms of incremental complexity, starting with a simple, intra-procedural version, and ending up with an inter-procedural, alias-aware pattern propagation algorithm. Section 7.7 presents some uses for pattern propagation in PHP proprams. Sections 7.8 and 7.9 show and discuss the results. Section 7.10 presents related work and section 7.11 concludes the paper.
The main contributions of this paper are as follows:

- Several reusable pattern propagation algorithms of various precision, implemented as Datalog programs, readily executable in the `bddbddb` framework [172].
- An integrated alias analysis, tailored for the semantic of the PHP language.
- Evaluation of the proposed algorithms for the extraction of security checks in PHP applications and approximation of the target of dynamic include statements.

7.2 The Datalog language

All the algorithms presented in this paper are executable Datalog [164] programs. In this section, we provide a brief introduction to the Datalog language.

Datalog is a deductive database language that is conceptually related to Prolog. Inputs to a Datalog program are relations, similar to relational database tables. Datalog relations are represented with two-dimensional tables where columns represent attributes and rows represent tuples of attributes. Tuples in a relation are interpreted as predicates. For example, if the tuple (a, b, c) is in the relation A, we consider the predicate $A(a, b, c)$ to be true. If the tuple is not in the relation, the predicate is false. For example, suppose the *Parent* relation, representing the parenthood relationship between a parent and a child. The *Parent* relation can be expressed with the $Parent(parent, child)$ Datalog relation. Each tuple in the *Parent* relation represents the fact that *parent* is the parent of *child*. Thus, if Sarah is the mother of Jack, the tuple $(Sarah, Jack)$ is in the *Parent* relation and we say that the predicate $Parent(Sarah, Jack)$ is true.

A Datalog program consists of a set of rules over predicates. Rules define new predicates based on the conjunction of other predicates. Moreover, Datalog allows for the recursive definition of predicates. For example, suppose that we want to compute the $Ancestor(ancestor, child)$ relation, representing the fact that *ancestor* is an ancestor of *child*. The *Ancestor* relation can be defined as the transitive closure over the *Parent* relation:

```
Ancestor(X,Y) :- Parent(X,Y).
Ancestor(X,Z) :- Ancestor(X,Y),
                 Ancestor(Y,Z).
```

In the Datalog program above, the first rule initializes the *Ancestor* relation and states that every parent X is also an ancestor of his child Y. The second rule computes the transitive closure and states that if X is the ancestor of Y and Y is the ancestor of Z, then X is the ancestor of Z. Note that the second rule recursively defines the *Ancestor* relation over itself. Datalog engines automatically resolve recursive rules until a fixed-point is reached.

Negation in Datalog programs can lead to infinite loops and must be handled with care. Consider the following rule:

```
P(x) :- E(x), !P(x).
```

Suppose that $E(1)$ is true and $P(1)$ is false. Then, $P(1)$ would be true after the first iteration, false after the second, etc. Thus, Datalog engines that allow negation only accept *stratified* programs [30]. Informally, a stratified program allows for the gouping of rules into *strata* that can be solved in sequence. Informally, a requirement for program stratification is that any negated relation must not depend on the head of the predicate. Besides stratified negation, the `bddbddb` framework also allows the use of constants (literals enclosed in quotes) and don't cares (an underscore) in place of variables.

7.3 Intra-procedural pattern propagation algorithms

This section presents intra-procedural algorithms for pattern propagation in PHP applications. It introduces the reader to program analysis with Dalatog programs and shows a flow-insensitive and a flow-sensitive version of the intra-procedural pattern propagation algorithm.

7.3.1 Intra-procedural, flow-insensitive pattern propagation

The first algorithm we present is intra-procedural and flow-insensitive. It only considers assignments of patterns to variables (e.g. `$a = check('permission')`) and variable copy (e.g. `$b = $a`). The inputs to this algorithm can be computed directly from the Abstract Syntax Tree (AST) of a program.

DOMAINS

V	15734	`variable.map`
Fn	9107	`functions.map`
L	217	`patterns.map`
F	3009	`files.map`

RELATIONS

input	$PatAssign$	(v:V, fn:Fn, fi:F, l:L)
input	$VarAssign$	($v1$:V, fn:Fn, fi:F, $v2$:V)
output	Pat	(v:V, fn:Fn, fi:F, l:L)

RULES

$$Pat(v, fn, fi, l) \ :- \ PatAssign(v, fn, fi, l). \tag{7.1}$$

$$Pat(v, fn, fi, l) \ :- \ VarAssign(v, fn, fi, w), \ Pat(w, fn, fi, l). \tag{7.2}$$

Algorithm 7.1: Intra-procedural, flow-insensitive, pattern propagation analysis

The structure of a Datalog program can be separated in three sections. The first section defines the domain of the attributes in the program. For example, the first line of the domain section defines the domain V to be of size 15734. The `variable.map` file maps the integer in the range [0,15734[to a string pattern representing the variable name. The domains used in Algorithm 7.1 are:

- V is the domain of variables. It represents all variable names in a PHP program.

- Fn is the domain of functions. The global scope of every PHP script is represented with the special "main" function. In PHP, all the instructions outside a class or a function belong to the global scope.
- L is the domain of patterns. In the context of this paper, we investigated patterns that represent security checks and summarized them as the permission they verify.
- F is the domain of files. All the files in a PHP application are listed in `files.map`.

The second section defines the relations of the Datalog program. Relation declarations are preceded by an optional keyword, defining whether it is an input or an output relation. If no keyword is specified, the relation is internal; it is used for rule computation but not written out. The second part of the relation declaration consists of the relation name and the last part defines the tuples' attributes, together with their domain. The relations used in Algorithm 7.1 are:

- $PatAssign$: V $\times$ Fn $\times$ F $\times$ L represents pattern assignment statements. For example, if the predicate $PatAssign(v, fn, fi, l)$ is true, it means that the variable v in function fn, in file fi is assigned the pattern l.
- $VarAssign$: V $\times$ Fn $\times$ F $\times$ V represents variable copy statements. If $VarAssign(v, fn, fi, w)$ is true, it means that the variable w is copied into the variable v in function fn, in file fi.
- Pat: V $\times$ Fn $\times$ F $\times$ L is the output relation, representing the association between a variable and a pattern. If $Pat(v, fn, fi, l)$ is true, it means the variable v in function fn, in file f must hold the pattern l.

The last section defines the rules of the Datalog program. The head of a rule, the part on the left-and side of the `:-` operator, consists of a predicate that will be true if the predicate on the right-hand side is true. As mentioned earlier, rules can be recursively defined, as is the case for Rule 7.2 in the program above. Rule 7.1 initializes the Pat relation. Rule 7.2 states that if the variable w is copied to the variable v in function fn, in file fi and w must hold the pattern l, then v must also hold l.

7.3.2 Intra-procedural, flow-sensitive pattern propagation

It is well known that flow sensitivity increases the precision of data-flow analyses. Flow-sensitive pattern propagation is very similar in nature to the classic constant propagation algorithm. Instead of constants, it reports the patterns a variable may be aliased to at each

point of a program. Adding flow sensitivity however requires more work from the front-end part of the analysis. While the inputs to the previous algorithm could be computed at the AST level, a flow sensitive analysis requires information about the sequence of instructions, which is usually extracted from a control-flow graph.

DOMAINS
Domains from Algorithm 7.1 plus:

S 404399 `statements.map`

RELATIONS

input	$PatAssign$	(v:V, s:S, l:L)
input	$VarAssign$	($v1$:V, s:S, $v2$:V)
input	$Follows$	(i:S, j:S)
input	$Unset$	(v:V, i:S)
output	Pat	(v:V, i:S, l:L)

RULES

$$Pat(v,i,l) \ :- \ PatAssign(v,i,l). \tag{7.3}$$

$$Pat(v,i,l) \ :- \ Pat(w,i,l), \\ VarAssign(v,i,w). \tag{7.4}$$

$$Pat(v,j,l) \ :- \ Pat(v,i,l), \\ !PatAssign(v,j,\_), \\ !VarAssign(v,j,\_), \\ !Unset(v,j), \\ Follows(i,j). \tag{7.5}$$

Algorithm 7.2: Intra-procedural, flow-sensitive, pattern propagation analysis

Adding flow sensitivity to the previous algorithm is simple, given that a pre-computed control-flow graph is available. Patterns are now propagated through a sequence of instructions until the variable gets reassigned or destroyed. The domain S replaces the domain Fn of Algorithm 7.1:

- S is the domain of statements. It replaces Fn in Algorithm 7.1. Pattern information is now computed at the granularity of statements instead of functions.

The second attribute Fn of $PatAssign$, $VarAssign$ and Pat relations is replaced by S to reflect the fact that pattern information is now available at every statement instead of every function. New relations were also introduced:

- *Follows*: S × S represents the sequence of statements in the program. If $Follows(i, j)$ is true, it means that the statement j immediately follows the statement i in the control-flow graph.
- *Unset*: V × S represents the PHP *unset* operator. This operator destroys a variable.

Rules 7.3 and 7.4 were drawn from Algorithm 7.1 and modified to support flow-sensitive analysis. Rule 7.5 states that if a variable v is associated with a pattern l at statement i, it will also be associated with l at statement j if it is not reassigned or destroyed at j and if j follows i. The negation operator is represented with an ! and _ represents don't care values.

7.4 Supporting inter-procedural analysis

Inter-procedural analysis takes function calls into account. By modeling passing of parameters and return of values, inter-procedural analysis increases the precision and the complexity of pattern propagation analysis. When dealing with inter-procedural analysis, an important distinction must be made between context sensitive and insensitive analyses. A context sensitive analysis can distinguish different call sites to a function, while a context-insensitive analysis merges information from all call sites.

While the algorithms presented in this paper are fundamentally context-insensitive, it has been shown that context-sensitivity can be achieved with context-insensitive algorithms if the call graph is *cloned* [172]. In a cloned call graph, functions and methods are duplicated in such a way that every call site calls a different clone. In this context, applying context-insensitive algorithms to the cloned call graph yields context-sensitive results. While cloning leads to a combinatorial explosion of the number of nodes in the call graph, it has been shown that cloned call graphs can be represented in a very compact manner in the `bddbddb` framework using binary decisions diagrams (BDDs). Discussion about achieving context-sensitivity using BDDs is beyond the scope of this paper. The interested reader can refer to [172, 24, 102]. An alternative, B-tree based Datalog engine for program analysis was also developed by Bravenboer et al. [29].

Flow sensitivity adds precision and complexity to the analysis. Algorithm 7.2 was presented to give the reader an idea of how flow sensitivity can be easily supported in Datalog programs. Flow sensitivity however requires one to keep information about every variable at every program point and increases the size of the search space significantly.

As mentioned earlier, BDDs can technically represent very large amounts data in a very compact manner. However, the data has to be organized in such a way that it takes advantage of the structure of BDDs. Unfortunately, finding the optimal organization of data for BDDs is an NP-complete problem [28]. Preliminary experiments with inter-procedural, flow-sensitive

analyzes did not yield conclusive results in terms of memory usage. Hence, we only present the flow-insensitive versions of the inter-procedural algorithms we developed. Results however show that flow-insensitivity do not hinder the quality of the results in a significant manner.

7.5 Inter-procedural pattern propagation

The following algorithm models inter-procedural calls and returns and computes the call graph on the fly, based on the name of the called function and the available functions in the caller's scope. For the sake of simplicity, we defer the processing of the `global` operator and the `$GLOBALS` associative array to a further version, presented in section 7.6.
In PHP, there are two distinct scopes: the *local* and the *global* scope. A local variable is defined in a function. On the other hand, global variables appear outside any function. In PHP, the global scope spans all included files, making this kind of operations possible:

```
1 //File includee.php
2 echo $a; //Prints 1;
3 $a = 2;
4
5 //File main.php
6 $a = 1;
7 include(“includee.php”);
8 echo $a; //Prints 2;
```

Listing 7.1: Example of global scope sharing in PHP

The `includee.php` file can print the value of the variable `$a` at line 2 since it was defined in `main.php` and both files share the same global scope. The reassignment of `$a` at line 3 if reflected in `main.php` when the value of `$a` is printed at line 8.
In the presented algorithms, the global scope of a script is represented with the special "main" function. At the inter-procedural level, the global scope of a script thus spans the "main" function of all the scripts it includes. The following program implements inter-procedural pattern propagation:
Algorithm 7.3 uses domains from Algorithm 7.1 and adds the domain P:

- P is the domain of parameter positions. The positions of the actual parameters at call sites and formal parameters in function signatures are represented by integers in the domain of P.

The following relations were introduced to model inter-procedural calls and returns:

DOMAINS
Domains from Algorithm 7.1 plus:

P	256	`positions.map`

RELATIONS
Relations from Algorithm 7.1. plus:

input	*Formal*	(p:P, fn:Fn, fi:F, v:V)
input	*Actual*	(t:Fn, p:P, v:V, c:Fn, f:F)
input	*RetAssign*	(t:Fn, v:V, c:Fn, f:F)
input	*Return*	(t:Fn, f:F, v:V)
input	*Include*0	($f1$:F,$f2$:F)
	Include	($f1$:F,$f2$:F)

RULES
Rules from Algorithm 7.1 plus:

$$Include(f1, f2) \ :- \ Include0(f1, f2). \tag{7.6}$$

$$Include(f1, f2) \ :- \ Include0(f2, f1). \tag{7.7}$$

$$Include(f1, f3) \ :- \ Include(f1, f2), \ Include(f2, f3). \tag{7.8}$$

$$Pat(w, t, f2, l) \ :- \ Actual(t, p, v, c, f1), \ Formal(p, t, f2, w), \ Include(f1, f2), \ Pat(v, c, f1, l). \tag{7.9}$$

$$Pat(w, c, f1, l) \ :- \ RetAssign(t, w, c, f1), \ Return(t, f2, v), \ Include(f1, f2), \ Pat(v, t, f2, l). \tag{7.10}$$

$$Pat(v, \text{`}main\text{'}, f1, l) \ :- \ Pat(v, \text{`}main\text{'}, f2, l), \ Include(f1, f2). \tag{7.11}$$

Algorithm 7.3: Inter-procedural, flow-insensitive, context-insensitive, pattern propagation analysis

- *Formal*: P ×Fn × F × V represents the formal parameters in a function signature. $Formal(p, fn, f, v)$ means that the parameter in position p, in the signature of function fn, in file f, is the variable v.

- *Actual*: Fn × P × V × Fn × F represents the actual parameters at call sites. $Actual(t, p, v, c, f)$ means that the actual parameter that is passed to the target function t, in position p is the variable v in the caller function c in file f.

- *RetAssign*: Fn × V × Fn × F represents the assignment of a return value to a variable. $RetAssign(t, v, c, f)$ means that the return value of the function t is assigned to the variable v in the caller function c, in file f.

- *Return*: Fn × F × V represents function returns. $Return(t, f, v)$ means that the function t in file f returns the variable v.

- $Include0$: F × F represents the initial include operations. $Include0(f1, f2)$ means that the file $f1$ includes the file $f2$. Note that for the purpose of the algorithm, the $Include0$ relation is reflexive: a file always "includes" itself.

- *Include*: F × F represents the transitive closure over the $Include0$ relation. $Include(f1, f2)$ means that the file $f1$ transitively includes the file $f2$. Also note that the *Include* relation is symmetric to model that the global scope is shared between included files.

Rules 7.6 and 7.7 initialize the symmetric *Include* relation. Rule 7.8 computes the transitive closure over the *Include* relation. As a result, the global scope of a file $f1$ is shared with all the files $f2$ for which the predicate $Include(f1, f2)$ is true. Rule 7.9 models the transfer of patterns between actual and formal parameters during a call. The patterns aliased to the actual parameter v in position p, in file $f1$ are transferred to the formal parameter w, in position p, in file $f2$ if the caller and callee names match and if $f1$ includes $f2$. Rule 7.10 models the assignment of a return value to a variable. If the variable w, in function c, in file $f1$ is assigned the result of a call to function t, and t in file $f2$ returns the variable v and $f1$ includes $f2$, then the patterns v must hold are transferred to w.
Rule 7.11 models the sharing of global variables between files. The global scope of a file is represented as the "main" function. Rule 7.11 states that if the variable v in the main function of the file $f2$ holds the pattern l, the variable v in the main function of the file $f1$ must also hold l if $f1$ and $f2$ share the same global scope.

7.6 Alias analysis

Several studies have been devoted to the area of alias analysis [24, 172, 44] and, more recently, [154, 133, 77]. However, the vast majority of alias analyses were developed and optimized for languages such as C or Java, in the context of pointers and references, and it seems unclear how they can be straightforwardly translated for alias analysis of pattern-based properties in PHP programs.
Before going into the details of inter-procedural alias analysis, let's remind that, in PHP, there are two distinct scopes: the *local* the *global* scope. As we mentioned earlier, global variables are shared between the "main" functions all included files. However, global variables can also be accessed from the local scope through the `global` operator or the `GLOBALS` associative array, which is available at every point of the program. Declaring a variable with the `global` operator, makes the variable a reference to its global version. The `$GLOBALS` associative array holds references to all variables in the global scope. The names of the global variables are the keys of this associative array. More information about references in PHP can be found in the PHP manual [135].

7.6.1 Inter-procedural, alias-aware pattern propagation

There are three main operations performed on references in PHP:

1. Assign-by-reference: `$a =& $b`. After assignment by reference, `$a` and `$b` point to the same content in the symbol table.

2. Pass-by-reference: `function foo(&$var)`. After the call to function `foo`, the actual parameter and the formal `var` parameter will point to the same content in the symbol table.

3. Return by reference: `function &foo($var)`. The value returned by `foo` will be a reference.

That being said, it is clear that pattern propagation depends on alias analysis to produce sound results. Indeed, whenever a variable is assigned a pattern, all the variables pointing to the same content in the symbol table will point to the new pattern. Algorithm 7.4 implements an inter-procedural, alias-aware, pattern propagation algorithm.
Algorithm 7.4 inherits all the domains from Algorithm 7.3 and introduces these new relations:

- Ref: V $\times$ Fn $\times$ F $\times$ V represents the assign-by-reference operations. $Ref(v, fn, f, w)$ states that the variable w is assigned by reference to the variable v in function fn, in file f.

DOMAINS
Domains from Algorithm 7.3.

RELATIONS
Relations from Algorithm 7.3. plus:

input	*Ref*	(v:V,fn:Fn,f:F,w:V)
input	*RFormal*	(p:P, fn:Fn, fi:F, v:V)
input	*RRetAssign*	(t:Fn, v:V, c:Fn, f:F)
input	*RReturn*	(t:Fn, f:F, v:V)
input	*Global*	(v:V, fn:Fn, f:F)
	*Al*0	(v:V,$t1$:Fn,$f1$:F,w:V,$t2$:Fn,$f2$:F)
	Al	(v:V,$t1$:Fn,$f1$:F,w:V,$t2$:Fn,$f2$:F)

RULES
Rules from Algorithm 7.3. plus:

$$Al0(v, fn, f, w, fn, f) \ :- \ Ref(v, fn, f, w). \tag{7.12}$$

$$Al0(w, t, f2, v, c, f1) \ :- \ Actual(t, p, v, c, f1), \ RFormal(p, t, f2, w), \ Include(f1, f2). \tag{7.13}$$

$$Al0(w, t, f2, v, c, f1) \ :- \ RRetAssign(t, w, c, f1), \ RReturn(t, f2, v), \ Include(f1, f2). \tag{7.14}$$

$$Al0(v, t, f, v, \text{`}main\text{'}, f) \ :- \ Global(v, t, f). \tag{7.15}$$

$$Al(v, t1, f1, w, t2, f2) \ :- \ Al0(v, t1, f1, w, t2, f2). \tag{7.16}$$

$$Al(w, t2, f2, v, t1, f1) \ :- \ Al0(v, t1, f1, w, t2, f2). \tag{7.17}$$

$$Al(v, t1, f1, x, t3, f3) \ :- \ Al(v, t1, f1, w, t2, f2) \ Al(w, t2, f2, x, t3, f3). \tag{7.18}$$

$$Pat(v, t1, f1, l) \ :- \ Al(v, t1, f1, w, t2, f2), \ Pat(w, t2, f2, l). \tag{7.19}$$

Algorithm 7.4: Alias-aware pattern propagation analysis

- *RFormal*: P × Fn × F × V represents the formal parameters that are passed by reference in a function signature. $RFormal(p, fn, f, v)$ means that the parameter in position p, in function fn, in file f is the variable v and is passed by reference.

- *RRetAssign*: Fn × V × Fn × F represents the assignment by reference of a return value to a variable. $RRetAssign(t, v, c, f)$ means that the return value of the function t is assigned by reference to the variable v in the caller function c, in file f.

- *RReturn*: Fn × F × V represents return-by-reference operations. $Return(fn, f, v)$ means that the function fn in file f returns a reference to the variable v.

- *Global*: V × Fn × F represents variables that reference the global scope. Either they are declared with the `global` keyword or they are accessed from the `$GLOBALS` associative array. $Global(v, fn, f)$ means that the variable v in function fn, in file f is a global variable.

- *Al*0: V × Fn × F × V × Fn × F represents the initial aliases, induced either by assign-by-reference, call-by-reference, return-by-reference or the declaration of a reference to a global variable. $Al0(v, t1, f1, w, t2, f2)$ means that the variable v in function $t1$ in file $f1$ is aliased to the variable w in function $t2$, in file $f2$.

- *Al*: V × Fn × F × V × Fn × F represents the final alias relations. Conceptually, aliased variables can be represented as alias sets where each member of an alias set is aliased to every other member of the set. The *Al* relation is computed from *Al*0 and represents the alias sets in the program. $Al(v, t1, f1, w, t2, f2)$ means that the variable v in function $t1$ in file $f1$ is aliased to the variable w in function $t2$, in file $f2$.

Rule 7.12 initializes the *Al*0 relation. Whenever a variable w is assigned by reference to a variable v, the two variables are aliased. Rule 7.13 models the alias relationship that is created between formal and actual parameters on call-by-reference operations in a similar manner to Rule 7.9. Rule 7.14 models the alias relationship that is created on return-by-reference operations, similarly to Rule 7.10. Rule 7.15 models the alias relation that is created through the use of the `global` operator of the `$GLOBALS` associative array. When a variable v is declared global in function t, in file f, v is aliased with its global version: the variable v in the "main" function of file f. Rules 7.16 to 7.18 compute the set of all ordered pairs of aliases from the *Al*0 relation. The resulting *Al* relation represents the sets of aliases in the system. Rule 7.19 propagates patterns through alias sets. If the variable $v1$ in function $t1$, in file $f1$, is aliased to the variable $v2$ in function $t2$, in file$f2$ and $v2$ must hold the pattern l, then $v1$ must also hold l.

7.7 Uses of pattern propagation for the analysis of PHP programs

In this section, we briefly introduce some uses of pattern propagation for static analysis of PHP programs. Apart from security checks, we show how pattern propagation also proves useful to model the propagation of literals in PHP applications. First, we show how security checks can be identified with syntactic patterns. We then use a simple example to explain how pattern propagation increases the precision of security check detection. Afterwards, we review how pattern propagation also helps approximate the literals a variable may hold and how this information can enhance the precision of static analysis of PHP programs.

7.7.1 Pattern propagation and security checks

Access control models in Web applications are typically enforced with heavily stereotyped security checks that are easily detected at the source code level with application-dependent syntactic patterns. In a previous study [60], we proposed a technique for the extraction of access control models in PHP applications. In summary, this technique extracts, for each statement, some of the permissions a user must own to execute the statement. In order to produce such results, our technique first identifies the security checks in a program based on application-dependent syntactic patterns. Listings 7.2 to 7.4 show examples of security checks in three different PHP applications.
As can be observed, security checks vary from one system to another. However, one can observe that these these checks either call special functions or verify specific variables. Identifying security checks at the abstract syntax tree (AST) level thus requires a few syntactic patterns that can be customized from one application to the other. To facilitate the pattern propagation analysis, we summarize security checks as the permission they verify. For example, in Listing 7.3, the `$_SESSION["admin"] == "yes"` security check would be summarized as the "admin" permission.

```
1 require_capability('moodle/site:uploadusers');
2 // Stops the execution if the current user doesn't have the 'moodle/site:uploadusers' capability
3
4 if (has_capability('moodle/course:viewparticipants')) {
5     // Do something if the current user has the 'moodle/course:viewparticipants' capability
6 }
```

Listing 7.2: Examples of a security checks in Moodle

```
1 if ( $_SESSION["admin"] == "yes" ) {
2     // Do something if the current user is the administrator
3 }
```

Listing 7.3: Example of a security check in PHP Calendars

```
1 if (check_admin_login()) {
2     // Do something if the logged in user is an administrator
3 }
```

Listing 7.4: Example of a security check in YaPiG

```
1 $can_upload = has_capability('moodle/site:uploadusers');
2 if ($can_upload) {
3     // Do some operation protected by the 'moodle/site:uploadusers' permission
4 }
```

Listing 7.5: Example of a security check in Moodle that require literal analysis.

Detection of security checks in this manner is however limited to cases where:

1. The security check is performed in a conditional statement.
2. All the components of the security check (e.g. the permission name, the role name or the expected variable value) are static literals.

To illustrate our point, consider the following snippet of code in Listing 7.5. On the one hand, syntactic patterns can detect the `has_capability('moodle/site:uploadusers')` security check. However, pattern propagation is required to determine that `$can_upload` is conceptually aliased to the "moodle/site:uploadusers" permission.

7.7.2 Pattern propagation and literals

Unlike in languages like C, C++ or Java, file inclusion targets in PHP are not *required* to be represented with static literals. In PHP, any arbitrary expression that evaluates to a string can be used as a file inclusion target. In this context, pattern propagation can help approximate the target of dynamic include statements. Consider the following example: The target of the dynamic include statement in Listing 7.6 is revolved at run time based on the literal values of `$ROOT` and `$lang`. Without further information, static analysis must conservatively assume that this statement can include *any* file in *any* "templates" directory.

```
1 //Inclusion of a file based on a dynamically built string
2 include($ROOT . '/templates/' . $lang);
```

Listing 7.6: Example of a dynamic include statement

From a pattern propagation perspective, literals can be viewed as extremely simple syntactic patterns that are summarized as their literal value. From that perspective, we see that the presented algorithms can be used for the propagation of literals through a program. Using literal information, it is possible to approximate the target of dynamic include statements and thus improve the precision of various static analyses for PHP applications. Section 7.8 shows how pattern propagation helps approximate the target of dynamic include statements in the investigated applications.

The PHP language also supports *variable* variables and functions. In such cases, the name of the variable or the function is resolved at runtime through the evaluation of a potentially arbitrary expression that returns a string literal. PHP also support the use of the `eval(arg)` function where `arg` is an arbitrary expression that returns a string literal. On calls to `eval`, the PHP interpreter executes the content of `arg` as if it was PHP code. Pattern propagation analysis can be used to approximate the content of `eval` arguments as well as the target of *variable* variables and functions.

7.7.3 Limitations

Our current implementation neither supports arrays, nor object-oriented features of PHP. Uses of arrays, object member variables and methods are treated in a pessimistic way, meaning that all the variables implied in such constructs are assigned the special "unknown" value, to represent that the analysis lost track of the patterns this variable might hold. On the other hand, calls to the `eval` function, *variable* variables and *variable* functions are treated in an optimistic way, meaning that these constructs are considered not to affect pattern analysis. The pessimistic, overly conservative approximation, would have been to consider that these constructs can potentially alter *any* variables and call *any* functions. The presented analyses are thus of the *must* type since the extracted information is true for *some* program executions. A discussion about *may* and *must* analyses is presented in [66].

7.8 Experimental results

From the experimental perspective, we performed two kind of investigations. First, we investigated how pattern propagation analysis can help track security checks in applications

that implement an access control models. In section 7.8.1 we compare the performances of pattern propagation algorithms for the detection of security checks.
We also investigated how pattern propagation analysis can improve the static approximation of dynamic include targets. Algorithms 7.1 to 7.4 were applied to eight PHP applications. Section 7.8.3 presents the resolution rates before/after pattern propagation analysis for each applications and each algorithm.
Table 7.1 shows characteristics about investigated applications. The version numbers are listed together with the application name, when available. The reported numbers of lines of code exclude blank lines and comments and were calculated with the cloc.pl software [43].

7.8.1 Security check detection

As illustrated in Listing 7.5, pattern propagation analysis can model the propagation of security checks through a program. Table 7.2 reports the results of using pattern propagation analysis to model the propagation of security checks in investigated applications.
Numbers in table 7.2 represent the number of detected security checks. The *baseline* column shows the number of security checks that were detected with syntactic patterns only. The remaining four columns report the additional security checks that were detected by pattern propagation analysis. The *intra f-i* column refers to the intra-procedural, flow-insensitive algorithm. The *intra f-s* column shows results for the intra-procedural, flow-sensitive algorithm. Column *inter* reports results for the inter-procedural algorithm without aliases and column *Inter al.* shows results for the inter-procedural algorithm with processing of aliases.
Syntactic patterns are sufficient to detect all the security checks in all investigated applications but PHPoll and Moodle. Intra-procedural, flow-insensitive pattern propagation is sufficient to detect all the security checks in PHPoll with an execution time of 0.017 second.

Table 7.1: Characteristics of the evaluated applications

Application	Files	LOC	
		PHP	HTML
SCARF	25	1,318	0
Events Lister 2.03	37	2,076	544
PHP Calendars	67	1,350	0
PHPoll 0.97	93	2,571	0
PHP iCalendar 1.1	183	8,276	0
AWCM 2.1	668	12,942	5,106
YaPiG 0.95	134	4,801	1,271
Moodle 1.9.5	**5124**	**404,399**	**30,547**

Table 7.2: Number or detected security checks with each pattern propagation algorithms.

Application	Baseline	Algorithm			
		Intra f-i	Intra f-s	Inter	Inter al.
SCARF	16	16	16	16	16
Events Lister 2.03	12	12	12	12	12
PHP Calendars	2	2	2	2	2
PHPoll 0.97	0	3	3	3	3
PHP iCalendar 1.1	1	1	1	1	1
AWCM 2.1	1	1	1	1	1
YaPiG 0.95	8	8	8	8	8
Moodle 1.9.5	992	1062	1063	1072	1072

For Moodle, syntactic patterns alone detected 992 checks. Intra-procedural, flow-insensitive analysis detects 70 supplementary checks in 0.098 second. The intra-procedural, flow-sensitive algorithm detects one more security check with an execution time of 105.958 seconds. Inter-procedural analysis detected 9 more security checks in 15.661 seconds. The alias-aware version executed in 26.542 seconds and did not detect any new security check.

7.8.2 Precision and recall

We manually calculated the precision of our algorithms for all investigated applications. Indeed, we confirmed, through code inspection, the validity of *all* the reported security checks. Moreover, apart from Moodle, we manually calculated the recall of our algorithms for all the applications. Through code inspection, we certified that our approach detected *all* the security checks in investigated programs. Moodle was omitted due to its size and complexity level. Overall, our approach achieved 100% precision and 100% recall for 7 applications out of 8 in terms of security check detection.

In the case of Moodle, our intra-procedural, flow-insensitive, pattern propagation analysis produced 1 false positive. In that particular case, the same variable is assigned the result of two different security checks. However, only one "version" of the variable is used. The intra-procedural, flow-sensitive pattern propagation analysis removes this ambiguity. Inter-procedural analysis detected 9 additional security checks and introduced 2 false positives.

In summary, the precision of our security check detection algorithms varies between 96% and 100%. Overall, when combined with syntactic patterns, the precision of our security check detection algorithms varies between 99.7% and 100%. Recall could not be calculated for Moodle due to its size and complexity.

7.8.3 Resolving includes

Table 7.3 shows resolution rates of include targets before/after pattern propagation analysis. The baseline results, before pattern propagation, were generated with an "intelligent" include resolution algorithm that processes commonly used built-in functions like `dirname()`, and built-in constants like `__DIR__` and `__FILE__`. Results show that our approach resolves 81% to 100% of the include statements. Previous work by Jovanovic et al. [89] reported include resolution rates between 72% and 100% on seven PHP applications.

Table 7.3 shows that include statements in SCARF and Events Lister can all be resolved without pattern propagation analysis. In the case of PHP Calendars, inter-procedural pattern propagation is required to resolve all includes. Manual reviews revealed that the unresolved includes in PHPoll point to inexistent files. PHP iCalendar makes heavy use of constants to generate include statements. Moreover, in PHP iCalendar, the constants that are used in dynamic include statements are always defined locally, before the include statement. Intra-procedural analysis is thus sufficient to resolve almost all includes in the program. The unresolved includes in AWCM are built dynamically based on database values, a feature our algorithms do not yet support. YaPiG and Moodle use constants in a similar manner to AWCM to dynamically generate include targets and inter-procedural pattern propagation yields the best gains in precision.

Unresolved include statements in YaPiG and Moodle are generated based on constructs that are beyond the scope of our pattern propagation analyses: classes, user inputs and database values. For all investigated applications, flow-sensitivity and treatment of aliases did not yield better resolution of include statements.

Table 7.4 shows the execution times for the propagation of literals with each pattern propagation algorithms. Times are in seconds unless indicated otherwise. All the analyses were run on a computer with Intel Core2 Duo 3.0GHz processors and 4 GB of RAM. Columns names refer to the presented algorithms.

Since no assign-by-reference, pass-by-reference or return-by-reference were detected in PHPoll and AWCM, we did not run the alias sensitive algorithm on these programs. All the reported times exclude I/O operations and correspond to the time spent solving Datalog rules in the `bddbddb` framework. Overall, Table 7.4 shows that the implemented analyses are fast and scale to medium-large applications. In the case of Moodle, some atypical structures in the program slow down the alias-aware propagation of literals.

Table 7.3: Include resolution rates in percentage (%) with different pattern propagation algorithms.

Application	Baseline	Algorithm			
		Intra f-i	Intra f-s	Inter	Inter al.
SCARF	100	–	–	–	–
Events Lister 2.03	100	–	–	–	–
PHP Calendars	97	97	97	100	100
PHPoll 0.97	96	96	96	96	96
PHP iCalendar 1.1	24	98	98	100	100
AWCM 2.1	95	95	95	95	95
YaPiG 0.95	49	49	49	87	87
Moodle 1.9.5	55	56	56	81	81

Table 7.4: Execution times in seconds for the propagation of literals with different pattern propagation algorithms.

Application	Algorithm			
	Intra f-i	Intra f-s	Inter	Inter al.
SCARF	0.008	0.587	0.379	0.398
Events Lister 2.03	0.093	0.793	0.336	0.447
PHP Calendars	0.206	0.979	0.445	0.569
PHPoll 0.97	0.118	0.992	0.354	–
PHP iCalendar 1.1	0.160	0.39451	0.736	0.810
AWCM 2.1	0.126	7.566	0.905	–
YaPiG 0.95	0.101	0.825	0.702	0.728
Moodle 1.9.5	0.531	105.958	567.514	2h54m48s

7.9 Discussion

In this paper, we presented four pattern propagation analysis algorithms: intra-procedural and flow-insensitive, intra-procedural and flow-sensitive as well as inter-procedural algorithms with and without processing of aliases. To our knowledge, we are among the first to propose such algorithms for the PHP language.

All the algorithms introduced in this paper are Datalog programs readily reusable in the `bddbddb` framework or any Datalog engine. Data-flow analyses are typically tedious to implement, especially when no framework is available for the target language, as is the case for PHP. Moreover, implementations are typically hard to share as they are usually tied to a specific front-end. For these reasons, we were won over by the expressiveness, compactness and simplicity of the Datalog language to express data-flow analyses and we hope the reader shares our enthusiasm.

From the results perspective, Table 7.3 shows that our pattern propagation analyses can resolve up to 76% more include statements when compared to baseline results. Given the highly dynamic nature of the PHP language, we were pleased to observe that our algorithms performed so well, despite their approximated nature. In this case, there seems to be a dichotomy between the possibilities the language offers and the features that are used in practice. Indeed, in the case of include statements, developers seem to make a *wise* use of the dynamic features of PHP, leading to good resolution rates by our algorithms. In other words, while PHP does not limit the complexity of the expressions in an include statement, it seems that developers naturally adopted *good practices* that ease their resolution.

Interestingly, when it comes to security checks, our algorithms also performed very well. In the context of security, the motto seems to be: keep it simple. For 6 applications out of 8, we manually validated that all the security checks were detected using syntactic patterns only. In the case of PHPoll, an intra-procedural, flow-insensitive analysis was sufficient to detect all the security checks. In the case of Moodle, the vast majority (992) of identified security checks (1072) are detected with syntactic patterns. Intra-procedural analyses detected 71 supplementary security checks and inter-procedural analysis detected 9 more.

The results show that when it comes to security checks, developers seem to attach importance to simplicity. For example, they often prefer to explicitly check the same permission repeatedly rather than storing the result of the security check in a variable that will be checked multiple times. Moreover, at the inter-procedural level, we only observed a few cases where security checks were passed as parameters. These implicit, unofficial coding practices greatly simplify the analyses in practice and yield good results with the proposed algorithms.

7.10 Related work

Jovanovic et al. [88, 89] developed the Pixy tool for the detection of cross-site scripting vulnerabilities in PHP applications. Pixy uses an inter-procedural, context and flow-sensitive analysis to propagate taint information and detect vulnerabilities. Pixy also uses a literal analysis algorithm to resolve include statements with a precision ranging from 72% to 100%. While the investigated applications differ, our approach performed slightly better, with a precision between 81% and 100%. Also, while our algorithms process the whole PHP program at once, their analysis has to be run for each and every entry point of the application.

Recently, Son et al. [152] introduced RoleCast, a tool that is specifically designed for access control vulnerabilities detection. RoleCast statically infers the access control checks by analyzing the variables that are usually verified before the execution of security-sensitive operations. To overcome the difficulties induced by the static analysis of PHP applications, they convert PHP applications to Java programs and perform several post-processing operations to produce well-formed Java files. While our approach relies on application-dependent patterns to identify security checks, their approach relies on numerous assumptions about the design and implementation of access control models. Whether or not these assumptions are always verified in practice is unclear. Moreover, the authors do not clearly address the issues of mapping back the information from the generated Java code back to the PHP program.

Work by Sun et al. [156] relies on string analysis techniques [168, 169] to identify access control vulnerabilities in PHP applications. Their analysis also relies on application- dependent patterns to identify security checks. Furthermore, their approach requires the *manual* specification of the application's "critical states." According to the authors, a critical state includes information such as: session values, cookie values, request parameter values, database records, variable values or function return values. For medium to large applications, this seems like an unattainable goal. We think, however, that our pattern propagation algorithms might be useful in this context, to help developers identify potential values for critical states.

In [4] Alalfi et al. proposed a framework, based on static and dynamic analysis, for the reverse-engineering of access control models in PHP applications. The models are extracted in secureUML and suitable for verification and validation.

Several studies have been devoted to the area of alias analysis [24, 172, 44], and more recently [154, 133, 77]. However, the focus of most of these studies was on typed languages like C and Java. The semantic differences between PHP and typed languages made the existing analyses difficult to reuse.

The `bddbddb` framework was previously used by Livshits and Lam to perform security analyses of Java applications [106]. Their goal was to detect SQL injection and cross-site scripting

vulnerabilities in Web applications using static analysis.

7.11 Conclusion

In this paper we presented several pattern propagation algorithms, tailored for PHP applications. All the algorithms were implemented as Datalog programs and can be reused as-is, provided that a PHP front-end is available to supply the inputs to the programs. Our study confirmed previous claims about the simplicity and effectiveness of Datalog to implement reputably complex static analyses.

While the main goal of this paper was to improve to the precision of security check detection through pattern propagation, we showed how the proposed algorithms could also be applied to other problems, like the resolution of dynamic include statements. Empirical results revealed that, for the investigated problems, context and flow-insensitive algorithms yield very precise results in a much faster manner than context and flow-sensitive approaches.

In conclusion, we showed how the proposed algorithms propagate simple patterns in a fast and precise manner. In subsequent work, we plan to address the current limitations of our approach. We also plan to investigate the impact of operations, other than assignement, on patterns. For example, we observed that security checks are sometimes combined with Boolean operations. It would be interesting to model the impact of such operations on patterns and provide more insights about the source code.

Acknowledgements

This research is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program.

CHAPTER 8

CORRECTIONS FOR PAPER 2

The previous chapter contains the published version of the paper entitled: "Alias-Aware Propagation of Simple Pattern-Based Properties in PHP Applications". The current chapter addresses comments and corrections by the jury regarding this paper.

In section 7.2, we stated that: "Datalog is a deductive database language that is conceptually related to Prolog." In fact, Datalog is a *subset* of Prolog.

In section 7.4, we mentioned that: "Results however show that flow-insensitivity does not hinder the quality of the results in a significant manner." and we were asked why flow-sensitivity was needed earlier since it does not seem to significantly impact results. The fact is that we did not know *a priori* whether or not flow-sensitivity would have a significant impact. After determining that it did not significantly impact the quality of the results, we decided to pursue our experiments with flow-insensitive algorithms only.

In algorithm 7.3, at rule 7.8, a member of the jury remarked that the first "Include" could be replaced by "Include0" to eliminate a source of recursion.

In subsection 7.7.3, we mentioned that: "our current implementation neither supports arrays, nor object oriented features of PHP" and a member of the jury asked to elaborate on impacts over the false positive rate.

In the context of this study, we tracked the propagation of privilege checks throughout variables and parameters. In this context, false positives refer to cases where a variable or parameter is reported to contain the result of a privilege check when it's not. In this study, we manually verified each and every variable and parameter that was reported to contain the result of an access check and determined that every result was correct, for a false positive rate of 0%. Intuition dictates that if we were to support arrays or object-oriented features, the false positive rate would significantly increase. However, we performed no experiment to validate this intuition. In the context of this study, the fact that we stop propagating security patterns whenever we encounter an unsupported structure mainly impacts the false negative rate.

In subsection 7.8.2, we mentioned that the precision of pattern propagation algorithms varied between 96% and 100% and jumped to 99,7% to 100% when combined with syntactic patterns. A reviewer asked how syntactic patterns could improve precision. The reason is simple. The precision of pattern propagation algorithms was calculated based only on the *new* security checks that were detected and ignoring those that were already detected by syntactic patterns.

Since syntactic patterns always achieve a precision of 100%, the precision increases when we combine both approaches.

CHAPTER 9

PAPER 3: INVESTIGATION OF ACCESS CONTROL MODELS WITH FORMAL CONCEPT ANALYSIS: A CASE STUDY

ABSTRACT

Web applications manage increasingly large amounts of sensitive information and often need to implement access control (AC) models. However, documentation about the implemented AC model is often sparse and few, if no tool exists to support AC model investigation. Based on the results of a previous study, we show how formal concept analysis (FCA) can support the understanding and visualization of reverse-engineered AC models. Results of applying FCA to Moodle, a medium-sized (625 473 LOC) Web application, are presented and discussed. We show how FCA enhances the overall comprehension of reverse-engineered AC models and sheds light on previously unknown features of Moodle's AC model.

9.1 Introduction

Every day, millions of people communicate, shop, bank, gather information, and perform numerous common tasks using web applications. Those applications must restrict access to private and security-sensitive information they routinely handle such as: credit card numbers, addresses, and financial data.

Role-Based Access Control (RBAC) models [143], have been widely adopted by the industry, in part because RBAC models simplify user management. Instead of managing a large number of user-permissions relationships, administrators can grant users sets of pre-defined permissions, called *roles* [35].

On one hand, an RBAC model, in its simplest form, is defined based on a set of binary relations: a *user* owns a set of *roles*, a *role* owns a set of *permissions* and a *permission* associates an *operation* to an *object*. More sophisticated RBAC models will sometimes introduce the concepts of inheritance or constraint relationships between roles. On the other hand, formal concept analysis, (FCA) takes a binary relation as input and identify clusters of objects that share identical attributes before ordering them in a Galois lattice [59]. The two approaches are obviously complementary.

Many researchers indeed suggested FCA-based approaches to support the design [117, 46, 126] of RBAC models. However, none of these approaches tackles the problem of investigating

the implemented access control (AC) model when no specifications are available. Whereas many web applications (e.g. Moodle, Drupal, phpBB and Wordpress) [1] implement RBAC-like access control models, specification and documentation of the implemented models are often sparse.

9.1.1 Contributions

To our knowledge, FCA has never been used for the investigation of AC models. Furthermore, while some previous studies tackled the problem of extracting [60, 96, 167] AC models from source code, we are among the first to target the investigation of the extracted models. The main contributions of this work are:

- A novel approach to AC model investigation.
- An experimental study of applying FCA to Moodle, a medium-sized PHP application with a sparsely documented AC model.
- A discussion about the knowledge that can be extracted through FCA of reverse-engineered AC models.

9.2 Access Control Models

9.2.1 Formal RBAC models

In RBAC, an administrator assigns permissions to roles and roles to users. According to the ANSI standard defining RBAC models [2], a *core RBAC* model consists of the following elements: $USERS$, $ROLES$, OPS, OBS; the sets of users, roles, operations and objects respectively and the following four mappings: UA, PA, Op and Ob; the user-role, permission-role, permission-to-operation and permission-to-object mappings respectively. *Hierarchical RBAC* introduces the notion of permission inheritance through a role lattice while a *constrained RBAC* restricts the allowed combinations of roles.

9.2.2 Access control models in Web applications

Open source Web applications often implement a simplified form of RBAC model where *operations* and *objects* are merged together. For example, the action of reading a file in a classic RBAC model would be implemented with a permission that grants the *read* operation on the $file$ object. In open source Web applications, the same action would often be implemented with a simple $read\_file$ permission.

[1] See: http://moodle.org, http://drupal.org, http://www.phpbb.com and http://wordpress.org
[2] See ANSI-INCITS 359-2004, February 2004

Therefore, we characterized AC models in Web applications with three mappings instead of four: user-to-role, role-to-permission and permission-to-statements. The later replaces the Op and Ob mappings in the *core RBAC* model and represents the statements that a permission protects.

9.3 Extraction of AC models from source code

In this section, we present the methodology we use to extract the AC model from the source code of a Web application. This section is a summary of the methodology used and presented in a previous paper [60].

9.3.1 Extraction of role-to-permission relationships

The storage of role-to-permission relationships is specific to every Web applications and their extraction requires manual operations. However, for all the investigated applications, the set of manual operations was limited to: parsing an HTML table from the "admin" console or launching a simple query to the application's database.

9.3.2 Extraction of permission-to-code relationships

In the absence of security specifications or informal description of a permission, all is left to understand the effect of a permission is to observe the source code it protects. In a previous study [60], we presented an approach to extract necessarily enabled capabilities ($NECs$) from PHP source code. $NECs$ represent the permissions a user must have to execute a statement. Our method for NEC extraction consists of four steps: identification of access control patterns, extraction of an inter-procedural control flow-graph (CFG), conversion of the CFG to a set of formal automata and model checking of the automata to extract $NECs$.

Access control patterns

Access control routines are used to assert that a user owns a given permission. More precisely, these routines verify that the user belongs to at least one role such that the role owns the permission. Identification of access control routines is thus a crucial step for the extraction of $NECs$. Listings 1 to 3 show examples of access controls routines in three different PHP applications.

```
require_capability('moodle/site:uploadusers');
// Stops the execution if the current user doesn't have the 'moodle/site:uploadusers' capability

if (has_capability('moodle/course:viewparticipants')) {
    // Do something if the current user has the 'moodle/course:viewparticipants' capability
}
```

Listing 9.1: Example of access control routines in Moodle

```
if ( $user->has_cap( 'publish_posts' ) ) {
    // Do something if $user has the ''publish_posts'' capability
}

if ( current_user_can( 'edit_post' ) ) {
    // Do something if the current user has the ''edit_post'' capability
}
```

Listing 9.2: Example of access control routines in Wordpress

```
if (user_access('post comments')) {
    // Do something if the current user has the ''post comment'' capability
}
```

Listing 9.3: Example of an access control routine in Drupal

As can be observed, the routines vary from one system to another. However, we see that all these routines receive a string argument, representing the permission, and return a Boolean value asserting whether or not the user can perform the corresponding action. From a syntactic point of view, these functions are almost identical. Therefore, identifying access control routines at the abstract syntax tree (AST) level requires a single access control pattern, with minor customizations from one application to the other.

Control flow graph

Starting from the AST, an inter-procedural CFG is extracted:

$$CFG = (V_{CFG},\ E_{CFG}) \tag{9.1}$$

with a unique *entry* node $v_{in} \in V_{CFG}$ and a unique *exit* node $v_{out} \in V_{CFG}$. Nodes in V_{CFG} can be of type *generic*, *call_begin*, *call_end*, *entry*, or *exit*. Nodes of type *generic* are involved

in intra-procedural control flow; nodes of type *call_begin*, *call_end*, *entry*, and *exit* are used in inter-procedural control flow.
Edges in E_{CFG} can be of type *generic*, *allow_*c_i, *prevent_*c_i, *call*, or *return*. Edges of type *generic* represent intra-procedural transfers of control, edges of type *allow_*c_i represent intra-procedural transfers of control that enable a capability c_i, likewise *prevent_*c_i are edges that disable a capability c_i and edges of type *call* and *return* represent inter-procedural control flow links.
Edges of type *allow_*c_i and *prevent_*c_i are created according to the access control routines that were identified in the previous step. For example, observe Listing 9.3. In that case, the edge representing the *true* branch of the if statement would be of type *allow_"post comments"*. Conversely, the edge representing its $false$ branch would be of type *prevent_"post comments"*.

Model checking

Software model checking [33] is the algorithmic analysis of programs to prove properties of their executions.
In order to extract permission-to-code relationships, the CFG is converted into several model checking automata, each representing a single capability. A custom model checker then process each automaton in order to identify the statements *stmt* for which the following equation hold:

$$\Diamond stmt_j \wedge \Box(\ stmt_j \wedge c_i) \tag{9.2}$$

meaning that the automaton eventually reaches a statement $stmt_j$ ($\Diamond stmt_j$) and always when $stmt_j$ is reachable, the "capability" c_i is allowed ($\Box(\ stmt_j \wedge c_i)$). Statements satisfying Equation 9.2 are *only* reachable by an execution for which the capability c_i is enabled. In this case, the capability c_i is a NEC for the statement $stmt_j$. The AC model extraction technique is described in more details in [60].
Using this methodology we have successfully reverse-engineered the $NECs$ from Moodle, a medium-sized course management system totaling 625 473 LOC across 2331 PHP files.
In the context of this paper, the NEC-to-statements relationship is equivalent to the permission-to-statements relationship, as presented in subsection 9.2.2.

9.4 Formal concept analysis

Formal concept analysis builds a concept lattice (or Galois lattice) from a matrix, called a formal context, representing binary relations between *objects* (not to be confused with RBAC *objects*) and *attributes* [59]. Below, we present the basic definitions necessary for the

comprehension of FCA.
A formal context is a triple (G, M, I) where G and M are sets of *objects* and *attributes* respectively. $I \subseteq G \times M$ is a binary relation between G and M. For $g \in G$ and $m \in M$, $gIm \rightarrow (g, m) \in I$: there exists a binary relation between g and m.
A formal concept of the context (G, M, I) is a pair (X, Y), where $X \subseteq G$ and $Y \subseteq M$ satisfy the following properties:

- $X = \{g \in G \mid (\forall m \in Y)\ gIm\}$, i.e., X is the set of all *objects* that share all *attributes* in Y.
- $Y = \{m \in M \mid (\forall g \in X)\ gIm\}$, i.e., Y is the set of all *attributes* shared by all *objects* in X.

In the Galois lattice resulting from FCA, concepts are partially ordered according to the following partial order relation: $(X_1, Y_1) \preceq (X_2, Y_2)$ if $X_1 \subseteq X_2$ or equivalently if $Y_1 \supseteq Y_2$. In FCA terminology, X and Y are often referred to as the *extent* and the *intent* of the concept respectively.
Also, we say that a concept c is labeled with an *object* $g \in G$ if c is the smallest concept in which g appears. Conversely, we say that a concept c is labeled with an *attribute* $m \in M$ if c is the largest concept in which m appears. In both figures 9.1 and 9.2, a circle represents a concept and will have its lower half colored in black if it has an *object* label. Conversely, it will have its upper half colored in blue if it has an *attribute* label.
Starting from a concept lattice, it is possible to derive a base of implications: rules that hold for the whole formal context. In this paper, implications were calculated based on Duquenne-Guiges methodology [71]. Concept lattices and Duquenne-Guiges implications were generated with the Concept Explorer software [177].

9.5 Results and discussion

In this section, we present the preliminary results obtained by performing FCA on Moodle's reverse-engineered AC model. Results will be discussed as they are presented.

9.5.1 Analyzing role-to-permissions mapping

Moodle's role-to-permissions formal context comprises 7 *objects* (roles) and 145 *attributes* (permissions). Fig. 9.1 shows the Galois lattice obtained by performing FCA on that context. Nodes in the lattice represent formal concepts; pairs (R,P) where each $role \in R$ owns all the $permission \in P$ and each $permission \in P$ belongs to every $role \in R$. Either R or P can be empty.

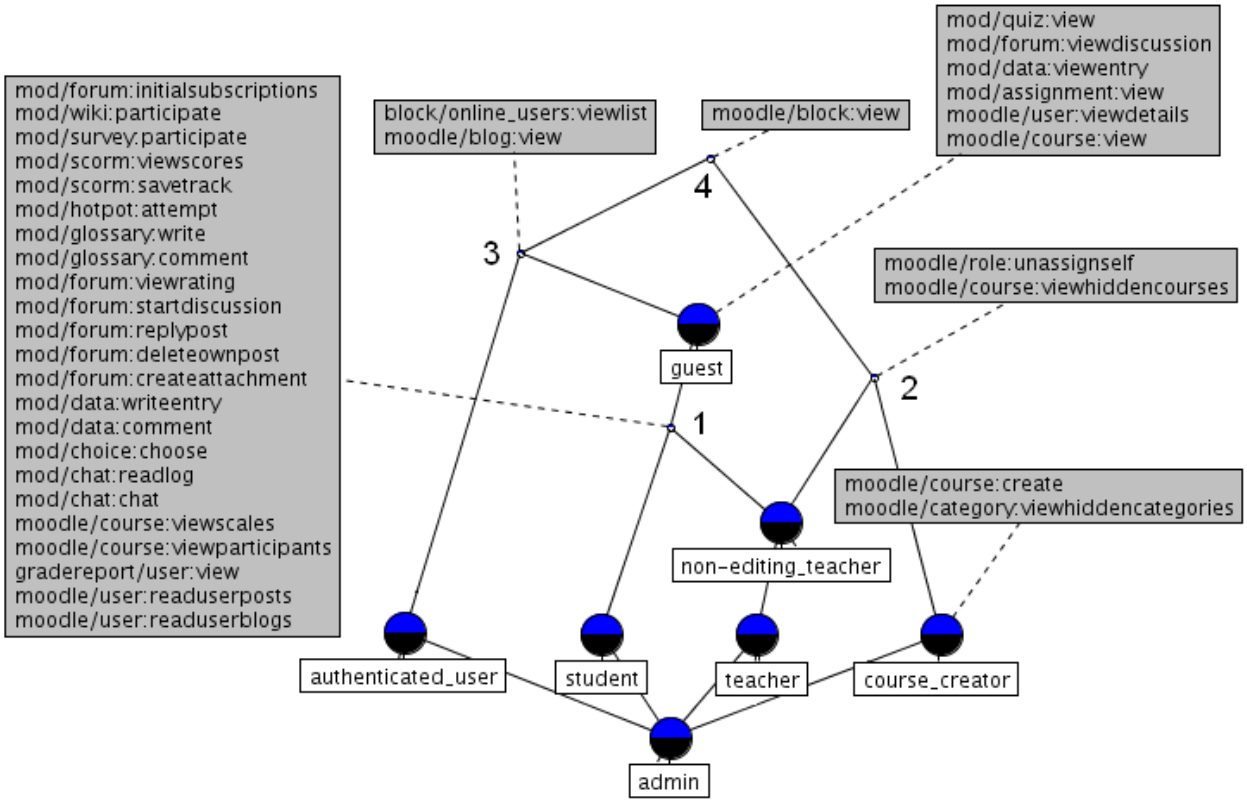


Figure 9.1: Galois lattice representing Moodle's role-to-permission mappings. Concepts represented with bigger circles have a role label (white rectangles). Selected permission labels (grayed rectangles) are also displayed.

In the lattice of Figure 9.1, the *intent* of the concept at the bottom, the infimum, is the set of all 145 permissions and corresponds to the admin role. Conversely, the *extent* of the concept at the top, the supremum, is the set of all 7 roles.
Due to space limitations, we only displayed selected permissions labels (grayed rectangles) since some of them can comprise more than 100 permissions. As can be expected, no role shares the same concept in the lattice. However, it is interesting to observe that there exists concepts with no associated role. We say that such concepts represent *implicit* roles; roles that are inherent to the formal context but that don't have a name. An interesting feature of implicit roles is that they can be added to the AC model without modifying the structure of the lattice.

Investigating implicit roles

While they don't have a name, *implicit* roles undoubtedly have a meaning; they reflect, in some ways, the need to have some *explicit* roles share specific permissions. Moodle has four

implicit roles, numbered from 1 to 4 on Figure 9.1. Let's investigate them in more details:
Implicit role 1: The assistant. The first implicit role is situated at the intersection between a student, a non-editing teacher and a guest in the lattice. Contrary to the guest role, it owns the permissions to start discussions on forums, participate to surveys, read posts and chat with other users. On the other hand, it cannot grade or submit homework like a teacher or a student. This seems to be a suitable role for teacher assistants or lab techs, who often need to interact with students and teacher of a course but that don't need to grade homework or submit assignments. In this case, FCA might have identified an interesting addition to the existing AC model.
Implicit role 2: The UI customization. Implicit role 2 represents the permissions that are shared between the course_creator and non-editing_teacher roles. The permission label consists of two permissions. It is interesting to observe that whereas the *moodle/course:viewhiddencourses* permission is common to both course_creator and non-editing_teacher roles, the *moodle/course:viewhiddencategories* permission is reserved to the course_creator role.
In Moodle, a category defines a group of courses like physics, history or chemistry. Moreover, "hidden" courses or categories are not visible to users who don't own the corresponding *viewhidden* permission. Thus, a course_creator can view all the courses in all the categories while a teacher will only see all the courses in his "visible" categories (e.g. a chemistry teacher will only see chemistry courses). In this case, FCA revealed how the AC model is used to customize the user interface (UI) of the application in function of the user's role. Although we had the intuition that Moodle's AC model was sometimes used in this fashion [60], FCA proved it beyond doubts.
Implicit role 3: The ambiguous. Implicit role 3 represents the intersection of permissions between the authenticated_user and the guest roles. Particularly, we see that while the *moodle/blog:view* permission is shared between both roles, similar *view* permissions, such as *mod/forum:viewdiscussions* and *mod/assignment:view* are specific to the guest role. This choice of implementation may seem questionable. In that context, FCA might have revealed a discrepancy between the AC model as planned by the developers and as implemented in the system.
Implicit role 4: The call for re-factoring. Implicit role 4, the supremum of the lattice is labeled with one permission: *moodle/block:view*. Therefore, this permission is shared by all the roles in the system. This is a serious indication that either this permission is useless and should be tagged for deletion or it was granted to a role that should not own it. Either way, this case should be investigated by developers.

Investigating the role hierarchy

One can observe that FCA naturally reveals the implicit role hierarchy that is created by the role-to-permissions mapping. Indeed, for any two concepts *labeled* with roles r_1 and r_2, $c_1 \preceq c_2 \Rightarrow permissions(r_2) \subseteq permissions(r_1)$. Thus, although Moodle doesn't explicitly support the notion of role hierarchy like hierarchical RBAC models do, FCA revealed how its AC model implicitly implement one.
Investigating Moodle's role hierarchy, we were surprised to observe that:

1. The course_creator and authenticated_user roles do not subsume the guest role.

2. The authenticated_user role is only subsumed by the admin role

Intuitively, we would have thought that the "guest" privileges were shared by all other roles. In a similar manner, it seemed intuitive to us that any user of the system other than "guest" was an "authenticated_user". Without visual support nor clear documentation, these are the kind of facts that an administrator must learn by trial-and-error or code inspection.
Reasoning about the aforementioned observations, we realized that there seems to exist a dichotomy among the roles: support vs. full-fledged roles. A support role, as its name suggests, was not designed to be used on its own. Authenticated_user and course_creator are two such roles. They do not own the permissions of a guest (paths from the infimum to the supremum that go through these roles do not go through the guest role) and thus cannot interact with the application in the most basic way. They, however, possess some "special" permissions that can complement the other, full-fledged, roles; authenticated_users can edit their profile while course_creators, as the name suggests, have the ability to create courses.
Contrary to a support role, a full-fledged role can be assigned to a user without any further modifications. Student, teacher, non-editing_teacher and guest are examples of such roles. They cluster in the center of the lattice and their hierarchy is more complex. Student, non-editing_teacher and teacher roles own all the privileges of a guest and the teacher role owns all the privileges of a non-editing_teacher, as can be expected. Furthermore, we see that teachers and students only share a subset of permissions.
Moodle currently has no mechanism to distinguish support roles from full-fledged roles. Still, it would seem pertinent to enforce the prerequisite that a support role can only be granted to users with a full-fledged role, in order to prevent misconceptions and unexpected behaviors of the application. Interestingly, the RBAC standard defines the *constrained* RBAC model that is specifically designed to enforce such restrictions.
In the case of Moodle, FCA revealed how an apparently trivial AC model can in fact contain lots of unexpected, implicit features such as a role hierarchy and role constraints.

Supporting role addition and redefinition

A recurrent need among users of Web applications is that of customizing the AC model. This will generally results either in modifications of the default role-to-permission relationships or the creation of new roles.
Although legitimate operations, modification and creation of roles may alter the structure of the lattice, introducing new inheritance relations and removing others. We think it may be hard to reason about the underlying AC lattice without formal and visual support. For example, a suggested supplementary role in Moodle is the parent role. According to Moodle's documentation, this role should include three permissions that would allow the parent to view his child's grades as well as read his blog and forum posts.
Adding such a simple role to the existing AC model resulted in three new concepts in the role lattice: one concept for the parent role; the two others representing two new implicit roles (*data not shown*). We already showed how some implicit roles in the default AC model might require developer's attention. In a similar manner, we strongly suggest that any newly introduced role, either implicit or explicit, should be reviewed by the administrator of the application. Interestingly, while implicit roles are very hard to identify by hand, FCA naturally brings them to light.

9.5.2 Analyzing NEC-to-statements mapping

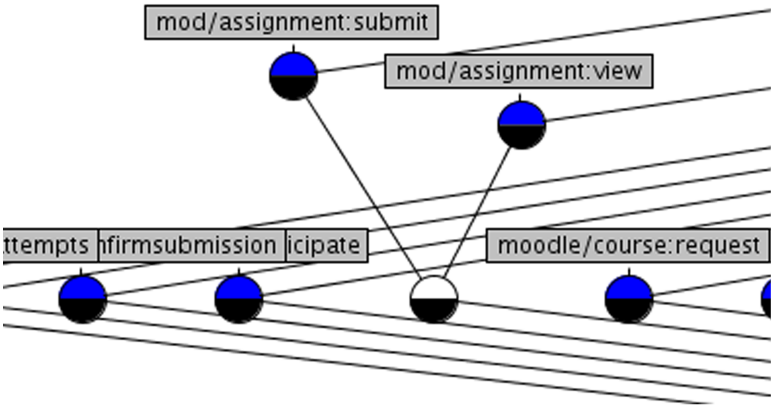


Figure 9.2: Zoomed-in section of the Galois lattice representing Moodle's permission-to-statements mappings.

As mentioned earlier, $NECs$ represent the permissions that are required to execute a statement. Representing the NEC-to-statements mapping of Moodle as a formal context results in a 145 $NECs \times$ 62059 statements matrix. To avoid useless calculations, statements that are

not protected by any NEC do not appear in the formal context. Fig. 9.2 shows a zoomed-in section of the Galois lattice resulting from FCA on that formal context. The complete lattice comprises 166 concepts and is too wide to be presented here.
Circles in this lattice represent formal concepts; pairs (N,S) where each $NEC \in N$ protect all the *statements* $\in S$ and each *statement* $\in S$ is protected by every $NEC \in N$.

Investigation of the permission-to-statements lattice

In the lattice, the vast majority of concepts are strictly independent from one another; any path from the infimum to the supremum typically goes through one concept only. Moreover, every concept in the lattice is labeled with at most one permission. Thus, in Moodle, different $NECs$ usually protect different sets of statements.
Nevertheless, not every concepts are independent from one another. While most concepts (135) represent statements that are protected by a single NEC, 28 concepts represent statements that are protected by 2 $NECs$ and 3 concepts represent statements that are protected by 3 $NECs$.
The zoomed-in section in Figure 9.2 represents a typical section of the lattice: most concepts are mutually independent and aligned in a "flat" manner while some concepts "inherit" $NECs$ from other concepts. Once again, without a priori knowledge of the system nor clear documentation, some NEC inheritance relationships might seem counter-intuitive. Formal concept analysis provides visual support for developers or administrators to observe the implemented AC model "as-is" and judge if the reported concepts are legitimate or not.

Investigation of the Duquenne-Guigues implications

In section 9.4, we mentioned that starting from a formal context, it is possible to derive what is known as a Duquenne-Guigues base of implications. Without going into details, let's mention that this base of implications is minimal in the sense that it contains no redundant implications, holds for the whole formal context and, in the case of the permission-to-statements context, represents implications between $NECs$. In other words, if there exists an implication between $NEC\ A$ and $NEC\ B$ ($A \rightarrow B$), it means that whenever a statement in Moodle is protected by $NEC\ A$, it is also protected by $NEC\ B$.
Table 9.1 shows the implications that were detected on the formal context representing Moodle's NEC-to-statements relationships. Statements that are protected by NEC in the *permission* column are always also protected by the corresponding NEC in the *implication* column.
For example, considering the first implication in Table 9.1, we see that *moodle/grade:export* is

Table 9.1: Duquenne-Guigues implications extracted from Moodle's NEC-to-statements formal context

Permission	Implication
gradeexport/ods:view	moodle/grade:export
gradeexport/txt:view	
gradeexport/xls:view	
gradeexport/xml:view	
gradeimport/csv:view	moodle/grade:import
gradeimport/xml:view	
gradereport/grader:view moodle/grade:edit	moodle/grade:viewall
gradereport/grader:view moodle/grade:manage	
moodle/grade:edit moodle/grade:viewall	gradereport/grader:view
moodle/grade:manage moodle/grade:viewall	
moodle/question:useall	mod/quiz:manage
mod/chat:deletelog	mod/chat:readlog
mod/forum:movediscussions	mod/forum:viewdiscussion
mod/scorm:deleteresponses	mod/scorm:viewreport
mod/forum:startdiscussion	mod/forum:movediscussions mod/forum:viewdiscussion

a pre-requisite to *gradeexport/ods:view*, *gradeexport/txt:view*, *gradeexport/xls:view* and *gradeexport/xml:view*. An administrator ignoring the existence of these implications might end up granting a role with one of the four permissions without granting *moodle/grade:export*. As a result, the role would not behave as *expected*; it would not be able to export grades. Whereas we do not advocate the systematic elimination of implications between $NECs$, we strongly suggest that documenting such implications might help prevent errors and unexpected behaviors of the application.

Another interesting implication can be found in the last row of Table 9.1. It shows that statements that are protected with *mod/forum:startdiscussion* are also protected by *mod/forum:movediscussion* and *mod/forum:viewdiscussion*. Mapping back this information on the role-to-permission lattice, we realized that each of these permissions appears at different levels in the lattice of Figure 9.1. *Mod/forum:viewdiscussion* appears for the first time at the guest concept, *mod/forum:startdiscussion* appears at the "assistant" concept, and *mod/forum:movediscussion* appears at the non-editing_teacher concept.

Furthermore, studying the lattice of Figure 9.1, we see that these three roles are organized

in an "inheritance" chain where non-editing_teacher $\prec$ assistant $\prec$ guest. Thus, permissions(guest) $\subset$ permissions(assistant) $\subset$ permissions(non-editing_teacher), and, with the default configuration, non-editing_teacher, teacher and admin are the only roles that can execute these statements.
Observe however that, because of the role hierarchy, while the statements are protected by three different permissions, only one additional permission is required for an "assistant" to execute them while a guest would need two more permissions. For anyone not familiar with AC models, intuition might dictates that stacking permission checks leads to a "safer" AC model, in the sense that even if one of the permissions is mistakenly granted or stolen through a privilege escalation attack, the remaining permissions still enforce protection. Considering the reported implications, it becomes clear that this intuition is wrong. In this case, the number of privileges that an attacker must gain to execute the protected statements primarily depends on the attacker's granted role.

9.6 Conclusion

In this paper, we presented novel results from applying FCA to Moodle's reverse-engineered AC model. FCA of the extracted role-to-permissions relationship did: 1. reveal a new, potentially useful role (the assistant), 2. show how the AC model is sometimes used for UI customization, 3. draw attention to a faulty permission, shared by all the roles of the application, 4. reveal the implicit role hierarchy, 5. shed light over a dichotomy between support vs. full-fledged roles and 6. illustrate how the addition of a simple role (the parent) can have unexpected impacts over the role lattice.
Furthermore, FCA of the extracted permission-to-statements relationship brought to light some implications between permissions, as implemented in the source code. On one hand, we showed how these undocumented implications might lead to unexpected behavior of the application. On the other hand, we explained how permission implications may induce a false sense of security against misconfigurations and privilege escalation attacks.

Acknowledgements

This research is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program.

CHAPTER 10

CORRECTIONS FOR PAPER 3

The previous chapter contains the published version of the paper entitled: "Investigation of Access Control Models with Formal Concept Analysis: A Case Study". The current chapter addresses comments and corrections by the jury regarding this paper.

In section 9.4, we briefly introduced the Duquenne-Guiges base of implications as a set of rules that holds for a whole formal context and were asked to elaborate. A clearer way to explain what implications are is the following: an implication between a set of attributes A and B, denoted as $A \rightarrow B$, can be read as: if an object has all attributes from A, then it also has all attributes from B. A Duquenne-Guiges basis of implications simply reports the minimal set of implications that exist in a formal context.

In section 9.5.1, we discussed how FCA could reveal and help reason about what we called *implicit* roles. We were asked if we had the chance to pursue our investigations to determine whether these implicit roles have a real impact or are only byproducts of FCA. Since the publication of this paper, we investigated 67 releases of Moodle and monitored the evolution of its access control model. In summary, in Moodle, implicit roles do not become explicit roles over time. However, implicit roles with few privileges (implicit roles 2, 3 and 4 in the paper) are very *unstable*, in the sense that they often merge with other roles and split back to implicit roles from one version to another. In other words, developers should look at implicit roles with few privileges and take actions to eliminate them as these implicit roles seem to be a source of confusion in the long term.

In subsection 9.5.2 we explained that in the NEC to statement formal context, if an implication exists between to sets of NECs A and B, it means that any statement that is protected by all $NEC \in A$ is also protected by all $NEC \in B$. Implications between NECs can be caused by different situations. Nested access checks are one of them. If all access checks for NECs in A are nested in access checks for NECs in B, then there is an implication of the form $A \rightarrow B$. Access checks can also be "inter-procedurally" nested. In other words, if a function that performs access checks for NECs in A is only called from contexts that perform access checks for NECs in B, then there is also an implication of the form $A \rightarrow B$.

In subsection 9.5.2, we also presented a section of a formal concept lattice that was derived from a formal context containing 145 NECs $\times$ 62,059 statements. Some members of the jury expressed some reservations regarding the manual investigation of such a large lattice. We totally agree with the jury that such a lattice is not practical, *as-is*, for end-users. However,

several properties can be calculated in a fully automated manner based on a formal concept lattice. The Duquenne-Guiges basis of implications, that we used in the paper to draw conclusions about interactions between privileges, is only one of the several properties that can be automatically calculated based on a formal concept lattice.

CHAPTER 11

PAPER 4: FAST DETECTION OF ACCESS CONTROL VULNERABILITIES IN PHP APPLICATIONS

ABSTRACT

Access control vulnerabilities in web applications are on the rise. In its 2010 "Top 10 Most Critical Web Applications Security Risks", the OWASP reported that the prevalence of access control vulnerabilities in web applications increased compared to 2007. However, in contrast to SQL injection and cross-site scripting flaws, access control vulnerabilities comparatively received much less attention from the research community.

This paper presents ACMA (Access Control Model Analyzer), a model checking-based tool for the detection of access control vulnerabilities in PHP applications. The core of ACMA uses a lightweight model checker to detect the privileges that are enforced at each statement of an application. Based on this information, ACMA can detect several types of access control vulnerabilities: from forced browsing vulnerabilities to faulty access controls. We show how, when compared to the state of the art, ACMA achieves advantageously comparable results with accelerations up to 890 times faster. Moreover, contrary to the state of the art, ACMA scales up to medium-large applications with large access control models, as shown by the analysis of Moodle, a 400,000+ LOC application counting more than 200 distinct privileges. Results show that ACMA is fast, precise and scalable making it a practical tool for the detection of access control vulnerabilities in real-world applications. A discussion about further extensions to ACMA is also presented.

11.1 Introduction

Every day, millions of people communicate, shop, bank, gather information, and perform numerous tasks using web applications. An increasing number of web applications now deal with private or security sensitive information. Such applications must implement access control mechanisms to protect the privacy of their users. However, the design and implementation of an access control model is far from trivial, especially in web applications. Due to the stateless nature of the HTTP protocol, web applications must verify the credentials of a user for every

incoming requests that access privileged information.

To help alleviate this task, several standards, secure application development guides and application security APIs have been developed and released over the years, often for free. Still, a high proportion of web applications rely on ad-hoc solutions to enforce access controls. Access control models are thus often application-dependent. As a consequence, developing automated detection techniques for access control vulnerabilities is harder than for language-dependent vulnerabilities such as SQL injection (SQLi) and cross-site scripting (XSS).

While SQLi and XSS vulnerabilities still are the most prevalent flaws in web applications, the OWASP raised the "Failure to Restrict URL access" vulnerability from the 10th position (in 2007) to the 8th position (in 2010) of their "Top 10 Most Critical Web Application Security Risks" [129]. In 2010, they also added the "Unvalidated Redirects and Forwards" to their list. Both types of vulnerabilities stem from faulty access controls.

One of the main challenges for the detection of access control vulnerabilities lies in the identification of pages that *should* have a restricted access. Recently, some researchers [156, 152] observed that privileged pages are rarely left entirely unprotected. They argue that access control vulnerabilities usually occur because of "hidden" execution paths that lack access control checks rather than because of a complete absence of access control.

For example, web applications will typically hide links to privileged pages from unprivileged users. While this is a good practice, it is not sufficient to ensure security, as a malicious user can guess the URL of the hidden page and access the privileged information through this "hidden" path. In the context of this paper, we refer to this kind of access control vulnerabilities as "forced browsing" vulnerabilities. Forced browsing vulnerabilities are a subset of "hidden" paths vulnerabilities in access control models. The term was introduced by Sun et al. [156] as they were the first to tackle the automatic identification of such security flaws.

In this paper, we present ACMA (Access Control Models Analyzer) a novel approach for the detection of access control vulnerabilities in PHP web applications. Contrary to the approach presented in [156], ACMA can detect many types of access control vulnerabilities, not just forced browsing ones. Moreover, ACMA handles larger applications than previously published tools [156, 152] and identifies forced browsing vulnerability faster (up to 890 times faster) with positively comparable false positive and precision rates.

The main contributions of this paper are:

- The definition and implementation of a fast and precise approach, anchored in inter-procedural analysis and model checking theory, for the automatic detection of access control vulnerabilities.

- The comparison of ACMA with the state of the art. We ran our tool on all the applications previously investigated in [156]. Results show that ACMA is faster, more scalable and requires less manual efforts without sacrificing precision.
- A discussion of the additional types of access control vulnerabilities that are found in web applications and how ACMA can be used to report them.

The rest of this paper is organized as follows. In section 11.2, we define the concepts that are needed to model access control vulnerabilities. In section 11.3, we present ACMA's approach and detail its algorithms. Section 11.4 shows the running time and precision of ACMA and reports the vulnerabilities that were identified. In section 11.5 we present some application specific results and compare them to those in [156]. Section 11.6 emphasizes the speed and scalability of ACMA by investigating Moodle, a medium-large application with a large access control model. This section also proposes a classification of access control vulnerabilities and discusses how ACMA can help identify and report them. Finally, section 11.7 discusses related work and section 11.8 concludes the paper.

11.2 Definitions

This section presents the terms and definitions that will be used throughout this paper.
1. Privilege: A privilege grants the permission to perform certain actions on certain objects. For example, in a typical blog engine, the "post_comment" privilege allows a user to post (action) a comment (object).
2. Role: A role is simply a label for a group of privileges. In large systems with lots of privileges, it is often more practical to grant users a pre-defined set of privileges, called a role, instead of granting users each privilege separately. In small systems, roles often act as privileges.
3. Access control routine: An access control routine is a mechanism by which an application verifies that a user owns a given privilege or belongs to a given role.
4. Privileged user: In the context of this paper, a privileged user owns a specific privilege or belongs to a specific role. Access control routines grant access to privileged users.
5. *P*-protected links: A link is said to be p-protected if it is only displayed to users that own the privilege p.
6. *P*-privileged page: A page is p-privileged if it is only pointed to by p-protected links.
7. *P*-guarded page: A page is p-guarded if it is only accessible by users who own the privilege p. Note that p-privileged pages are not necessarily p-guarded. Typically, unprivileged users trying to access a guarded page will either be redirected to another page or presented with an error message before any sensitive content is displayed.

8. Normal browsing: Normal browsing defines the navigation of a user that only uses the intended entry points of an application and that only follows the links that are presented to him.

9. Forced browsing: We define forced browsing as the action of directly navigating to a page through its URL.

10. Forced browsing vulnerability: A forced browsing vulnerability exists if a p-privileged page is not p-guarded and thus reachable through forced browsing. Often, this kind of vulnerability occurs when developers try to "hide" a page by only displaying p-protected links to that page. In these cases, a malicious unprivileged user might be able to perform a privilege escalation attack by correctly guessing the URL of the "hidden" page.

11.3 The ACMA tool

This section aims to provide the reader with a detailed description of our methodology. ACMA builds on work previously published in [60]. For the purpose of this paper, we implemented several extensions to identify forced browsing vulnerabilities. Specifically, ACMA: 1. extracts the p-privileged pages of an application 2. identifies the p-guarded pages 3. verifies that the p-privileged pages are p-guarded and 4. reports the vulnerabilities.

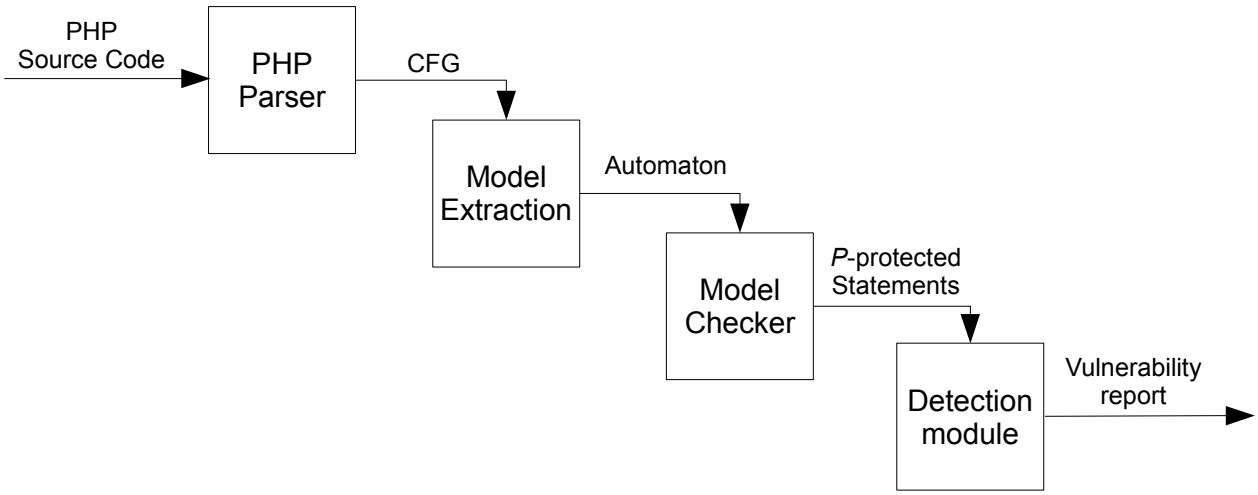


Figure 11.1: ACMA architecture

ACMA's architecture is illustrated in Figure 11.1. The process starts with PHP source code and finishes with the production of a vulnerability report.

11.3.1 Control-flow graph extraction

Our experimental setup uses a PHP parser generated by JavaCC, a common parser generator tool. The PHP grammar used with JavaCC is a modified version of [144]
The parser outputs a program Control Flow Graph (CFG):

$$CFG = (V_{CFG},\ E_{CFG}) \tag{11.1}$$

with multiple *entry* nodes $v_{in} \in V_{CFG}$ and a multiple *exit* nodes $v_{out} \in V_{CFG}$. Indeed, each PHP script of an application has its own *entry* and *exit* nodes. Nodes in V_{CFG} can be of type *generic*, *entry*, *exit*, *call_begin*, *call_end*, $function\_begin$ and $function\_end$. Nodes of type *generic* are involved in intra-procedural control flow; nodes of type *call_begin*, *call_end*, *entry*, *exit*, $function\_begin$ and $function\_end$ are used in inter-procedural control flow.
Edges in E_{CFG} can be of type *generic*, $grant\_p_i$, $revoke\_p_i$, *call*, or *return*. Edges of type *generic* represent intra-procedural transfers of control that do not affect the granting of privileges. Edges of type $grant\_p_i$ represent intra-procedural transfers of control that grant the privilege p_i. Likewise, $revoke\_p_i$ edges revoke the privilege p_i. Finally, edges of type *call* and *return* represent inter-procedural control flow links. Edges of type *call* link *call_begin* to $function\_begin$ nodes while edges of type *return* link $function\_end$ to *call_end* nodes.
Identifying access control routines involves finding some syntactic patterns in the Abstract Syntax Tree (AST). While these patterns are application-dependent, we observed that they usually are very similar in their syntactic structures. In some applications, a function receives a string argument, representing the privilege, and return a Boolean value asserting whether or not the user owns the privilege. In other systems, an explicit comparison is performed between a session variable, representing the privilege, and the required value of the variable. The application-specific access control routines are presented in section 11.5.
The access control routines that are identified at the AST level produce security edges in the CFG. For example, suppose an access check in the form of a predicate inside an `if` statement. If the predicate evaluates to true, the control is transferred to a block in which the privilege p_i is granted until block end. Conversely, if the predicate evaluates to false, the control is transferred to another block in which the privilege p_i is revoked. Those transfers of control are represented in the CFG with $grant\_p_i$ and $revoke\_p_i$ edges.

11.3.2 Model extraction and inter-procedural aspects

The model extraction uses the CFG produced by the PHP parser and transforms it into an automaton A suitable for model checking:

$$A \ = \ (Q_A, \ L_A, \ T_A, \ q_0, \ V_A, \ G_A, \ A_A) \tag{11.2}$$

where Q_A is a finite set of states; L_A is a finite set of labels applied on the states; $T_A \subseteq Q_A \times Q_A$ is a set of transitions; q_0 is the initial state; V_A is a set of variables used in "guards" and "assignments"; G_A is a set of "guards" that are logical propositions over V_A and are associated with transitions; and A_A is a set of assignments that modify the value of variables and are also associated with transitions.

The model extraction is performed by operations that include the rewriting of intra-procedural and inter-procedural nodes, and the identification of privilege granting functions through access control patterns. The intra-procedural nodes V_{CFG} and edges E_{CFG} are directly rewritten in the automaton A into the corresponding states Q_A and transitions T_A. A label x is associated to each state to indicate to which statement the state corresponds to.

Conceptually, we model the granting and revoking of privileges as the activation and deactivation of Boolean properties in the model checking automaton. To model these Boolean properties in the automaton, each state is duplicated in two versions $s_{x,0}$ and $s_{x,1}$, where $s_{x,0}$ represents that the property is false (deactivated) and $s_{x,1}$ represents that the property is true (activated). During the model extraction phase, *grant* edges, that represent the granting of a privilege, become transitions to $s_{x,1}$ states. Conversely, *revoke* edges, that represent the revoking of a privilege, become transitions to $s_{x,0}$ states.

From the inter-procedural perspective, ACMA performs a *security*-sensitive analysis. Unlike context-sensitive analysis, that distinguishes every calling context, ACMA only distinguishes calling contexts with different security levels. In the context of this paper, ACMA deals with Boolean privileges. The possible security contexts are thus limited to *true* and $false$, depending if the privilege was previously granted or revoked.

At the automaton level, ACMA treats the security contexts in a similar manner to privileges: each state in the automaton is duplicated in two versions, representing the two possible security contexts. At the end of the model extraction phase, the number of states in the automaton equals $4 \times |V_{CFG}|$. Indeed, to each vertex $v_x \in V_{CFG}$ corresponds four states $s_{x,0,0}$, $s_{x,0,1}$, $s_{x,1,0}$, $s_{x,1,1}$, representing every combination of security context and privilege activation values. Variables V_A, guards G_A and assignments A_A are used to reproduce the inter-procedural logic and propagate security contexts through inter-procedural call and return transitions, as detailed in [100].

Performing a security-sensitive analysis considerably reduces computation time at the expense of path-sensitivity. On the one hand, ACMA identifies all the privileges that *may* [66] be activated at each statement. On the other hand, since ACMA merges calling contexts with identical security levels, it cannot, for example, retrieve the execution path that misses an access control check. However, in the context of access controls vulnerabilities, we argue that retrieving the execution path that exposed a vulnerability is not as crucial as detecting the flaw. Indeed, information about the faulty execution path is usually not required for the repair of access control vulnerabilities. These flaws are generally repaired by the addition of an appropriate access control check.

11.3.3 Model checking of security properties

Software model checking [33] is the algorithmic analysis of programs to prove properties of their executions. While originating from logic and theorem proving fields, it has now evolved as a hybrid technique, simultaneously making use of analysis traditionally classified as theorem proving, model checking, or dataflow analysis [87].

A well-known limitation of model checking techniques is known as the combinatorial "state space explosion problem", where the model checker has to explore a combinatorial number of states in the system under study. Several papers proposed exploration strategies, heuristics and specialized data structures to circumvent this problem and analyze increasingly large systems.

The approach we adopted in ACMA differs from these strategies. Instead of solving "generic" model checking problems, ACMA uses a specialized model checker for the verification of Boolean control-flow properties. As a consequence, ACMA is able to verify Boolean privileges in a fast and precise way, making it suitable for the analysis of real-world applications.

Since the Boolean privileges and security contexts are directly encoded in the states of the automaton, ACMA can solve the security-sensitive inter-procedural propagation of privileges by computing state reachability over the automaton. Moreover, since the number of states in the automaton equals $4 \times |V_{CFG}|$, the reachability analysis in ACMA scales linearly with the size of the application. For more details about the state reachability algorithm, see [100].

A statement $stmt_j$ is defined as p-protected by the privilege p_i if the following formula is verified:

$$\Diamond stmt_j \wedge \Box(\ stmt_j \wedge p_i) \tag{11.3}$$

meaning that the statement $stmt_j$ is eventually reached ($\Diamond stmt_j$) and always when $stmt_j$ is reachable, the privilege p_i is granted ($\Box(\ stmt_j \wedge p_i)$). States satisfying Equation 11.3 correspond to statements that are reachable by an execution for which the privilege p_i is

granted, but that cannot be reached by an execution for which p_i was not granted.
The model checker in Figure 11.1 solves the reachability problem of states in the automaton and therefore solves the corresponding problem of identifying p-protected statements. To prevent combinatorial state space explosion, ACMA produces many automata, one per privilege, and analyzes them separately.
In summary, the model checker computes *may* information, that is true of all program executions [66], and reports the p-protected statements, that are guaranteed to be protected by the privilege p on *every* execution paths.

11.3.4 Privileged pages extraction

Our model checker reports p-protected statements for each privilege of an application, as presented in Equation 11.3. In section 11.2, we defined a page as p-privileged iff all the links pointing to it are p-protected.
Using the results from our model checker, we designed an algorithm, presented in Algorithm 11.1, to extract p-privileged pages from an application. The simplified form of the regular expressions used in the algorithm are the following:

link_re = `<a\b[^>]*href=(.*?)[\s>]`
form_re = `<form\b[^>]*action=(.*?)[\s>]`
frame_re = `<i?frame\b[^>]*src=(.*?)[\s>]`

These regular expressions are presented as a reference. ACMA uses a mix of regular expressions and post-processing routines to deal with malformed HTML and to extract a maximum number of links. In the algorithm, regular expressions are matched against statements using the *match* function (see line 5).
The algorithm presented in Algorithm 11.1 can be summarized into three steps:

1. Extract the targets of p-protected links.
2. Extract the targets of unprotected links.
3. Report the targets that are only pointed to by p-protected links for each privilege $p \in P$. Those are the p-privileged pages.

In its current version, the algorithm reports privileged pages on a per-privilege basis. In section 11.2, we defined a role as a label for a collection of privileges. For some applications, it might be more significant to report privileged pages on a per-role basis.

```
1:  P = {privileges}
2:  protected_pages = ∅
3:  unprotected_pages = ∅
4:  for all stmt ∈ {statements} do
5:    if link_re.match(stmt) || form_re.match(stmt) || frame_re.match(stmt) then
6:      link = stmt
7:      for all p ∈ P do
8:        if link ∈ {p-protected statements} then
9:          protected_pages[p] ∪= link.target
10:       else
11:         unprotected_pages[p] ∪= link.target
12:       end if
13:     end for
14:   end if
15: end for
16: privileged_pages = ∅
17: for all p ∈ P do
18:   privileged_pages[p] = protected_pages[p] \ unprotected_pages[p]
19: end for
20: return privileged_pages
```

Algorithm 11.1: Privileged pages extraction algorithm

From an algorithmic point of view, adding support for per-role privileged pages is simple. It is sufficient to replace the set of privileges at line 7 in Algorithm 11.1 by a set of privilege tuples $R = \{(p_1, \ldots, p_m), \ldots, (p_1, \ldots, p_n)\}$, where each tuple $r \in R$ represents a role. Then, at line 8, instead of verifying if the link is p-protected, it would be sufficient to verify if the link is r-protected, where:

$$r\text{-protected links} = \{p\text{-protected links} \mid p \in r\}.$$

11.3.5 Forced browsing analysis

Forced browsing analysis is performed by considering each PHP script or HTML file of an application as an entry point of the application. Indeed, when a user enters a URL that represents the path to an application's file, the server will try to execute the PHP script or render the HTML file. Note that in practice, a lot of forced browsing requests are simply filtered out by the web server or result in an error page. However, for the sake of circumspection, we consider each file of a Web application as a potential entry point.
First, the PHP parser identifies the entry point of each file at the AST level and converts them into *entry* nodes in the CFG. During the model extraction phase, each *entry* node becomes an *entry* state in the automaton. Forced browsing is then simulated during the model checking

phase by computing state reachability from each entry state in the automaton.

11.3.6 Detection of vulnerabilities

A forced browsing vulnerability exists if a page that *should* be guarded is accessible by unprivileged users through direct access to the URL of the page. However, identifying the pages that ought to be privileged is not a trivial task When the size of the application is small, asking the developers or some knowledgeable users to supply security specifications is a possible solution. However, as the number of pages increases, it might become difficult to find people with sufficient knowledge to provide specifications for the whole application.
ACMA infers the pages that should be guarded by comparing the pages that are reachable through "normal" browsing versus the pages that are reachable through "forced" browsing. On the one hand, p-privileged pages represents the pages that are only reachable by p-privileged users under normal browsing. On the other hand, p-guarded pages represent the pages that are only reachable by p-privileged users, even under forced browsing. ACMA makes the assumption that p-privileged pages should also be p-guarded.
As mentioned in section 11.2, a guarded page will usually quickly redirect unprivileged users to another page or display an error message. At the source code level, this behavior is usually implemented with an early access check coupled with a redirection in case of failure. According to this assumption, the last statement of a guarded page is only reachable by privileged users. Our approach thus reports a page as p-guarded when the last reachable statement of the corresponding PHP script is p-protected.
ACMA reports a vulnerability whenever it identifies a p-privileged page that is not p-guarded.

11.4 Experimental results

This section presents the performances of ACMA for access control vulnerabilities detection. We first show how ACMA outperforms the previous tool both in terms of speed and scalability with positively comparable precision and false positive rates. Second, we show how ACMA is the first tool of its kind to handle medium-large applications by investigating Moodle, a PHP application counting more than 400,000 LOC. All the analyzes were computed on a PC with a dual-core CPU (3.0GHz) and 4GB of RAM.
Table 11.1 shows characteristics about the investigated applications. The first seven applications were also studied in [156] and serve as a benchmark for ACMA while the last line shows statistics about Moodle, a newly investigated application. The version numbers are listed together with the application name, when available. The reported numbers of lines of code exclude blank lines and comments and where calculated with the cloc.pl software [43].

Table 11.1: Characteristics of the evaluated applications

Application	Files	LOC	
		PHP	HTML
SCARF	25	1,318	0
Events Lister 2.03	37	2,076	544
PHP Calendars	67	1,350	0
PHPoll 0.97	93	2,571	0
PHP iCalendar 1.1	183	8,276	0
AWCM 2.1	668	12,942	5,106
YaPiG 0.95	134	4,801	1,271
Moodle 1.9.5	**5124**	**404,399**	**30,547**

Table 11.2 reports the vulnerability analysis results for all the investigated applications. An important distinction must be made between "privileged" pages and "guarded" pages. As mentioned in section 11.2, ACMA reports a page as privileged if it is only pointed to by privileged links while a page is reported as guarded if it prevents unprivileged access with an explicit access control check.

In Table 11.2, column "Priv." shows the number of privileged pages that were identified in each application. Column "Vuln." reports the number of pages presenting a forced browsing vulnerability, while columns "FP" and "Guarded" show the number of false positive and the number of guarded pages respectively.

Manual verification revealed that ACMA identified all the known forced browsing vulnerabilities in the investigated applications. The vulnerabilities reported by ACMA are consistent with the results reported in [156]. Furthermore, the two approaches are comparable in term of false positive rate.

Table 11.3 shows the precision rates of our link extraction algorithm for all the applications.

Table 11.2: Forced browsing analysis results.

Application	Priv.	Vuln.	FP	Guarded
SCARF	4	1	0	3
Events Lister 2.03	3	2	0	10
PHP Calendars	1	1	0	2
PHPoll 0.97	3	3	0	0
PHP iCalendar 1.1	0	0	0	1
AWCM 2.1	44	1	1	48
YaPiG 0.95	3	0	0	6
Moodle 1.9.5	**43**	**0**	**0**	**161**

Table 11.3: Precision rates and causes of failure of the link extraction algorithm.

Application	Cause of failure	Precision
SCARF	–	100%
Events Lister	Erroneous links (2)	100%
PHP Calendars	–	100%
PHPoll	`${_POST[prepoll]}`	97%
	`${percorso_link}`	
PHP iCalendar	`${BASE}`	94%
	`${minical_view}`	
AWCM	Erroneous links (2)	100%
YaPiG	–	100%
Moodle	Unresolved paths (13)	98%
	`${securewwwroot}`	
	`${wwwroot}`	
	`CALENDAR_URL`	
	`${path}`	

The "cause of failure" column reveals three different causes that prevent our algorithm from resolving a link:

1. Erroneous links. These links are *correctly* processed by our algorithm, but point to *erroneous* pages in the application. In other words, a user clicking on of these links would get an "Error 404 – page not found" error. Erroneous links were not considered to decrease the precision of our algorithm.

2. Variables and constants. Some links are dynamically generated using variables. Table 11.3 shows the variables that prevented the resolution of some links by our algorithm. The number of variables involved in the dynamic generation of links is very small; it varies between 0 and 4 in the investigated applications.

3. Unresolved path. Some PHP scripts contain functions that generate HTML links with "invalid" relative paths. In practice, PHP resolves the relative path of a link based on the path of the caller script. Consequently, these "invalid" relative paths may become valid at runtime, depending on the caller script.

Column "Precision" shows that, despite some unprocessed links, ACMA still extracts and resolves links with a precision rate of 94% to 100% for the investigated applications.
Up to now, we showed how ACMA detected all the known forced browsing vulnerabilities and how its link extraction algorithm is able to process the vast majority of links in the investigated applications. However, the greatest strengths of our approach are speed and scalability.

Table 11.4 shows the execution times of our approach with and without I/O. The execution times without I/O are indicated in parenthesis. ACMA is currently designed for flexibility at the expense of a lot of I/O operations. Execution times without I/O represents the actual computation time of ACMA.

The "Acceleration" column illustrates how much ACMA outperforms the approach in [156] in term of execution time. For PHP iCalendar, we see that ACMA finds the forced browsing vulnerabilities about 890 times faster. Accelerations were estimated based on the execution times (without I/O) reported in [156]. Although the two experimental setups slightly differ (quad-core CPU (2.40GHz) vs. dual-core CPU (3.0GHz), both with 4 GB of RAM), we strongly argue that the reported accelerations provide fair estimates of the practical accelerations.

Investigating Table 11.4, we see that the highest accelerations were obtained for YaPiG (223 times faster) and PHP iCalendar (890 times faster). These two applications count 4,801 and 8,276 LOC respectively. ACMA ran about 50 times faster on AWCM which counts 12,942 LOC.

On the one hand, ACMA scales linearly and has predictable performances. On the other hand, the tool in [156] presents highly variable performances that seem application-dependent.

11.4.1 Current limitations and future work

Identification of access control routines

Access control routines are application dependent. It its current version, ACMA requires the manual specification of the access control routines of an application. Some work by Son et al. [152] addresses the problem of inferring the access control checks of an application. However, their approach is heuristic and requires more than 6 hours to analyze a 13,862 LOC application. We would be very surprised if it scaled to applications of the size of Moodle.

On the other hand, access control routines are usually easy to identify and syntactically

Table 11.4: Analysis times

Application	Time (s)					
	Parsing	Model checking	Vulnerability detection	Total time with I/O	Total time without I/O	Acceleration (× faster)
SCARF	1.393 (0.097)	2.930 (0.143)	0.259	4.582	0.499	12.06
PHP Calendars	1.147 (0.106)	2.528 (0.128)	0.204	3.879	0.438	11.62
Events Lister 1.03	1.102 (0.084)	2.830 (0.156)	0.193	4.125	0.433	8.87
PHPoll 0.97	1.100 (0.120)	2.796 (0.165)	0.224	4.120	0.509	8.37
YaPiG 0.95	2.09 (0.188)	7.847 (0.355)	0.390	10.327	0.933	223.34
PHP iCalendar 1.1	2.426 (0.246)	7.329 (0.321)	0.287	10.042	0.854	890.66
AWCM 2.1	2.913 (0.252)	6.509 (0.452)	1.086	10.508	1.79	49.99
Moodle 1.9.5	**21.555 (3.046)**	**435.612 (4.921)**	**8.330**	**465.497**	**16.172**	–

similar from one application to the other. Indeed, while automatic approaches must consider all variable comparisons and function calls as potential access control routines, humans are very efficient in locating security-related variables and functions (e.g. the `has_capability` function in Moodle or the `admin` variable in PHP Calendars). Consequently, the amount of manual work required to analyze a new application with ACMA is fairly limited.
Specifically, ACMA currently identifies access control routines in the AST, with the help of visitors [55]. The amount of manual work required to analyze new applications with ACMA thus essentially boils down to the creation of application-specific visitors. To give the reader an idea, the application-specific visitors used in this study are about 13 lines of "original" code long and very easy to implement.
Compared to ACMA, the approach in [156] not only requires the manual specification of access control routines, it also requires the manual identification of the application's entry points as well as the manual specification of "critical" function return, session, cookie, request parameter, database record and variable values. Moreover, no indication is provided as to how to identify those "critical" values. From our understanding, it seems that one must have a thorough knowledge of the application in order to provide such information. Thus, ACMA does not only requires less manual work, it also considerably mitigates the need for a thorough comprehension of the software under study.

Link extraction algorithm

Our approach for link extraction is based on the matching of regular expressions against the source code. Consequently, some links, that use variables and constants for example, cannot be processed by our approach. However, Table 11.3 shows that this is a minor limitation as ACMA currently processes 94% to 100% of the links in the investigated applications. In a further release, we plan to extend ACMA with the inter-procedural pattern propagation algorithm presented in [61] to enhance the resolution rate of dynamically generated links.

11.5 Analysis of benchmark applications

11.5.1 SCARF

SCARF stands for the Stanford Conference And Research Forum. Its access controls are implemented with the `is_admin` and `require_admin` functions. Our tool detected a previously reported vulnerability (CVE-2006-5909) concerning the generaloption.php page. This page is only accessible through privileged links but is not guarded.
ACMA succeeded in processing all the links of the application.

11.5.2 Events Lister

As its name suggests, Events Lister allows users to display and manage their events. The current version is 2.04 and we had to use forced browsing to download the 2.03 version containing the security flaw! Access checks are performed by calling the `checkUser` function. Our tool detected previously reported forced browsing vulnerabilities for admin/user_add.php and admin_setup.php pages.
While false positives were previously reported for this application, ACMA did not report any. Another interesting consideration is the number of reported guarded pages. ACMA reported 10 guarded pages while only 5 guarded pages were reported in [156].
On the one hand, in [156], the authors claim that their approach reports a page as guarded when: "the context-free grammar sizes of the two roles are different because of the different HTML outputs that are presented.". On the other hand, the `checkUser` function redirects the user to the login page if the access check fails. Consequently, any script that invokes the `checkUser` function renders HTML that differs in function of the user's role. Since Events Lister counts 10 scripts that invoke the `checkUser` function, their approach should have reported 10 guarded pages. We therefore conclude that their approach erroneously reported 5 pages as unguarded. ACMA correctly reported all guarded pages.
From the link extraction perspective, ACMA processed all the valid links. The only two links ACMA did not process were erroneous. One link points to a file that does not exists and would thus generate an "page not found" error. The other link is literally: `<a href=events.php></a>`. There is no anchor text and the link is therefore unusable!

11.5.3 PHP Calendars

PHP Calendars is one of the many calendar management systems available online. Access checks are performed through the comparison `$_SESSION["admin"] == "yes"` in the admin/access.php page. Our tool detected the known vulnerability (CVE-2010-0380) concerning the install.php file. Like in [156], we had to manually add the install.php page to the set of privileged pages in order to detect the vulnerability.
Interestingly, while ACMA correctly reported the two guarded pages: admin/import.php and admin/index.php, the tool in [156] not only failed to report admin/index.php as guarded, it also erroneously reported powerfeed.php as guarded. Since powerfeed.php is a library of functions and displays no HTML by itself it should not have been reported as guarded. Contrariwise, the admin/index.php generates HTML, has an access check and should have been reported as guarded.
ACMA did process all the links of this application.

11.5.4 PHPoll

PHPoll is an online poll application where access checks are performed by verifying that the user provided correct values for `$_COOKIE[$string_cook_login]` and `$_COOKIE[$string_cook_password]`. If the verification succeeds, the application sets the `$test_log` variable to true. Otherwise, it is set to false. ACMA detected all known vulnerabilities. The three privileged pages: modifica_band.php, modifica_configurazione.php and modifica_votanti.php are all vulnerable to forced browsing attacks.
Only two links were unprocessed in this application for an overall link extraction precision rate of 97%.

11.5.5 PHP iCalendar

PHP iCalendar, like PHP Calendars, is a calendar management system. Access checks are implemented by verifying the value of the `HTTP_SESSION_VARS ["phpical_loggedin"]` variable. Our tool correctly identified the admin.php page as guarded but did not report it as a privileged page since no link points to that page; it is an entry point of the application.
Some links were unprocessed due to the use of variables for dynamic link generation. ACMA did process 94% of the links in this application.

11.5.6 AWCM

AWCM stands for Arabic Web Content Manager. Access checks are performed through the comparison `$_SESSION["awcm_cp"] == "yes"` in the control/common.php page. ACMA correctly identified a previously reported vulnerability (CVE-2010-1066) in the control/db_backup.php file.
ACMA reported one false positive for this application. The control/logout.php file is privileged but not guarded and thus reported as vulnerable to forced browsing attacks. However, since the purpose of that page is simply to log out the current user, it does not qualify as a security vulnerability.
The only two links that were unprocessed by ACMA are erroneous. One of them points to a page that does not exist. The other one points to `news_show..php`. The double dots are intentional. This is obviously a typo. ACMA thus achieved a precision rate of 100% for link extraction.

11.5.7 YaPiG

YaPiG stands for Yet Another PHP Image Gallery and implements access checks through the `check_admin_login` function. ACMA did not identify any forced browsing vulnerabilities in

this application.
Our link extraction algorithm succeeded in processing all the links of the application.

11.6 Analysis of Moodle

Through the investigation of the previous applications, we showed how ACMA identifies forced browsing vulnerabilities in a precise, fast and scalable way. We also demonstrated how ACMA outperforms the existing forced browsing vulnerability identification tool with accelerations up to 890 times faster.
ACMA, however, can identify several types of access control vulnerabilities, not just forced browsing ones. ACMA also scales to medium-large applications. In this section, we investigate Moodle, a 400,000+ LOC application with a large access control model that presents different types of access control vulnerabilities.
Moodle is a course management system written in PHP. The Moodle website reports a total of 57,111,699 users across 219 countries [119]. Moodle implements access controls with two functions: `has_capability` and `require_capability`. The `has_capability` function returns a Boolean value indicating whether or not the user owns the privilege that is passed as a parameter. The `require_capability` function stops the execution if the user does not own the privilege.
Unlike the other investigated applications, that only had one privilege (admin), Moodle has a more elaborate access control model. The investigated version of Moodle counts more than 200 distinct privileges. With the analysis Moodle, we show how ACMA not only scales to larger application but also to larger access control models.
In order to provide a fair comparison with the other applications, in Table 11.4, we reported the average model checking and vulnerability analysis times for one privilege. Our tool requires around 40 minutes to perform the model checking phase on all of Moodle's privileges and around 6 minutes to perform the vulnerability analysis. The parsing step only needs to be executed once, no matter the number of privileges.
ACMA did process 98% of the links in Moodle. Of the 18 unprocessed links, 5 are dynamically generated with variables. The remaining 13 links were unprocessed due to unresolved relative paths.

11.6.1 Beyond forced browsing vulnerabilities

While previous applications only presented forced browsing vulnerabilities, they are not the only kind of access control vulnerabilities. A manual review of the reported access control vulnerabilities in Moodle and other PHP applications revealed that access control vulnerabilities

can be classified in three categories:

1. The undetected execution path. An access control is enforced to protect some part of the application but an unexpected execution path allows unprivileged users to bypass the access control routine. Forced browsing vulnerabilities fall into this category. None were detected in Moodle.

2. The faulty control. The access control routine is correctly implemented but does not check the proper privilege according to the logic or the documentation of the application. Moodle presents vulnerabilities of this type, as described below.

3. The missing access control. A part of the application that *should* be protected by an access control check is not. Moodle also has vulnerabilities of this type, as described below.

The approach proposed in this paper is geared towards the detection of forced browsing vulnerabilities. By default, ACMA detects *web pages* that are improperly protected due to undetected execution paths. However, the core of ACMA is based on a process that has a much finer granularity. The model checker identifies the protected *statements* of an application. This makes our tool suitable for the detection of finer, more subtle access control vulnerabilities.

Access control vulnerability detection approaches that abstract web applications to files or web pages lack the necessary precision for the identification of several types of vulnerabilities. Indeed, none of the reported access control vulnerabilities in Moodle concern a page; they are always related to specific statements of a PHP script.

For example, a permission escalation vulnerability (CVE-2010-1616) was identified in the file `backup/restorelib.php`. This file is a library of functions. It does not display HTML code and is thus not vulnerable to forced browsing attacks. However, one of its function performs privileged operations without proper privilege verification. This vulnerability is of the "missing access control" type. In order to identify such a vulnerability, we augmented ACMA with supplementary rules to report unprivileged database access statements.

Another access control vulnerability (CVE-2010-1617) was found in the `user/view.php` file. In that case, a privileged statement is protected by the *wrong* privilege. In fact, contrary to guarded pages that present an error message or redirect unprivileged users, this page customizes the displayed HTML in function of the user's privileges. The page either displays the name of all the registered students of a course (privileged version) or only the name of the current user (unprivileged version).

A single statement differs between the privileged and the unprivileged version of the script. ACMA correctly reports the (wrong) privilege that currently protects the privileged statement. Only ACMA has the sufficient granularity to identify such statement specific vulnerabilities. This vulnerability is of the "faulty access control" type.

Similarly, the `mod/glossary/showentry.php` file also has a "faulty access control" vulnerability (CVE-2009-4299). Originally, the script checked if the user had the `course:viewhiddenactivities` privilege and redirected him to another page in case of failure. In the corrected version, the `glossary:approve` privilege is verified instead. In the corrected version, if the check fails, privileged information is simply not displayed to the user.

11.7 Related work

11.7.1 Taint analysis

In some respects, access control vulnerabilities detection approaches relates to taint analysis when they are applied to access control models that contain only one Boolean privilege (e.g. admin/user models).
Generally speaking, one refers to *tainted* variables do designate untrusted data that can flow to security-sensitive code. Untrusted data are therefore tagged as *tainted* until some sanitizing routine sanitizes them. Our approach implements a form of control flow taint analysis where access control routines are analogous to sanitizing routines that sanitize *tainted* executions. However, in contrast with classic taint analyzes, in which no distinction is made between sanitizing routines, ACMA tags executions with precise sanitation information, in the form of multiple and independent privileges.
Tripp et al. [163] implemented a static taint analysis for Java (TAJ). Their approach aims at identifying security vulnerabilities in Web applications. The authors share our willingness to provide a scalable approach, suited for industrial size applications. Their approach mainly differs from ours from the language (Java vs. PHP) point of view.
Clause et al. [34] presented a framework (DYTAN) for conservative dynamic tainting. Their framework takes a user supplied configuration file, describing the taint analysis to be performed, and instruments x86 binaries accordingly. To our knowledge, they are among the first authors to explicitly address the problem of control flow tainting.
Jovanovic et al. [88, 89] developed a static taint analysis for identification of cross-site scripting (XSS) vulnerabilities in PHP Web applications. They used an inter-procedural and context-sensitive data flow analysis enhanced with literal analysis to detect potential XSS vulnerabilities in PHP scripts.

11.7.2 Vulnerability detection in web applications

Finding SQLi and XSS vulnerabilities. SQLi and XSS are data-flow vulnerabilities. They occur when untrusted inputs from users can reach *sensitive* parts of a web application

without proper sanitation. Several research papers have been devoted to the subject [105, 173, 83, 72, 92]. However, data-flow approaches are not well suited for the detection of access control flaws, which are control-flow vulnerabilities.

Other researchers [168, 169] used string analysis to detect SQLi and XSS vulnerabilities. Their approach is an extension of the string analyzer that was originally published by Minamide [116]. String analysis is a powerful and interesting approach, but it does not seem to scale well to large applications. In their previous work, Wassermann et al. reported that their tool failed to analyze Drupal, a 43,000 LOC PHP application. This raises questions about the scalability of their approach. In this paper, we showed how ACMA handles Moodle, a 400,000+ LOC PHP application.

Access control vulnerabilities as workflow violations. In [73], Hallé et al. proposed to model web applications as state machine and dynamically verify that runtime operations don't violate some pre-defined temporal properties. Similarly, Dalton et al. proposed a tool, called Nemesis [41] that takes a user specified access control model as input and dynamically detects access control violations at runtime. These approaches suffer from the excessive amount of manual work that is required by developers and from the runtime overhead introduced by dynamic verifications of security properties.

Inferring the underlying security model. To alleviate the need of manual workflow specifications, some researchers proposed techniques to automatically infer the security model of an application. In [11], the authors present a semi-automated approach for the reverse-engineering of SecureUML models from dynamic web applications. The task of performing security verifications on the extracted model is left to the user.

Swaddler [38] automatically learns the relationships between an application's critical execution points and internal states to infer a workflow. It then detects anomalous behaviors by reporting executions that violate the workflow. In subsequent work [51] they enhanced their tool to infer invariants of an application from its execution traces. Invariants violations are then reported at runtime. The heuristic nature of their approach makes it subject to errors.

Recently, Son et al. [152] introduced RoleCast, a tool that is specifically designed for access control vulnerabilities detection. Their tool infers the access control checks with static analysis. A vulnerability is reported if an execution path that lead to a sensitive operation misses an access control check. Their tool assumes that the only sensitive operations of a web application are: `INSERT`, `UPDATE` and `DELETE` database operations. In Moodle, ACMA identified some vulnerabilities that involve the display of sensitive information through `SELECT` operations. Moreover, their tool assumes that different roles will usually execute different files. This is not the case in Moodle as ACMA identified vulnerabilities in scripts that customize the displayed HTML according to the user's role. ACMA thus reports vulnerabilities that

their approach would fail to detect.

Detecting forced browsing vulnerabilities. In section 11.4 we compared the performances of ACMA against the tool published in [156] for the detection of forced browsing vulnerabilities. Contrary to ACMA, that detects several types of access control vulnerabilities, their approach only detects forced browsing ones. It uses and extends the string analyzer of Wassermann et al. [168, 169]. Similarly to ACMA, their approach first identifies privileged pages. It then detects forced browsing vulnerabilities by approximating the role-dependent output of the privileged page. If the output of the privileged page is identical for two roles A and B where A is more privileged than B, a vulnerability is reported.

Contrary to ACMA, their approach requires manual specification of the role-dependent entry points of the application. In PHP applications, every file is a potential entry point. In the absence of documentation or specifications about the entry points, as is the case for all the investigated applications, manual identification of the role-dependent entry points is a tedious task, especially for medium-large applications. To illustrate our point, Moodle counts about eight times more files than AWCM, the largest application investigated in [156].

Furthermore, as mentioned in subsection 11.4.1 their approach requires the manual specification of the application's "critical states", which include information such as: session, cookie, request parameter, variable and function return values as well as database records [156]. To give the reader an idea, Moodle defines approximately 19,000 functions. In that context, asking developers to identify the "critical" functions and further specify their return values seems unrealistic. Besides, this process must be repeated for session values, cookie values, and so on.

11.8 Conclusions and future work

Access control vulnerabilities are on the rise [129] and pose serious threats to web applications. In this paper, we presented ACMA, a tool for the detection and repair of access control vulnerabilities in web applications. ACMA is anchored in inter-procedural analysis and model checking theory and proved to be precise, fast and scalable.

In section 11.4, we showed how ACMA outperforms the existing tool for forced browsing vulnerabilities detection both in terms of speed (up to 890 times faster) and scalability, with positively comparable precision and false positive rates. We also showed how ACMA can handle medium-large applications with large access control models by investigating Moodle, a 400,000+ LOC application counting more than 200 distinct privileges.

Results demonstrated how ACMA can detect access control vulnerabilities of variable granularity: from vulnerable pages to ill protected statements. The finer granularity of ACMA

allowed it to detect subtle access control vulnerabilities in Moodle that would have been missed by other approaches.

In a further version, we plan to extend ACMA with an automatic access control vulnerabilities repair module. Currently, when a vulnerability is identified, ACMA has information about: the privilege that needs to be enforced, the routine to enforce the privilege and the statements that need to be privileged. This vulnerability information, combined with the control-flow information in the CFG, opens the door to automatic repair of access control vulnerabilities through automated synthesis of missing access control checks.

Acknowledgments

This research is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program.

CHAPTER 12

CORRECTIONS FOR PAPER 4

The previous chapter contains the published version of the paper entitled: "Fast Detection of Access Control Vulnerabilities in PHP Applications". The current chapter addresses comments and corrections by the jury regarding this paper.

This paper was about automatic detection of forced browsing vulnerabilities. In section 11.1, we mentioned that the term "forced browsing" was introduced in 2011 by Sun et al. [156]. However, a member of the jury observed that while this type of flaw was not known as "forced browsing" before 2011, they were already listed in the OWASP Top Ten of 2007 [128] as "Insecure Direct Object Reference". In fact, "Insecure Direct Object Reference" vulnerabilities are a superset of forced browsing vulnerabilities as they are not limited to direct access to web pages. Fortunately, the approach that we presented is perfectly suitable for the detection of general "Insecure Direct Object Reference" since objects in web applications are also accessed through hyperlinks and URLs.

In section 11.3, we presented the ACMA tool, that is based on the work that was presented in chapter 5. The technique we used for the extraction of the CFG and its conversion of to multiple model checking automata does not differ from that presented in chapter 5. We extract as many automata as there are privileges and analyze each and every one of them independently. For every automaton, the number of states equals $4 \times |V_{CFG}|$ to represent all possible combinations of Boolean privileges and security contexts ($2^2 = 4$) at every statement.

In subsection 11.3.4, we presented the technique we employed to extract privileged pages from an application where one step requires to extract hyperlinks. Some members of the jury asked how we could statically extract every hyperlinks from highly dynamic PHP applications. This is indeed a limitation of the approach and we cannot guarantee that we statically extract *every* hyperlinks from an application. In this context, dynamic analysis could be used to complement static analysis and identify hyperlinks that are created at runtime.

Another promising line of research is to use the inter-procedural pattern propagation algorithm that was presented in chapter 7 to statically resolve dynamically generated links. Indeed, in chapter 7, we showed how our pattern propagation algorithm could be adapted to resolve dynamic include statements that are built based on concatenations of strings and string variables. Since several dynamically generated hyperlinks are built based on the same concatenation logic, the same algorithm could be applied in a straightforward manner to resolve several dynamically generated hyperlinks.

In section 11.4, we mentioned that: “ACMA resolves links with a precision rate of 94% to 100%”. What we meant by that is that among all the hyperlinks that ACMA extracts, between 94% and 100% of them are resolved to valid, complete URLs. Indeed, in some cases, ACMA detects hyperlinks (e.g. strings that contain an `href` token) that it cannot resolve to actual URLs. Table 11.3 shows unresolved hyperlinks and explicits the reason why ACMA failed to resolve them. Unresolved hyperlinks were considered as false positives, hence the precision measures.

In subsection 11.6.1, we mentioned that we: “augmented ACMA with supplementary rules to report unprivileged database access statements”. To do so, we simply identified typical routines that perform database access in PHP (e.g. `mysql_query`) and extended ACMA to report any such statement that was left unprotected.

CHAPTER 13

PAPER 5: SEMANTIC SMELLS AND ERRORS IN ACCESS CONTROL MODELS: A CASE STUDY IN PHP

ABSTRACT

Access control models implement mechanisms to restrict access to sensitive data from unprivileged users. Access controls typically check privileges that capture the semantics of the operations they protect. Semantic smells and errors in access control models stem from privileges that are partially or totally unrelated to the action they protect.
This paper presents a novel approach, partly based on static analysis and information retrieval techniques, for the automatic detection of semantic smells and errors in access control models. Investigation of the case study application revealed 31 smells and 2 errors. Errors were reported to developers who quickly confirmed their relevance and took actions to correct them. Based on the obtained results, we also propose three categories of semantic smells and errors to lay the foundations for further research on access control smells in other systems and domains.

13.1 Introduction

Every day, millions of people communicate, shop, bank, gather information, and perform numerous tasks using web applications. An increasing number of web applications now deal with private or security sensitive information. Such applications must implement access control mechanisms to protect the privacy of their users. Failure to do so results in access control vulnerabilities.
Access control vulnerabilities can take several shapes. For example, several studies [62, 51, 152, 41] target the identification of missing access controls, where sensitive operations are left unprotected. While missing access controls pose serious security threats, the focus of this paper is the identification of a more subtle, but no less relevant type of access control vulnerability: semantic smells and errors. Analogously to code smells [53] that reflect poor solutions to implementation problems, semantic smells reflect poor implementations of the semantic of an access control model. Semantic errors, on the other hand, are wrong implementations that must be corrected.

In the context of access control models, users are granted with privileges that allow them to perform certain actions. As such, a privilege should capture the semantic of the action it protects. In practice, however, the semantic of the privilege does not always clearly relates to the semantic of the protected action. In some cases, the semantic relationship is only partial and gives rise to semantic smells. In other cases, the absence of relationship leads to semantic errors. Listing 13.1 shows a semantic error in Moodle where the *user:update* privilege protects the download of user information instead of the update of a user account, as specified in the documentation.

```
1 /**
2 * script for downloading of user lists
3 */
4 require_capability('moodle/user:update', CONTEXT_SYSTEM);
5 ...
6 function user_download_xls($fields) {
7 ...
8 }
```

Listing 13.1: Semantic error in Moodle. The enforced update capability is semantically unrelated to the download code it protects.

The focus of this paper is the automated identification of semantic smells and errors in access control models. The key insight behind our approach is the following: semantically related sections of source code usually perform similar actions and should therefore be protected by similar privileges. Otherwise, these sections of code may be affected by semantic smells and errors.
To our knowledge, we are the first to propose an approach for the automatic identification of semantic smells and errors in access control models. The main contributions of this paper are:

- A novel, statistically sound approach, based on static analysis, model checking and information retrieval techniques, to identify semantic smells and errors in access control models.
- A proof of concept that our approach can be applied to medium-size, open-source PHP applications, as shown by our case study of Moodle.
- Identification and classification of previously unknown semantic smells and errors in Moodle's access control model. Semantic errors were reported to developers who swiftly confirmed their relevance and took actions to correct them.

13.2 Related Work

13.2.1 Detection of Access Control Vulnerabilities

In [41], Dalton et al. proposed an approach, called Nemesis, that takes a user specified access control model as input and dynamically detects access control violations at runtime. From our experience, few developers provide such specifications.

To alleviate the need of manual workflow specifications, some researchers proposed techniques to automatically infer the security model of an application. In [51], Felmetsger et al. presented a tool to detect invariants from execution traces and report invariant violations at runtime. Semantic smells and errors induce *erroneous* invariants and cannot be detected with such strategies.

Recently, Son et al. [152] introduced RoleCast, a tool that statically detects missing access controls in Web applications. Their tool performs a control-flow *taint* analysis where unprivileged `INSERT`, `UPDATE` and `DELETE` database queries are reported as potential security flaws. From our experience, privileged actions in Web applications are not limited to database queries. For example, work by Gauthier et al. [62] targets the detection of *forced browsing* vulnerabilities, where no assumption is made about the nature of privileged actions.

All these approaches are solely based on static analysis and lack the necessary information to detect semantic smells and errors in access control models.

13.2.2 Information Retrieval in Software Engineering

In recent years, information retrieval (IR) techniques have been used for a variety of software engineering tasks.

In [15], Asuncion et al. presented an approach, based on Latent Dirichlet Allocation (LDA) [26], to retrieve traceability links between software artifacts, such as documentation, source code, tests, etc..

In [179], Zhou et al. proposed a novel IR technique, called revised Vector Space Model (rVSM), for the identification of source code artifacts that are relevant to a particular bug report, with good results.

In [69], Grant et al. proposed the use of LDA for the reverse engineering of co-maintenance relationships. Their study revealed interesting co-maintenance patterns in several systems, written in different languages.

While their goals differ, these studies share a common denominator: they show that IR can efficiently identify code artifacts that are semantically related to text documents (e.g. a bug description) or to other code artifacts (e.g. co-maintenance relationships). In the context of this paper, we use LDA to report semantically related blocks of code for which the enforced

privileges differ. To our knowledge, this is the first paper that addresses the detection of security flaws using IR techniques.

13.3 Methodology

13.3.1 Analysis Overview

As previously mentioned, the key assumption behind our work is that semantically related sections of source code usually perform similar actions and should therefore be protected by similar privileges. Otherwise, we assume there might be semantic smells and errors. In order to verify this assumption, we designed the following protocol:

1. Extract the mapping between privileges and source code using a model-checking based, inter-procedural, static analysis [60]. As a result, we obtain the list the privileges that are enforced at each statement of an application.

2. Perform unsupervised Latent Dirichlet Allocation [26] analysis to extract latent topics in the source code. It is assumed that sections of code that belong to the same latent topics are semantically related.

3. Identify the topics that are significantly associated to some privileges by the mean of logistic regression.

4. Infer the privileges that *should* protect each block of code based on the topics obtained at step 3. Report the blocks of code for which the enforced privileges differ from the semantically inferred ones.

5. Submit the observed discrepancies to developers.

13.3.2 Step 1: Mapping Privileges to Source Code

As reported in [60] and [152], access control checks in Web applications are control-flow constructs. Consider the following snippet of PHP code:

```
1 if (has_capability("user:update")) {
2   mysql_query($update_query);
3 }
```

Listing 13.2: Privilege check in Moodle. The update query is executed only if the check succeeds.

In this case, the update query is only executed if the access control check succeeds. We thus define the `mysql_query` statement as *protected* by the *user:update* privilege.
In previous work [62, 60], we presented a linear-time, inter-procedural approach to statically map privileges to protected statements. In summary, a control-flow graph (CFG), annotated with access control checks, is first extracted from the source code. The CFG is then converted to multiple and independent model checking automata, each modeling one privilege. A custom-built model checker then processes each automaton and identifies the statements that are only reached by execution paths that contain an access control check for the corresponding privilege. Statements that are reached by both privileged and unprivileged execution paths are discarded. While similar results could have been obtained using regular data-flow analysis, we consider that model checking formalism simplifies the definition and implementation of our approach.
In this study, we post-processed the results to map privileges to *blocks* of protected statements: consecutive statements, enclosed in braces, that are all protected by the same privileges.

13.3.3 Step 2: Topic Extraction with LDA

Originally developed for the analysis of natural language documents, Latent Dirichlet Allocation (LDA) [26], has been shown an efficient tool for program comprehension, bug localization and other software engineering tasks [15, 179, 69]. LDA probabilistically models text documents as mixtures of latent topics, where topics correspond to key concepts in the corpus of documents [26].
The first step toward the extraction of an LDA model from the source code is to build a collection of documents. Other studies usually define a document as a class in the source code. We adopted a slightly different approach. Experience taught us that privileges rarely protect entire classes. On the contrary, classes often comprise several blocks of code that are protected by different privileges. In order to accurately map privileges to documents, we defined documents as blocks of statements: consecutive statements enclosed in braces.
During LDA modeling, documents are treated as bags of words. The definition of "words" in a source code artifact varies from one study to another. In the context of this study, the term "word" refers to the identifiers (variable names, function names, etc.) in a block. Comments were discarded as they generally refer to whole classes or methods, not to specific blocks. Strings in Web applications often contain HTML, CSS or SQL code that mostly add noise and were also discarded.
LDA modeling was performed with the *GibbsLDA++* [174] tool with default parameters. When the modeling completes, a document-topic probability matrix is produced, showing

the probability that a given document (block) belongs to a given topic. We then combine these block-topics probabilities with the previously extracted block-privileges mapping to identify the topics that are strongly associated to some privileges.

13.3.4 Step 3: Associating Topics to Privileges

We now have two independent sources of information: on one hand, an exact mapping between privileges and blocks of protected statements and on the other hand a probabilistic mapping between blocks and latent topics. Our goal is now to identify those topics that capture the semantic of some privileges. We did so by the mean of logistic regression.
Logistic regression models the relationship between a dependent binary variable and one or more independent categorical or continuous variables. In its simplest form, a logistic regression models the influence of a single independent variable on a binary outcome:

$$\pi(X) = \frac{e^{\beta_0+\beta_1 X}}{1 + e^{\beta_0+\beta_1 X}} \tag{13.1}$$

where, (i) X is the independent variable, (ii) β_i are the model coefficients and (iii) $0 \leq \pi(X) \leq 1$ represents a value on the sigmoid regression curve. The closer $\pi(X)$ is to 1, the higher the likelihood that the outcome is true. In the context of this study, we investigated the relationship between binary privileges (protected or not) and latent topics.
Training a logistic regression models involves estimating the values of the β_i coefficients in such a way to maximize the fit between the sigmoid regression curve and the observed outcomes. By the end of the training phase, a p-value is associated to each independent variable, representing the significance of the association between the variable and the outcome.
When performing logistic regression, one of the main pre-requisite is for independent variables to be uncorrelated. LDA, however, provides no strong guarantee about the correlation between the extracted topics. In order to subtract our study from this statistical bias, we modeled the relationship between each privilege and each latent topic with a separate logistic regression model, resulting in 30,700 (100 topics $\times$ 307 privileges) logistic regressions.
On the other side, performing such a high number of statistical tests induces a higher probability of Type II errors, where totally random associations are misinterpreted as significant. All the p-values were thus corrected for multiple testing using the Bonferroni correction, where the p-value is multiplied by the number of statistical tests. Assuming a threshold of 0.05, it means that the original p-value, had to be $< 1.63 \times 10^{-6}$ to be considered significant after the Bonferroni correction.

13.3.5 Step 4: Inferring Privileges Based on Topics

Once a logistic regression model is trained, it is possible to supply it with new observations in order to determine the likelihood of the outcome. For each topic that was found to be significantly associated to a privilege (corrected p-value < 0.05), we performed privilege prediction on blocks, based on their topic probability.

The idea behind this procedure is to infer the privileges that *should* protect a block of code, based on semantic information. The output of the prediction is a value between 0 and 1, representing the probability that the block of code is protected by the privilege. In the context of this study, we fixed the prediction threshold at 0.95. In order to identify faulty access controls, we investigated the blocks of code for which the enforced privilege differs from the semantically predicted one.

13.4 Results

As a proof of concept, we applied this methodology to Moodle, an open-source PHP course management system with an elaborate access control model. Moodle 2.3.2 counts 735,485 sLOC and 307 privileges.

Applying the proposed methodology, we obtained a list of 83 blocks for which the semantically inferred privilege differed from the enforced one. After a post-filtering step to eliminate embedded blocks of code, we obtained a list of 59 blocks.

Manual inspection of the results was completed in around two hours and revealed 31 smells and 2 errors that were further classified in three categories. Reported smells and errors were reviewed by a group of experts who assessed their significance. Table 13.1 details the categories and numbers of semantic smells and errors that were identified. Errors were reported to developers who quickly confirmed them and took actions to correct them. Overall, we identified two types of semantic smells and one type of semantic error:

1. Implicitly granted privileges. This smell captures the fact that some privileges are *implicitly* granted with other privileges. The *lesson:edit* and *lesson:manage* privileges are recurring examples of this smell in Moodle. As expected, routines that manage lessons are protected by the *lesson:manage* privilege and routines that edit lessons are protected by the

Table 13.1: Semantic smells and errors in Moodle's access control model

	Name	# Occurrences
Smells	Implicitly granted privileges	15
	Semantically related privileges	16
Errors	Privileges used as a role	2

lesson:edit privilege. However, edit routines often also perform manage related operations without being explicitly protected by the *lesson:manage* privilege. In that perspective, it seems that the *lesson:manage* privilege is *implicitly* granted with the *lesson:edit* privilege.

2. Semantically related privileges. In Moodle, this smell reflects a bad coupling between privileges and components. Consider this example: Moodle counts 82 view-related privileges. All of these privileges share a common semantic: they grant the right to "view", but for different components of the system. This coupling between privileges and components led to a steady increase of the number of privileges as Moodle evolved. In approximately three years, from version 1.9.5 to version 2.3.2, the number of privileges in Moodle grew from 217 to 307, mostly due to semantically related privileges.

3. Privileges verified in place of a role. This semantic error characterizes blocks of code that are protected by semantically unrelated privileges in place of a proper role verification. For example, in Moodle, the *user:delete* and *user:update* privileges are owned by administrators only. According to Moodle's documentation, these two privileges respectively grant the rights to update and delete a user account. However, we observed that they also protect semantically unrelated blocks of code that are responsible for the download of user information (see Listing 13.1) and the display of private data. Those two cases were submitted to developers (see [121] [122]) as potential bugs.

13.5 Discussion

Implicitly granted privileges qualify as bad smells because of the confusion they cause among users, as shown by a discussion thread about the *lesson:edit* and *lesson:manage* privileges [120]. Interestingly, there exists some theoretical access control models that can explicitly handle such constraints between privileges [1].

Semantically related privileges increase the complexity of the access control model. Interestingly, semantically related privileges that are spread across several components can be straightforwardly modeled as cross-cutting concerns. Aspect-oriented approaches [157] seem well tailored to handle such type of semantic smells. Alternatively, since semantically related privileges are often granted to a very limited number of roles, many of these privilege checks could be replaced by a few proper role verifications.

Privileges used in place of a role suppose an equivalence relation between the privileges and the role. While such a relation might exist in the default configuration of the access control model, nothing prevents users from breaking it by altering the default model. Two occurrences of privileges used in place of a role were identified in Moodle.

Both cases were submitted to Moodle's bug tracker (see [121, 122]) as potential security

issues. In order to minimize bias toward acceptance or rejection, both bug reports were filled in the most objective manner, deliberately omitting to mention university affiliation or the use of a research tool. For both bug reports, we obtained a response in less than 2 weekdays. A clarification discussion ensued and led to the acceptance of our claim that these pieces of code were inadequately protected. In one case, developers agreed to correct the bug in the next minor release. In the other case, they agreed to introduce a new, semantically related privilege in the next major release.

13.6 Conclusion and Perspectives

Access controls enforce protection by checking privileges that capture the semantic of sensitive operations. In this paper, we presented a novel approach for the identification of semantic smells and errors that can hinder the comprehension, increase the complexity and threaten the security of access control models.

The proposed methodology contrasts enforced privileges to semantically inferred ones and report discrepancies. Investigation of Moodle's access control model revealed 31 semantic smells and 2 semantic errors, distributed in 3 categories. Semantic errors were reported to developers who quickly confirmed their relevance and took actions to correct them.

The presented results are preliminary. In a further study, we plan to: (i) investigate several systems and domains to validate the proposed categories of semantic smells and errors and discover new ones, (ii) evaluate the accuracy of the logistic regression model through cross-validation experiments and (iii) test alternative approaches for topic extraction.

Acknowledgments

The authors wish to thank Scott Grant for the inspiring discussions that sparked this work. This research is funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program.

CHAPTER 14

CORRECTIONS FOR PAPER 5

The previous chapter contains the published version of the paper entitled: "Semantic Smells and Errors in Access Control Models: A Case Study in PHP". The current chapter addresses comments and corrections by the jury regarding this paper.
In subsection 13.3.4 we explained that: "we modelled the relationship between each privilege and each latent topic with a separate logistic regression model, resulting in 30,700 (100 topics × 307 privileges) logistic regressions" and a member of the jury remarked that this needed more explanations. For the sake of clarity, here is the exhaustive list of steps that we performed:

1. Model the program as a set of documents, where each document corresponds to a block of code.

2. Label each document with the set of privileges that protect it. This information comes from SPT analysis.

3. Model each documents as a distribution of latent topics using LDA.

4. Remove documents that are not protected by any privilege. The reason for this is that we aim to identify documents that are *protected* by wrong privileges, not to find documents that miss an access check.

5. For each privilege, partition remaining documents in two classes: protected and not protected by the privilege. These two classes will become the outcome of the logistic regression models.

6. For each privilege and each latent topic, train a logistic regression model where the binary outcomes are the classes that were defined in the previous step and the independent variable is the latent topic.

7. The previous step results in 30,700 (100 topics × 307 privileges) logistic regression models.

Once the models are trained, our goal is to use them to infer wrong access checks in the application on which they were trained. Consequently, contrary to standard machine learning studies that aim to build models that generalize to other applications, we actually want our

models to be specific to the application on which they were trained. For this reason, we decided to maximize the cardinality of our training set by including the whole dataset. As a consequence, test data is necessarily drawn from the training set.

While this is a limitation of the approach, experience revealed that it still yields interesting results. The reason is that the vast majority of access checks in a system are correctly implemented and wrong access check are the exception. Hence, the impacts of wrong access checks in the training set are negligible. In theory, the best approach would be to perform leave-one-out cross-validations on every regression model. However, given that the cardinality of the training set corresponds to the number of protected blocks of code in the system, the number of models to train and test would be very large (100 topics $\times$ 307 privileges $\times$ number of protected blocks). Furthermore, the larger the number of experiments, the more stringent the Bonferroni correction and the lesser the chances of reporting statistically significant results.

On a similar line of thoughts, a member of the jury observed that we fixed the prediction threshold at the rather high value of 0.95. The reason for this is simple. This study was the first to target the identification of semantically wrong access checks. The concept had yet to be validated by developers and we wanted to submit the most plausible cases only, hence the stringent prediction threshold.

CHAPTER 15

PAPER 6: UNCOVERING ACCESS CONTROL WEAKNESSES AND FLAWS WITH SECURITY-DISCORDANT SOFTWARE CLONES

ABSTRACT

Software clone detection techniques identify fragments of code that share some level of syntactic similarity. In this study, we investigate security-sensitive clone clusters: clusters of syntactically similar fragments of code that are protected by some privileges. From a security perspective, security-sensitive clone clusters can help reason about the implemented security model: given syntactically similar fragments of code, it is expected that they are protected by similar privileges. We hypothesize that clones that violate this assumption, defined as security-discordant clones, are likely to reveal weaknesses and flaws in access control models.

In order to characterize security-discordant clones, we investigated two of the largest and most popular open-source PHP applications: Joomla! and Moodle, with sizes ranging from hundred thousands to more than a million lines of code. Investigation of security-discordant clone clusters in these systems revealed several previously undocumented, recurring, and application-independent security weaknesses. Moreover, security-discordant clones also revealed four, previously unreported, security flaws. Results also show how these flaws were revealed through the investigation of as little as 2% of the code base. Distribution of weaknesses and flaws between the two systems is investigated and discussed. Potential extensions to this exploratory work are also presented.

15.1 Introduction

Software clones, that are blocks of code that share some level of syntactic similarity, have attracted the attention of the research community for more than a decade. Many studies have been dedicated to the development and enhancement of clone detection tools in such a way that clone detectors now identify syntactically similar fragments of code in a fairly fast and consistent manner. However, investigation and evaluation of ways to use clone information to solve specific problems received comparatively little attention.

In this paper, we investigate a specific subset of software clones, called security-sensitive

clones. Security-sensitive clones refer to cloned fragments that are protected by some privileges in the system. From a security perspective, security-sensitive clones are of particular interest as they capture syntactically similar fragments of code that perform privileged actions. In the absence of security specifications, security-sensitive clones can help developers reason about the implemented security model by supplying examples of similar pieces of code together with their enforced privileges.
On the other hand, security-sensitive clones can also help reveal discrepancies in the implemented security model. Intuitively, one expects syntactically similar fragments of code to be protected by similar privileges. Security-sensitive clones that violate this assumption, defined as security-discordant clones, will be the focal point of this paper. The primary goal of our study is to investigate security-discordant clones in the context of access control models in order to characterize them and evaluate whether they are symptomatic of security weaknesses and flaws.
In order to attain this goal, we investigated security-discordant clones in two of the largest and most popular open source PHP applications: Joomla! and Moodle. Our findings revealed several recurring security weaknesses that can hinder the maintainability, understandability and strength of security models. Investigation of security-discordant clones also revealed four new, previously unknown, security flaws in the investigated systems as well as a number of known flaws. The original contributions of this paper are many:

- To our knowledge this is the first study about security-discordant clones.
- Analysis of security-discordant clones, revealed several recurring categories of security weaknesses that hinder the maintainability, understandability and strength of security models.
- Our analysis also revealed four, previously unknown, security flaws in the investigated systems.

15.2 Motivating example

In the context of this study, we investigated security-discordant clones in the light of access control models. This section motivates the use of security-discordant clones in that context with a concrete example from Joomla!.
Joomla! is popular content management system, used by millions of users, to build different kinds of websites. It is divided in two parts: the front-end, that is publicly accessible, and the back-end, where administrator functions are found. The back-end itself is further divided

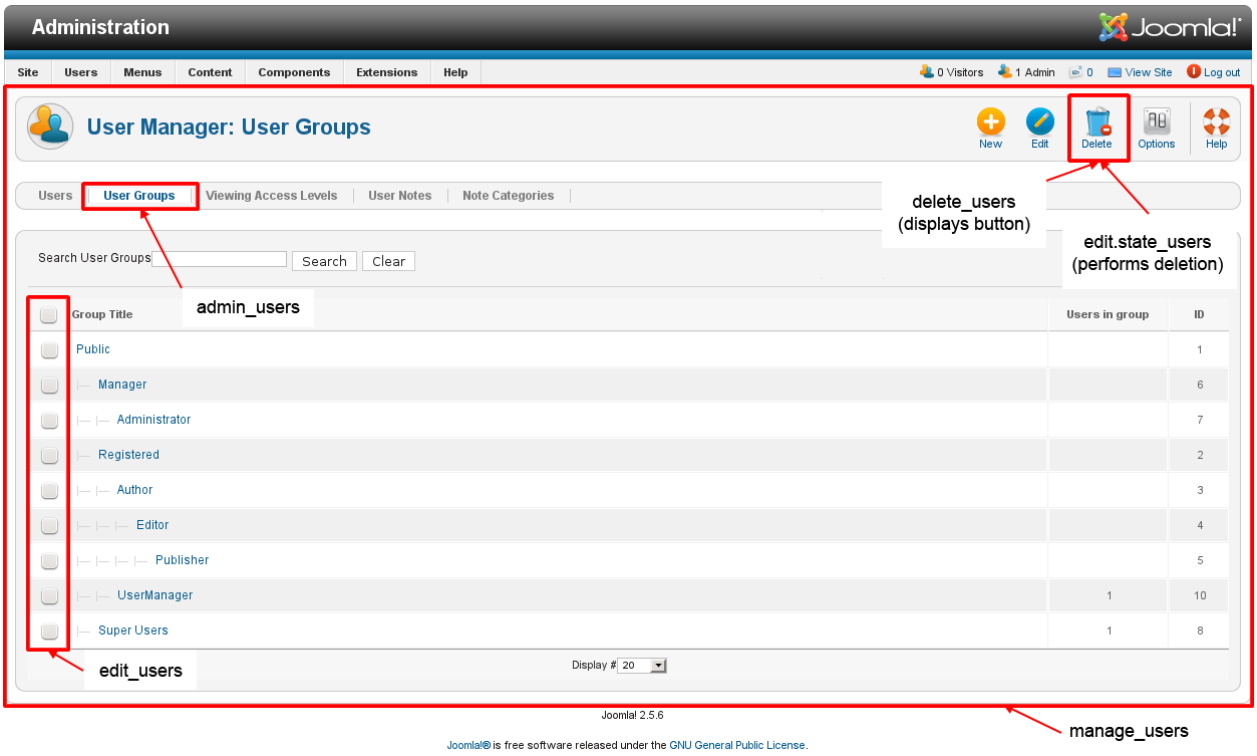


Figure 15.1: Privileges associated to different parts of the UI in Joomla!. Five privileges are required to delete a user group, while similar actions elsewhere in the system only require two. Requiring more privileges than necessary to perform an action violates the Least Privilege Principle.

in several "components" that are equivalent to modules and that control specific parts of the application. Privileges restrict access to back-end components in Joomla!.

Joomla! defines 6 basic privileges: admin, manage, create, edit, edit.state and delete. Each of these privileges can be granted on a per-component basis. For example, in the context of the "users" component, the following privileges can be granted: admin_users, manage_users, create_users, edit_users, edit.state_users and delete_users.

From a syntactic point of view, components in Joomla! often share a high level of similarity. For example, the code required to delete a user is syntactically similar to the code required to delete a plugin and both code fragments form a clone cluster. From a security point of view, these fragments are protected by the delete privilege and thus form a *security-sensitive* clone cluster. In this case, the privilege that is enforced is consistent with the intuition that syntactically similar fragments of code should be protected by similar privileges.

Security-discordant clones violate this intuition. For example, figure 15.1 shows an annotated screenshot from Joomla!. It illustrates the privileges required to delete a "user group": the

equivalent of a role in RBAC models. Observation of similar functions (clones) elsewhere in the system leads to the intuition that deleting a user group requires two privileges: manage_users to access the back-end interface of the users component and delete_users to delete a member of the users component. Surprisingly, this is far from the truth.
Figure 15.1 shows that manage_users and delete_users are indeed required, but not sufficient. To delete a user_group, one also needs the admin_users privilege to see the "user group" tab, the edit_users to see the checkboxes that are required to select the groups to be deleted and the edit.state_users privilege to actually perform the deletion since the delete_users privilege only displays the "delete" button! In this context, administrators are forced to violate the Least Privilege Principle by granting more privileges than necessary. Requiring counter-intuitive combinations of privileges to perform seemingly simple actions also hinder the understandability and maintainability of an access control model.
This case illustrates one of the many problematic situations that were brought to light by security-discordant clones. Indeed, these discrepancies were revealed trough the investigation of the clones of the "delete user group" function, which are themselves correctly protected. Security-discordant clones also revealed several security weaknesses and four, previously unreported, security flaws.

15.3 Methodology

15.3.1 Clone detection

The syntactic clone detection technique used in this experiment was presented in [97]. It computes a similarity distance using lexical features. Without providing the detailed algorithm, we will cover the basic steps of the method and the chosen parameters for this experiment.
Using a custom PHP parser, we extract the tokens of each block of code. After parsing, blocks are represented as strings of integers, where each integer conceptually represents a token of the PHP language. These strings of integers are then post-processed in order to compute the frequencies or their 2-grams, that are the ordered pairs of consecutive characters in the string. After post-processing, each block is represented by the frequency vector of its 2-grams.
Afterwards, a metric tree is built with the set of all frequency vectors. A metric tree [97] is a specialized data structure that optimizes the efficiency of several kinds of similarity queries. In this paper, we use range-queries, that take an element and a radius as parameters and return elements for which the distance to the queried element is below the specified radius. In this experiment, we perform a range query with a radius 0.3 for each frequency vector (block) in the metric tree. The resulting set contains the clones of the queried block. Clone clusters are then computed with the union-find algorithm [37] by transitively merging sets of

clones that share at least one element. These clone clusters, become the input for the next steps of the analysis.

15.3.2 Security pattern traversal

Security-sensitive clone clusters encompass syntactically similar fragments of code that are protected by some privileges. In the context of this paper, we focus our attention on access control models and define security-sensitive clones based on the access privileges that are enforced in cloned fragments.
Access control models define privileges that protect pieces of code which perform specific security-sensitive actions. Security-discordant clones thus reveal fragments of code that are syntactically similar but that are not protected by the same access privileges.
Identification of security-discordant clones thus requires to identify the access privileges that are enforced at every point of an application. To achieve this goal, we perform security pattern traversal analysis (SPT), that is a static, inter-procedural and security-sensitive analysis. In this section, we provide a brief overview of SPT. We implemented the algorithms presented in [100, 61] and refer the interested reader to these papers for algorithmic details.

Control-flow graph extraction

Our experimental setup uses a PHP parser generated by JavaCC, a common parser generator tool. The parser outputs a program Control Flow Graph (CFG), annotated with application-specific access checks.
Identifying access control routines involves finding some syntactic patterns in the Abstract Syntax Tree (AST). While these patterns are application-dependent, we observed that they usually are very similar in their syntactic structures. In the investigated applications, access check functions receive a string argument, representing the privilege, and return a Boolean value asserting whether or not the user owns the privilege. In the context of this study, access checks are detected at the AST level using visitors [37]. While the amount of effort required for their implementation varies from one system to another, on average, approximately 100 lines of highly stereotyped Java code are sufficient to properly capture access control routines in a system.
The access control routines that are identified at the AST level produce *grant* and *revoke* edges in the CFG. For example, suppose an access check inside an `if` statement. If the predicate evaluates to true, the control is transferred to a block in which the privilege p_i is *granted* until block end. Conversely, if the predicate evaluates to false, the control is transferred to another block in which the privilege p_i is *revoked*.

Noteworthy is the fact that the extracted CFG is approximate. Indeed, the dynamic nature of the PHP language prevents us from extracting sound and complete call graphs. Our approach currently only considers direct function calls, in which the name of the called function appears explicitly in the source code, similarly to [153]. Over time, however, SPT has proven to be very accurate in reporting privileges that *must* be granted at execution time.

Conversion to model checking automaton

Prior to computing SPT analysis, the CFG produced by the PHP parser is transformed into an automaton A suitable for model checking.
Conceptually, we model the granting and revoking of privileges as the activation and deactivation of Boolean properties. To model these Boolean properties, each node x in the CFG is duplicated in two states $s_{x,0}$ and $s_{x,1}$, where $s_{x,0}$ represents that the property is false and $s_{x,1}$ represents that the property is true. During the model extraction phase, *grant* edges become transitions to $s_{x,1}$ states and *revoke* edges become transitions to $s_{x,0}$ states.
From the inter-procedural perspective, our tool performs a *security*-sensitive analysis. Unlike context-sensitive analysis, that distinguishes every calling context, security-sensitive analysis only distinguishes calling contexts with different security levels. At the automaton level, each state is further duplicated in two versions, representing the two possible security contexts for a given privilege. At the end of the model extraction phase, the number of states in the automaton equals $4 \times |V_{CFG}|$.

Reachability analysis

Security pattern traversal (SPT) analysis is the process by which we determine what privileges are enforced at each statement of an application. In the extracted automaton, access checks are conceptually modeled as privilege granting *patterns*. Whenever a path in the automaton passes through such a pattern, the associated privilege is granted and carried along the rest of the path. By the end of the analysis, a report indicates the privileges that are enforced at each statement.
Since the Boolean privileges and security contexts are directly encoded in the states of the automaton, SPT analysis can be expressed as a state reachability problem over the automaton. More precisely, a statement $stmt_j$ is defined as protected by the privilege p_i if the following formula is verified:

$$\Diamond stmt_j \wedge \Box(\ stmt_j \wedge p_i) \tag{15.1}$$

meaning that the statement $stmt_j$ is eventually reached ($\Diamond stmt_j$) and always when $stmt_j$ is reachable, the privilege p_i is granted ($\Box(\ stmt_j \wedge p_i)$).

Without going into details, the reachability analysis has a complexity that is linear in the number of states in the automaton [100]. Consequently, SPT analysis is unbounded and does not induce approximations other than the ones inherent to control-flow graph extraction.

15.3.3 Identification of security-sensitive clone clusters

Given that clone clusters are available and that SPT analysis reported the privileges that are enforced at every statement of the system, one can identify security-sensitive clone clusters. Formally, given $C = \{c_1, c_2, \ldots, c_n\}$ the set of clone clusters in a system, and $c_i = \{f_1, f_2, \ldots, f_n\} \mid c_i \in C$ the set of syntactically similar fragments of code contained in the cluster c_i, we want to identify $C_{ss} \subset C$, the subset of security-sensitive clone clusters.
Given $P = \{p_1, p_2, \ldots, p_n\}$ the set of privileges, and $S = \{s_1, s_2, \ldots, s_n\}$ the set of statements in the system, SPT analysis reports $P_{s_i} = \{e_1, e_2, \ldots, e_m\} \mid e_i \in P, s_i \in S$, the set of privileges that are enforced at each statement.
In that context, a security-sensitive clone cluster can be defined as a set of syntactically similar code fragments where at least one fragment is protected by some privilege. Similarly, a security-discordant clone cluster can be defined as a set of code fragments where the enforced privileges differ between at least two fragments.

15.3.4 Investigation of security-discordant clones

Security-sensitive clone clusters report syntactically similar fragments that are protected by some privileges. As defined earlier, security-discordant clones are security-sensitive clones for which the enforced privileges differ. The fundamental assumption behind our work is that the vast majority of fragments in a security-discordant clone cluster are correctly protected and that security-discordant clones should therefore draw reviewers' attention on the minority of fragments that are inadequately protected.
Since we are the first to investigate security-discordant clone clusters, we had no expectation about the security weaknesses they would reveal, if any. Consequently, in the first screening of the results, we simply isolated security-discordant clone clusters that raised questions among reviewers from those where the discordance seemed justified. In a second step, we re-visited the first category of clusters and observed five recurring categories of security weaknesses. A description of each category is presented in section 15.5.
In the context of this paper, we distinguish security weaknesses from security flaws. Security weaknesses are defined as poor implementation choices that might induce security flaws. Security flaws, on the other hand, stem from security weaknesses and can be readily exploited

to gain unauthorized access to privileged resources. In the context of this study, we identified several security weaknesses and four, previously unknown, security flaws.

15.4 Corpus

For this study, we investigated two of the largest and most popular open source PHP applications: Joomla! and Moodle. According to *Ohloh*[1], a site that tracks open-source projects' popularity in terms of active users, both applications rank in the top 20 most popular PHP applications.

Table 15.1 shows characteristics of the investigated applications. The reported numbers of lines of code include blank lines and comments and were calculated with the cloc.pl software[2]. Joomla! and Moodle are all mature applications, their development dating back to 2005 and 2001 respectively, all have millions of active users and developers worldwide, and all have dedicated security teams that are in charge of performing code reviews and repair security flaws.

15.5 Results

Identification of security-discordant clone clusters can be summarized in three steps: identification of the clones in the system under study, Security Pattern Traversal (SPT) analysis, and computation of security-discordant clones.

Table 15.2 shows the computation times of each step together with the numbers of PHP lines of code and privileges, which are the two major factors impacting computation time. Computation times are only provided as an indication since different steps of the analysis were computed on different machines.

Figure 15.2 reports the percentages of LOCs and files that belong to some security-discordant clone cluster. In Joomla!, approximately 8% of the files and 4% of the PHP code base belong

[1]See: http://www.ohloh.net/tags/php
[2]See: http://cloc.sourceforge.net

Table 15.1: Corpus of investigated applications

Application	Version	Release date	Description	Files	LOC	
					PHP	Total
Joomla!	2.5.4	02-04-2012	Content management system	2,286	266,458	403,222
Moodle	2.3.2	10-09-2012	Course management system	7,480	1,245,417	2,049,301

Table 15.2: Computation times for each step of the analysis together with numbers of privileges and PHP lines of code.

Application	Computation times			Privileges	PHP LOC
	Clones	SPT	Security-discordant clone clusters		
Joomla! 2.5.4	38m27s	4m26s	27s	81	266,458
Moodle 2.3.2	10h12m24	2h43m36s	7m26s	307	1,245,417

to a security-discordant clone cluster. In Moodle, 12% of the files but only 2% of the PHP code base do.

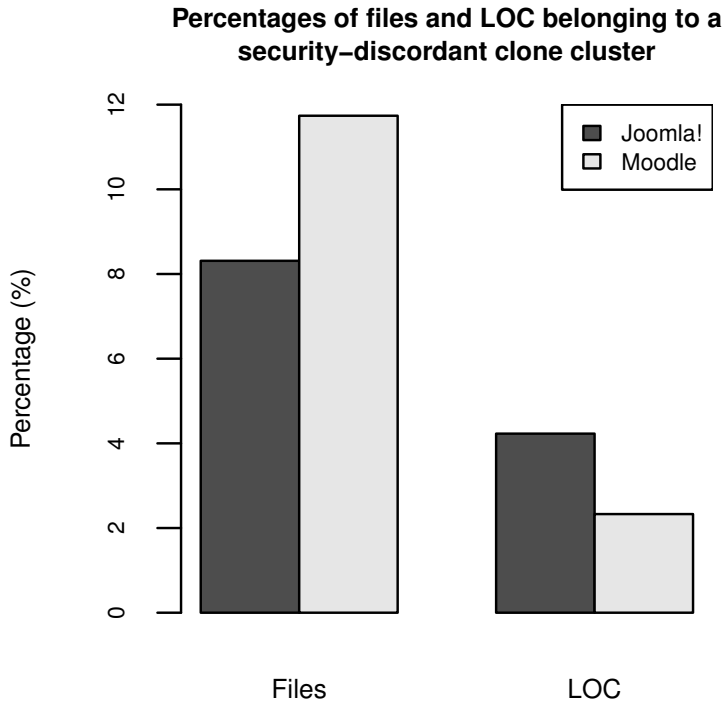


Figure 15.2: Percentages of files and PHP lines of code that belong to a security-discordant clone cluster.

Five categories of security weakness emerged during the investigation of security-discordant clones clusters: *semantic inconsistency*, *shaky logic*, *weak encapsulation*, *least privilege violation*, and *missing privilege*. In order to facilitate the comparison of our results with other studies, we mapped our categories of security weakness to CWE weakness ids, provided by

the Common Weakness Enumeration (CWE) organism[3]. Below, we provide a brief description of each category together with the corresponding CWE ids and their rank in the 2011 CWE/SANS Top 25 Most Dangerous Software Errors [40], when available.

1. **Semantic inconsistency** Clusters that fall in this category reveal fragments that are protected by privileges which are semantically unrelated to the action they perform. Determining the privileges that *should* protect a fragment of code fall within the area of security specifications. When security specifications are not available, as is the case for Joomla! and Moodle, identification of semantic inconsistency cases requires human judgment. In some recent work, however, authors have tackled the problem of automatically identifying semantic inconsistencies in the absence of security specifications [63]. Semantic inconsistency is related to CWE-285: Improper Authorization.

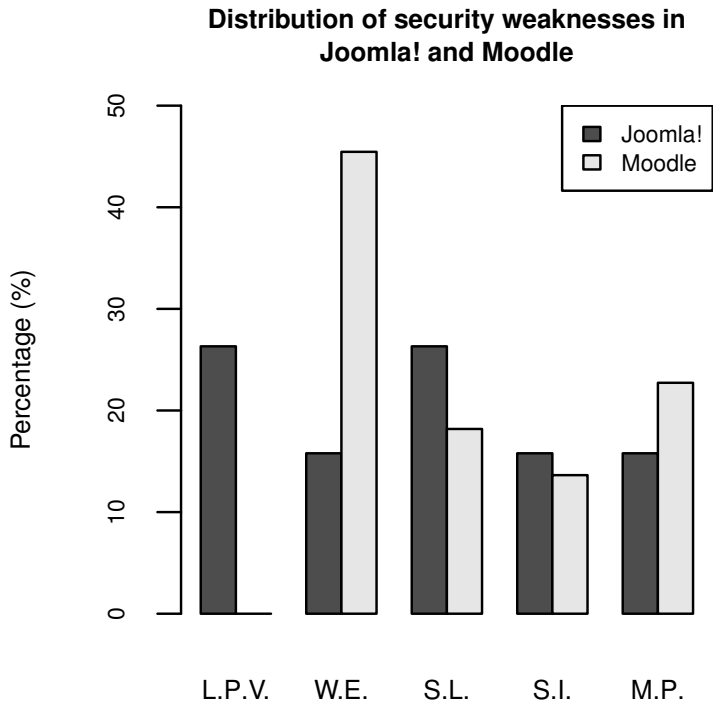
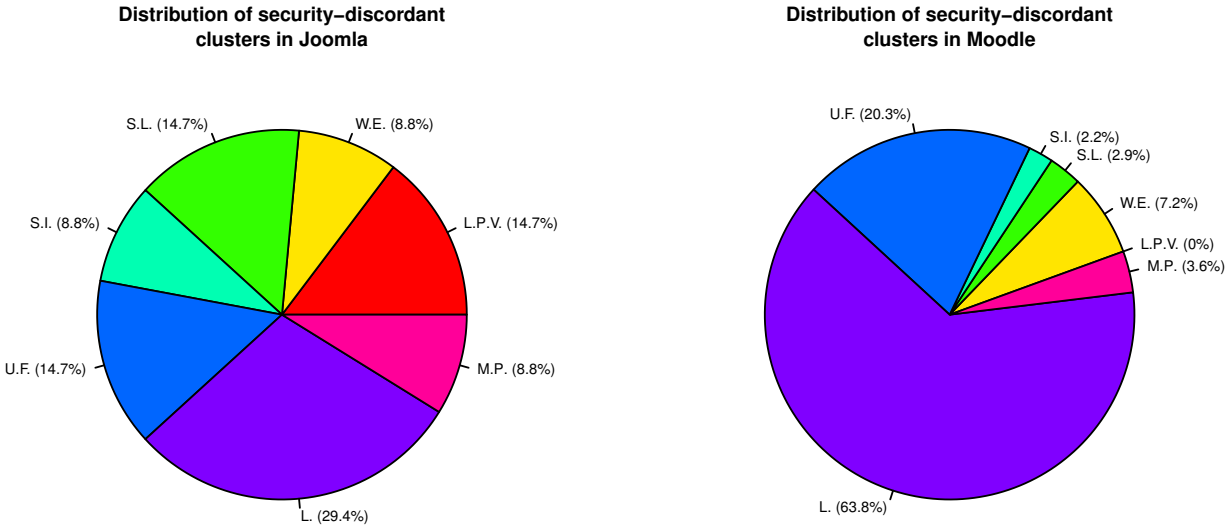


Figure 15.3: Distribution of security weaknesses in Joomla! and Moodle. Category names were abbreviated for space considerations: Least privilege violation (L.P.V.), Weak encapsulation (W.E.), Shaky logic (S.L.), Semantic inconsistency (S.I.) and Missing privilege (M.P.)

2. **Shaky logic**: The fragments that compose these clusters differ from a protection perspective but the logic behind the enforced privileges is not clear. In the context

[3]See: http://cwe.mitre.org



(a) Joomla! (b) Moodle

Figure 15.4: Distribution of the categories of security-discordant clone clusters in Joomla! and Moodle. Category names were abbreviated for space considerations: Least privilege violation (L.P.V.), Weak encapsulation (W.E.), Shaky logic (S.L.), Semantic inconsistency (S.I.), Missing privilege (M.P.), Utility function (U.F.) and Legitimate (L.)

of this study, we defined shaky logic cases as fragments that intermix privilege checks with external functions that are not related to access control. Shaky logic cases are related to CWE-637: Unnecessary Complexity in Protection Mechanism and CWE-863: Incorrect Authorization (15th).

3. **Weak encapsulation**: Security sensitive functions that do not perform access checks but that are always called from privileged contexts are considered weakly encapsulated. Where classic object-oriented encapsulation strictly restricts access to the encapsulated code, weak encapsulation relies on developers' discipline. Weak encapsulation is often found in administrator libraries where access checks are *expected* to be performed before calling the library function. Weak encapsulation is most closely related to CWE-749: Exposed Dangerous Method or Function and less specifically to CWE-863: Incorrect Authorization (15th).

4. **Least privilege violation**: Least privilege violations occur when fragments are protected by privileges that are too elevated for the task at hand. Least privilege violations are related to CWE-250: Execution with Unnecessary Privileges (11th).

5. **Missing privilege**: As its name suggests, this category encompasses clusters where

some fragments miss a privilege check. Missing privileges are related to CWE-862: Missing Authorization (6th).

Figure 15.3 shows the distribution of security weaknesses (true positives) in both systems. Bars represent the percentage of a particular type of weakness with respect to all weaknesses in a system. For space considerations, category names were abbreviated in the plot: Least privilege violation (L.P.V.), Weak encapsulation (W.E.), Shaky logic (S.L.), Semantic inconsistency (S.I.) and Missing privilege (M.P.).

Two obvious differences can be observed between both systems. First, no least privilege violations were identified in Moodle while they form a little more than 25% of the weaknesses in Joomla!. Second, weak encapsulation weaknesses are about three times more frequent in Moodle than in Joomla!. Percentages of other types of weaknesses are roughly comparable between both systems.

Figures 15.4(a) and 15.4(b) show the distribution of the categories of security-discordant clone clusters in the investigated applications. Security-discordant clusters that did not reveal security weaknesses (false positives) were classified in two additional categories: *utility functions* and *legitimate*. Security-discordant clusters of utility code fragments are classified in the utility functions category, which is abbreviated as U.F. in the figures. The second category encompasses security-discordant clone clusters for which the reported differences in privilege enforcement are justified. Such security-discordant clone clusters are classified in the legitimate category, which is abbreviated as L. in the figures.

Figures 15.4(a) and 15.4(b) show that 56% of security-discordant clusters in Joomla! and 16% security-discordant clusters in Moodle revealed security weaknesses (true positives). In the following paragraphs we provide detailed code examples for some categories of security-discordant clone clusters.

Listing 15.1 shows an example of a least privilege violation in Joomla!. Both cloned fragments define the canDelete function. The fragment on the left correctly verifies the delete privilege. The fragment on the right unnecessarily verifies the admin privilege, which ranks the highest in the privilege hierarchy. The admin privilege allows a user to grant himself supplementary privileges.

Listing 15.2 shows a shaky logic case from Moodle where both fragments contain complex conditional statements intermixing privilege checks and external functions. The conditional statement on the left only succeeds if the first three predicates are true and the user does *not* have the mod/workshop:manageexamples privilege. The conditional statement on the right succeeds if the first two predicates are true and the user *does* have the mod/workshop:submit privilege but *not* the mod/workshop:manageexamples privilege.

Listing 15.3 shows an example of semantic inconsistency where two fragments that both grant

access to a chat room in Moodle are protected by different privileges. The fragment on the left correctly checks the mod/chat:chat privilege. The fragment on the right only checks that the user is not a guest. As a consequence, users that do not have the mod/chat:chat privilege are able to bypass access checks and enter chat rooms without authorization. We were the first to report this flaw.
Listing 15.4 shows an example of missing access check. It shows three header files from different administrator components in Joomla!. Observe that the header file on the right misses an access check. This flaw was known at the time of the analysis but would have been detected by our approach.

15.5.1 Security flaws

Investigation of security-discordant clone clusters revealed four, previously unknown, security flaws in the investigated systems as well as five previously known flaws. Table 15.3 summarizes the numbers of novel and known access control flaws that were detected by our approach.
Known flaws that were missed by our approach are located in fragments that do not belong to a security-discordant clone cluster. Consequently, either these fragments do not share a minimal level of syntactic similarity with other fragments in the system or the clone cluster they belong to is not security-discordant. Both flaws that were missed in Moodle occurred in fragments that did not share a minimal level of syntactic similarity with other fragments.
Known flaws that were detected by our approach fall in several categories. As shown in Listing 15.4, the one known flaw in Joomla! (CVE-2012-2747) is of the *missing privilege* type. In Moodle, three of the previously known flaws that were detected (CVE-2012-6100, CVE-2012-6098 and CVE-2013-1835) are of the *shaky logic* type. The other one (CVE-2012-5481) is of the *semantic inconsistency* type.
Two novel access control flaws were identified in Joomla!. Both are of the *missing privilege* type. The first one (CVE-2013-3056) allowed remote authenticated users to bypass intended privilege requirements and delete the private messages of arbitrary users. The second one (CVE-2013-3057) allowed remote authenticated users to bypass intended privilege requirements and list the privileges of arbitrary users.
Two novel access control flaws were also identified in Moodle. The first one (CVE-2013-2246) is of the *missing privilege* type and allows remote authenticated users to bypass intended privilege requirements and access reports they should be denied access to. The second one (CVE-2013-2242) was presented in Listing 15.3 and allows remote authenticated users to bypass intended privilege requirements and access a chat room without the mod/chat:chat privilege. This flaw is of the *semantic inconsistency* type.

```
protected function canDelete($record)
{
  ...
  $user = JFactory::getUser();
  return $user->authorise('core.delete', '
       com_menus.item.'.(int) $record->id);
}
protected function canDelete($record)
{
  ...
  $user = JFactory::getUser();
  return $user->authorise('core.admin', '
       com_redirect');
}
```

Listing 15.1: Example of a least privilege violation in Joomla. The fragment on the right verifies a privilege that is too elevated in the privilege hierarchy.

```
if ($workshop->useexamples and
    $workshop->examplesmode and
    $phase->isreviewer and
    !has_capability('mod/workshop:
    manageexamples'))
{
  $task = new stdclass();
  ...
  $phase->tasks['examples'] = $task;
}
if ($workshop->useexamples and
    $workshop->examplesmode and
    has_capability('mod/workshop:submit') and
    !has_capability('mod/workshop:
    manageexamples'))
{
  $task = new stdclass();
  ...
  $phase->tasks['examples'] = $task;
}
```

Listing 15.2: Example of shaky logic in Moodle. Both fragments intermix privilege checks with external functions in complex conditional statements.

```
require_capability('mod/chat:chat', $context);


/// Check to see if groups are being used here
if ($groupmode = groups_get_activity_groupmode(
     $cm)) {
  ...
}
if (isguestuser()) {
    print_error('noguests', 'chat');
}
/// Check to see if groups are being used here
if ($groupmode = groups_get_activity_groupmode(
     $cm)) {
  ...
}
```

Listing 15.3: Example of semantic inconsistency security flaw in Moodle. Both fragments grant access to a chat room but the fragment on the right verifies a semantically inconsistent privilege.

```
1 defined('_JEXEC') or die;
2
3 if (!authorise('core.manage', '
      com_plugins'))
4   //return an error
5
6 jimport('joomla.application.
      component.controller');
```

```
7 defined('_JEXEC') or die;
8
9 if (!authorise('core.manage', '
      com_weblinks'))
10  //return an error
11
12 jimport('joomla.application.
      component.controller');
```

```
13 defined('_JEXEC') or die;
14
15
16 // Missing access check
17
18
19 jimport('joomla.application.
      component.controller');
```

Listing 15.4: Example of a missing privilege security flaw (CVE-2012-2747) in Joomla!. The header file of the finder component, on the right, misses an access control check, as highlighted by the two other fragments.

15.6 Discussion

Investigating clones from a security perspective adds interpretive power to otherwise structurally similar fragments of code. In the context of this paper, we investigated security-discordant clone clusters in two major PHP systems and classified the weaknesses they revealed in five categories: *semantic inconsistency*, *shaky logic*, *weak encapsulation*, *least privilege violation*, and *missing privilege*. We showed how these weaknesses revealed 4, previously unknown, security flaws in the investigated systems.

In section 15.2, we presented a motivating example from Joomla! showing how security weaknesses can hinder the comprehension and usability of a system. In that specific case, we showed how a seemingly simple task, deleting a user group, could require a counter-intuitive combination of privileges and lead to multiple least privilege violations.

In Figure 15.2, we presented the ratios of files and lines of code that belong to some security-discordant clone clusters. Since, in the context of this study, we only reviewed files and LOC that belonged to a security-discordant clone cluster, Figure 15.2 also reflects the reduction in code review effort when using our approach. Approximately 4% of Joomla! code base was reviewed and that ratio drops to only 2% for Moodle. Two days of work were required to perform code reviews of the security-discordant clones in both systems.

Table 15.3: Numbers of novel and known access control flaws that were revealed by security-discordant clone clusters.

Application	Novel flaws	Known flaws	
		Detected	Missed
Joomla! 2.5.4	2	1	0
Moodle 2.3.2	2	4	2

In the context of Joomla!, our analysis revealed two novel access control flaws. Considering that only one other access control flaw had previously been reported at the time of the analysis, it means that a review of 4% of the code base yielded a 200% increase of the number of reported access control flaws. In the context of Moodle, reviewing 2% of the code base yielded a 33% increase of the number of reported access control flaws. Considering that these are very popular and widely used systems both with dedicated security teams, we are very satisfied with these results.

In Figure 15.3, we compared the distributions of security weaknesses among the investigated applications. As we underlined in section 15.5, two obvious differences can be observed between both systems. First, no least privilege violations were identified in Moodle while they form a little more than 25% of the weaknesses in Joomla!. This is due to different implementation choices between both systems. In Joomla!, each component is protected by a limited number of privileges: create, delete, configure, admin, edit and edit.state. However, since no security specification or policy are available, the choice of what privilege to enforce in specific cases is left to developers. As a consequence, similar actions in different components end up being protected by privileges located at different levels in the privilege hierarchy, inducing least privilege violations.

Moodle, on the other hand, is located at the opposite side of the privilege spectrum: each component has its own set of privileges and each privilege is specifically designed to protect one specific action. Since there practically is a one-to-one mapping between privileges and actions, least privilege violations are rare. This implementation choice, however, has consequences on the overall complexity of the access control model: as the number of features grows, so does the number of privileges. Indeed, we observed a steady growth of the number of privileges in the last versions of Moodle.

Weak encapsulation cases also greatly differ between Joomla! and Moodle. This is explained by the fact that Moodle uses a larger number of administrator libraries that relies on external checks than Joomla!. A higher proportion of security-sensitive functions are thus weakly encapsulated in privileged portions of the Moodle system. Weak encapsulation can be a problem in systems like Joomla! and Moodle, where third-party plugins are supported.

Weak encapsulation indeed relies on non-written rules and developers' discipline to enforce security in the system. Third-party plugin developers might not be aware of such non-written rules and can easily break the access control model by inadvertently calling a weakly encapsulated function without performing the expected access check first.

Figures 15.4(a) and 15.4(b) show the distribution of security weaknesses (true positives) as well as legitimate and utility functions clusters (false positives) across both systems. Interestingly, these figures show that 56% of security-discordant clusters in Joomla! and 16%

security-discordant clusters in Moodle revealed security weaknesses and flaws, for a difference of 40% between the two systems. We attribute this important difference mainly to one factor: defensive programming.

In the context of access control models, defensive programming consists in performing access control checks sooner than later. In other words, given a PHP script that performs security-sensitive operations, defensive programmers will immediately perform an access check and redirect the user to an error page if it fails.

In a defensive programming paradigm, privileges therefore often protect far more code than is strictly necessary for security purposes. While, on the one hand, we are totally in favor of defensive programming, on the other hand such a practice significantly increases the number of false positives. Indeed, several cloned fragments that do not perform security-sensitive operations in Moodle are legitimately protected by different privileges.

In the context of this study, we investigated how security-discordant clones could reveal access control weaknesses and flaws in the absence of security specifications. Results shown how, without any input from developers, our approach can drastically reduce code review efforts and reveal novel security flaws. In its current form, however, our approach suffers from high false positive rates. A simple while convenient solution to this problem might be to ask developers to supply a "white list" of files and functions that are known not to perform security-sensitive operations. From our observations, we would expect false positive rates to decrease significantly.

The false negative rate, on the other hand, is a lot harder to estimate as it would imply knowing the total number of weaknesses and flaws (known and residual) in a system. In the context of our study, we can only estimate the false negative rate based on known flaws. Table 15.3 shows that our approach has an estimated false negative rate of 0% for Joomla! and 25% for Moodle. Overall, the estimated false negative rates suggest that, in practice, our approach is suitable for uncovering access control flaws.

15.7 Related work

15.7.1 Clone detection

Clone detection state of the art includes different techniques. For type-1 (identical) and type-2 (parametric) clones, AST-based detection has been introduced in [22]. Other detection methods for type-1 and type-2 clones include metrics-based clone detection as in [112], suffix tree-based clone detection as in [65], and string matching clone detection as in [47]. For a detailed survey of clone detection techniques, a good portrait is provided in [141].

Any of these clone detectors could have been used in the clone detection step of our analysis

and are expected to produce similar results. However, all of them would need to be adapted to the PHP language.
Using clones to detect errors and bugs is not a new idea. Recent work by [103], [175] and [86] extensively used clones to detect inconsistencies in programs and suggest potential bugs. In all cases, the clone detector leverages information from a defective piece of code to detect latent bugs. The main limitation of these approaches lies in the fact that a defective chunk of code is required *a priori*. Our approach works without such knowledge as it relies on inferred security specifications and privilege granting patterns extracted from the source code. From a security point of view, our approach has the advantage of detecting security flaws *before* a "day zero attack".
Interesting empirical work about the use of clones for security tasks has also been conducted by Dang et al. [42]. Of particular interest is their visualization interface that allows developers to explore and review code clones in a convivial manner.

15.7.2 Security analysis of access control models

Security analysis of access control models at the source code level is an emerging field and few research papers have been dedicated to the subject. One of the main challenge for security analysis of access control models is to compensate for the lack of specifications. Prior to our work, four main strategies have been explored: structural and behavioral based, security-sensitive statements based, hyperlink based, dynamic analysis based specification inference.
In recent work, Alalfi et al. proposed an approach based on structural information and dynamically recovered behavioral models to recover access control models from PHP applications [11]. Based on the recovered access control model, they design test scenarios that exercise typical access control flaws [5].
In 2011, Son et al. introduced RoleCast [152], a data-flow based approach that infer access control checks based on the values that are typically checked prior to executing security-sensitive statements. Security flaws are reported when some unprotected execution paths can reach security-sensitive statements.
Access control specifications have also been inferred based on the protection level of hyperlinks [156**?**]. The assumption behind these studies is the following: if all the hyperlinks that point to a page are privileged, direct URL access to the page should be denied. Otherwise, a *forced browsing* vulnerability is reported.
Other researchers also developed approaches to learn access control specifications based on dynamic analysis [19, 20]. Applying data-mining techniques to access logs that are collected over a certain period of time, their approach identifies access control misconfigurations, that are small differences between the privileges of a group of users.

On a similar line of thoughts, Das et al. developed the Baaz tool [45] to automatically detect misconfigurations in access control models. Contrary to the approach by Bauer et al. that is based on dynamic analysis, Baaz detects misconfigurations based on static access control policies.

In [57], authors proposed an approach to mine security-sensitive operations from legacy code using a mix of static and dynamic analysis. They first extract static code patterns and dynamic side-effects of operations that are known to be security-sensitive. Both static code patterns and dynamic side-effect constitute the fingerprint of an operation. Then, their tool automatically mitigate security flaws by protecting operations that share similar fingerprints with security-sensitive operations.

What distinguishes our work from all these studies mainly lies in two crucial points. First, our approach requires very little pre-requisites. Apart from the identification of access control patterns and implementation of routines to detect them, our tool is fully automated. Other approaches require developers to supply extensive access control policy specifications, or run an instrumented version of the application for weeks to collect access logs.

Second, our tool is agnostic to the *types* of access control flaws to be detected. Where other approaches make assumption about security-sensitive statements or specialize in the detection of a specific type of access control flaws, our tool simply reports security discrepancies at the implementation level. This flexibility allowed us to identify several types of flaws: information leaks, privilege escalation attacks and improper authorization cases.

15.8 Conclusion

In this exploratory study, we investigated security-discordant clone clusters, that are clusters of syntactically similar fragments of code which differ from a protection perspective. Our goal was two-fold: to characterize security-discordant clone clusters and to assess whether they might be symptomatic of security weaknesses and flaws. The characterization step revealed five recurring and application-independent categories of security-discordant clone clusters. Further investigation revealed that security-discordant clone clusters are often symptomatic of security weaknesses and flaws. We showed how some of these weaknesses can hinder the comprehension and usability of an access control model and induce security flaws. Investigation of security-discordant clone clusters indeed revealed four novel security flaws in Joomla! and Moodle.

The strength of our approach lies in the power of combining two orthogonal components: syntactic information from code clones and static security information from SPT analysis. Based on our promising results, we plan to investigate additional systems and application

domains and extract invariants for each category of weaknesses and flaws. Category-wise invariants will help automate the classification of security-discordant clone clusters and further simplify analysis of the results.

15.9 Acknowledgements

The authors thank Jean-Francois Le for his work on security granting patterns in Joomla!. This research has been funded by the Natural Sciences and Engineering Research Council of Canada (NSERC) under the Discovery Grants Program.

CHAPTER 16

CORRECTIONS FOR PAPER 6

The previous chapter contains the published version of the paper entitled: "Uncovering Access Control Weaknesses and Flaws with Security-Discordant Software Clones". The current chapter addresses comments and corrections by the jury regarding this paper.

In section 15.5, we presented shaky logic clones and defined them as: "fragments that intermix privilege checks with external functions that are not related to access control". A member of the jury rightly observed that software clones are not part of this definition and asked how were clones necessary to detect such cases. The answer is simple. In the context of this study, software clones were used as a way to highlight discordances in the protection level of syntactically similar fragments of code. However, the vulnerabilities that are detected are not necessarily *related* to software clones. In other words, shaky logic vulnerabilities were *revealed* through the use of security-discordant clones.

A limitation of the current approach is that shaky logic vulnerabilities are only reported if they belong to some security-discordant clone cluster. Now that we revealed the existence of shaky logic vulnerabilities, it would be interesting to design approaches that are specifically tailored for their detection. Doing so would enable us to detect shaky logic vulnerabilities that do not necessarily belong to some security-discordant clone cluster.

CHAPTER 17

GENERAL DISCUSSION

In this thesis, we adopted a reverse-engineering approach to the problem of investigating and detecting vulnerabilities in access control models. Chapters 5 and 7 detailed our technique to extract privilege protection from source code. Chapter 9 showed how Formal Concept Analysis can help developers detect design flaws and/or refactoring opportunities in their RBAC and privilege protection models. Finally, chapters 11 to 15 presented three different approaches for the detection of vulnerabilities in access control models.
Each of these topics has been extensively discussed in their respective chapters and the goal of this section is not to repeat these discussions here. Instead, we will discuss how the presented work can be integrated into the software development cycle and suggest potential lines of research to enhance application security in general.

17.1 Secure software development

Securing software is a notoriously difficult task. Security spans all aspects of software development, from design to deployment. In this section, we put the presented work in the perspective of a global secure software development approach and show how it can be integrated to the software development cycle.
In [113], McGraw et al. suggest seven software security best practices, called *touchpoints* and show how various touchpoints apply to various software artifacts, that are produced at each step of the software development cycle. Figure 17.1 summarizes their idea. In the figure, the software development cycle is illustrated with light gray arrows, artifacts are represented as boxes and touchpoints, that are numbered from 1 to 7, map to specific artifacts. According to McGraw, the seven touchpoints are:

1. Code review (tools)
2. Architectural risk analysis
3. Penetration testing
4. Risk-based security tests
5. Abuse cases

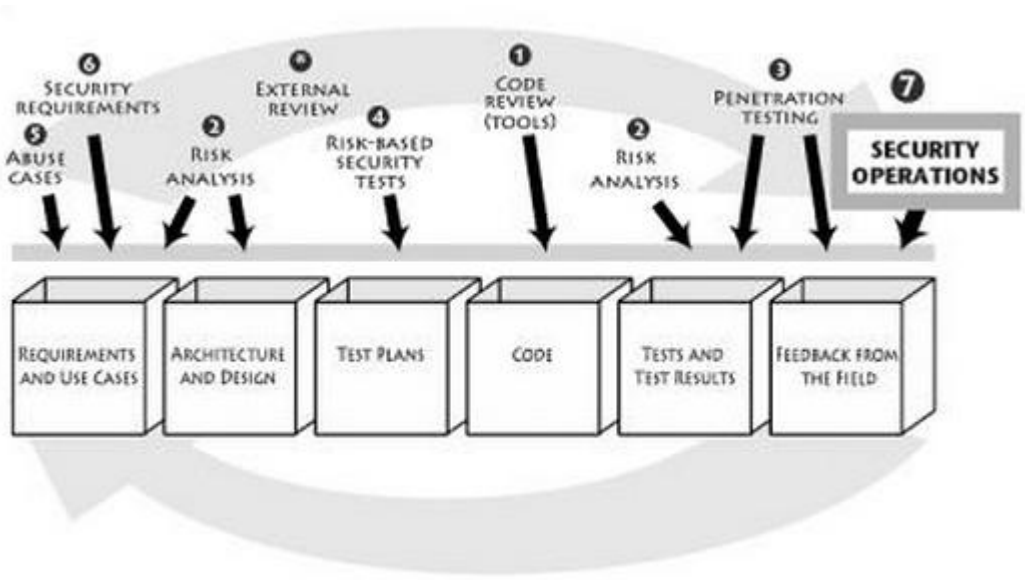


Figure 17.1: This figure was taken from Software Security: "Building Security In" by McGraw et al.. It shows how security touchpoints can be applied to various software artifacts.

6. Security requirements

7. Security operations

In the context of this discussion touchpoints will help us illustrate how the presented work can be integrated in a secure software development process.
According to McGraw et al., code review aims at finding bugs while architectural risk analysis aims at identifying flaws. They indeed make a distinction between bugs, defined as an "implementation-level software problem" and flaws, that are "problems at a deeper level". For example, according to their definitions, buffer overflow, SQLi, and XSS attacks stem from bugs while bad error handling and broken access controls are flaws. Penetration testing is the process by which an external entity, the penetration tester, plays the role of an attacker trying to break into the system. Risk-based security testing is very similar to penetration testing, with the exception that it occurs earlier in the software development process. While penetration testing operates on a functional system, risk-based testing can take place at the unit test level. Abuse cases and security requirements are the two sides of the same coin. On the one hand, abuse cases work from the outside in, describing attack scenarios against the system. On the other hand, security requirements, work from the inside out, describing how assets should be protected from external attacks. Finally, security operations encompass the security mechanisms that are put in place once the system is deployed.
Interestingly, while the work we presented in chapters 5 to 15 obviously works at the source

code level (code review touchpoint), its aim is to identify flaws in access control models (architectural risk analysis), often in the absence of security specifications (security requirements). This might seem counter-intuitive at first sight. Traditional approaches to software development (e.g. waterfall model [142] or spiral development [27]) strongly encourage forward-engineering approaches where requirements drive architectural design that in turn drive how the code is written, leaving little to no space to reverse-engineering approaches. In the last decade, however, strong arguments have been raised in favor of more flexible software development processes, often tagged as *agile* development methods [109], that favor shorter development cycles and encourage refinement of software design and architecture based on code reviews, test cases and feedback from the field. Examples of agile development processes include Scrum [146], eXtreme Programming [23], Feature-Driven Development [132] and others. The presented work is clearly more suited for these modern software development processes.

It is important to clarify, however, that we do not advocate against traditional forward-engineering approaches. Although much of the work that was presented in this thesis works in the absence of security requirements or architectural design documents, we strongly argue that it is advisable to take security into account at *every* step of the software development process. In fact, much of the work presented here would benefit from security requirements or architectural design documents. Indeed, given that such documents are available, the techniques presented in chapters 5 to 9, that reverse-engineer access control models from source code, could be adapted to verify the conformance between access control models and their implementations and detect even more vulnerabilities. Furthermore, the techniques presented in chapters 11 to 15 could be used highlight wrong or missing security requirements as well as weak design decisions that hinder the security of Web applications.

17.2 Improving application security

Tools that automatically detect security flaws are great and useful to the whole software community. However, a major drawback of vulnerability detection tools is that they can induce a false sense of security. Indeed, the fact that a code review tool does not find any vulnerability sometimes get wrongly interpreted as a certification that the application does not contain any more flaws.

As security researchers, our goal is not only to release tools to patch vulnerable applications, it is also to encourage practices that will help build secure applications right from the start. Similarly to the authors of [113], who coined the expression “building security in”, we strongly advocate that security is a concern that should be addressed at all the stages of the software

development process, just like scalability, user experience, performance, testability, and so on. We are deeply convinced that *building security in* has the potential to significantly improve the security of applications in general and, more importantly, provide better protection to end-users.

Building security in, however, requires deep cultural changes in the software community. In the following paragraphs, we discuss several avenues of research that could be investigated to attain this goal.

17.2.1 Education

Traditionally, computer security and software engineering were considered distinct disciplines. On the one hand, computer security experts were in charge of developing security policies, cryptographic algorithms, trust models, etc. On the other hand, software engineers were in charge of building software that is scalable, reliable, maintainable, usable, etc. Consequently, application security, that lies at the intersection between the two disciplines, received very little attention from both communities, resulting in applications that are plagued with security vulnerabilities that must be patched post-release.

In our opinion, the first step to produce more secure applications is to ensure that software developers receive better computer security training. Over the years, we had the occasion to exchange with several open-source software developers about our security findings. Apart from a few exceptions, we were astonished to observe how many software developers have little to no notions of application security. Fortunately, more and more computer science/engineering programs include computer security courses in their curriculum. We are deeply convinced that the prevalence of several simple security flaws will drastically diminish as more and more developers receive basic training in computer security.

17.2.2 Liability

Liability for software security is a topic that attracted a lot of attention in recent years, both from the software and legal communities. Traditionally, software companies have protected themselves from liability lawsuits by selling customers a license to use their software rather than the actual product and by requiring customers to sign lengthy and complicated End-User License Agreements (EULA) [39]. As a result, software companies often do not consider security as a top priority and release applications that still contain several vulnerabilities. Interestingly, some researchers explored how liability policies might encourage software companies in investing more time and money to better secure their software [16].

17.2.3 Secure coding standards

Software developers are usually accustomed to *coding standards* that define naming conventions, comment policies and general code style guidelines. Similarly, secure coding standards define best coding practices aimed at reducing the number of vulnerabilities in an application. In our opinion, enforcing secure coding standards is the easiest way to introduce software developers to application security and increase general security awareness in the software community. In the context of Web applications, a good starting point is the Open Web Application Security Project (OWASP) Secure Coding Practices document [130].

CHAPTER 18

CONCLUSION AND FUTURE WORK

In this thesis, we tackled the problems of reverse-engineering RBAC and privilege protection models, analyzing them and automatically detecting access control flaws. In chapters 5 and 7 we presented a reverse-engineering approach that extracts privilege protection from source code. Specifically, in chapter 5, we presented the Security Pattern Traversal analysis (SPT), that extracts the privilege protection model of an application. We further showed how SPT could help locate sensitive parts of an application and reason about the relative importance of privileges, based on the number of statements they protect.

SPT analysis drastically differs from related approaches. Typical inter-procedural program analysis approaches either sacrifice performance in favor of precision (context and flow-sensitive approaches) or precision in favor of performance (context and flow-insensitive approaches). SPT analysis, on the other hand, sacrifices neither precision nor performance. By focusing on a specific class of problems, that is the propagation of pattern-based security properties, we were able to develop an analysis that is fast, scalable and precise. SPT analysis also served as the cornerstone over which the rest of this work was built.

In chapter 7, we presented an extension to SPT that allows to track the inter-procedural propagation of privilege checks through variables and parameters. We showed how this extension could significantly increase the recall of SPT without hindering its precision. We also discussed how the same algorithms can be used to resolve dynamic include statements and improve the precision of call graphs in PHP applications.

Static analysis of highly dynamic languages is extremely difficult. However, increasing amounts of research tend to show that it is possible to achieve good precision/performance trade-offs in static analysis of dynamic languages through careful approximations [107, 50, 170]. The analysis we presented in chapter 7 follows this line of thinking. We observed how security patterns are used and propagated *in practice* and devised an algorithm that captured these usage patterns. Results showed how well our algorithm performed with precision rates between 96% and 100%.

Chapter 9 was dedicated to the investigation of RBAC models with formal concept analysis (FCA). In summary, we showed how FCA could reveal the implicit role hierarchy of an access control model and highlight the impacts of modifying the RBAC model. We also showed how FCA of the privilege protection model could reveal implications between privileges that are invisible to administrators and that can induce misunderstandings and vulnerabilities.

Access control model specifications or documentation are rarely available in the context of open source Web applications and this work aimed at helping developers and administrators getting a portrait of the implemented access control model. We strongly believe that empowering administrators and developers with relevant security information has the potential to drive down the number of security flaws in a system. This reason alone illustrates the relevance of this work. If formal access control specifications are available, however, then this work is even more relevant. Testing the conformance between access control models as *specified* and as *implemented* has the potential to uncover even more security flaws.

Chapters 11 to 15 presented three different strategies for the identification of access control vulnerabilities in Web applications.

Chapter 11 was dedicated to the detection of forced browsing vulnerabilities, where an attacker directly accesses the URL of an unprotected resource. We first showed how we could use SPT to *infer* the privileged resources of a system through the investigation of hyperlinks, the idea being that if all the hyperlinks that point to a resource are privileged, the resource is also privileged. We then showed how SPT statically computes the resources that are protected at runtime and reports forced browsing vulnerabilities for all resources that were *inferred* as privileged but that are not protected at runtime. A comparison with previous work, that was based on string analysis, revealed how this approach achieves equivalent precision levels with accelerations up to 890× faster.

Forced browsing vulnerabilities, also referred to as "Insecure Direct Object References" by the OWASP, are among the most prevalent flaws in Web applications. While dynamic approaches such as penetration testing have the potential to uncover these kind of flaws quite easily, we strongly argue that they should be used in conjunction with static approaches, such as the work presented in chapter 11. Indeed, in a security context any flaw is a flaw too many and dynamic approaches, by their very nature, usually only exercise a subset of system, potentially missing some flaws. Static approaches, on the other hand, typically consider every possible execution paths and more easily detect flaws in parts of an application that can otherwise be difficult to exercise dynamically. Moreover, given the very limited amount of manual work that is required, the significant speed gains and the high precision and recall levels that were achieved, we see no significant limitation to the adoption of this approach in a real-world context.

In chapter 13, we presented a strategy for the identification of access control flaws that is based on information retrieval techniques. In this work, we tackled the problem of identifying semantically wrong privilege checks, where the privilege is semantically unrelated to the action it protects. We showed how this approach was able to identify several semantic smells and errors in the investigated system. Detecting semantically wrong privilege checks,

however, is not an easy task. In order to detect wrong privilege checks, one must have an understanding of the underlying security specifications, which are rarely documented. We showed how we could partly overcome this difficulty by leveraging a natural language processing technique called Latent Dirichlet Allocation.

To our knowledge, we were the first to tackle the problem of identifying semantically *wrong* privilege checks. Most researchers indeed strive to identify *missing* privilege checks. While searching for missing privilege checks is very relevant from a security point of view, we strongly argue that searching for wrong privilege checks is at least as relevant. Wrong privilege checks indeed give rise to a lot more subtle but no less dangerous kind of security flaw. For example, if a privilege check wrongly verifies a privilege that is too low in the privilege hierarchy, it opens the door to privilege escalation attacks.

Finally, in chapter 15, we investigated yet another strategy for the identification of access control vulnerabilities that is based on clone analysis. In this work, we explored the idea that code clones that are protected in different ways, called security-discordant clones, might be symptomatic of access control vulnerabilities. We hypothesized that code clones usually perform similar operations in a system and that clones should thus be protected by similar privileges. Investigation of security-discordant clones indeed revealed several security flaws in the systems under study.

This work drastically differs from other clone-based vulnerability detection techniques in the sense that, contrary to previous approaches, our technique can detect day zero vulnerabilities. Indeed, clone based approaches for vulnerability detection typically start from a vulnerability report or a patch and try to identify pieces of code that are similar to the buggy or patched code. Our approach instead leverages discrepancies in the protection level of code clones to identify vulnerabilities, without *a priori* information.

Overall, this thesis focused on the development of static analysis approaches for the investigation of access control models and the identification of security vulnerabilities. Several research paths however remain unexplored.

18.1 Future work

Mixed static/dynamic analysis: Web applications are dynamic by nature and constructs like sessions, cookies and dynamic features of languages like PHP or JavaScript are very difficult to analyze statically. It would be interesting to investigate how dynamic analysis approaches, such as those presented in Alalfi et al. [7, 9, 10], could complement our work.

Model-based secure engineering: Model-based engineering approaches have been proposed in the literature for the design and deployment of access control models. I would like

to investigate how existing ad-hoc access control models can be adapted to this paradigm in order to facilitate their maintenance and refactoring. Investigating the evolution of security models in Web applications also seems a promising line of research.

Security testing: Security testing presents particular challenges that are not addressed by "classic" testing approaches. For example, classic coverage metrics are not well suited to measure the coverage of access control models, as underlined in Alalfi et al. [10]. It would be interesting to explore how static and dynamic analysis approaches (white-box testing) and model-based approaches (model-based testing) could be used to define novel security metrics and enhance testing of access control models. Investigating how existing test suites could be re-used and extended to test access control models also seems a promising research avenue.

Access control models in the perspective of semantic Web: As mentioned earlier, privileges in access control models are expected to be semantically related to the action they protect. However, this semantic relation is sometimes broken, giving rise to semantic errors. I would like investigate how semantic Web technologies (RDF, OWL, etc.) could help define and investigate security ontologies to identify semantic errors in access control models.

BIBLIOGRAPHY

[1] (2004). Information Technology - Role Based Access Control . Rapport technique 359-2004, ANSI and INCITS.

[2] AGRAWAL, H. et HORGAN, J. R. (1990). Dynamic program slicing. *ACM SIGPLAN Notices*, 25, 246–256.

[3] AHO, A. V., LAM, M. S., SETHI, R. et ULLMAN, J. D. (2007). Compilers: Principles, techniques, & tools with gradiance.

[4] ALALFI, M., CORDY, J. et DEAN, T. (2009). A verification framework for access control in dynamic web applications. *Proceedings of the 2nd Canadian Conference on Computer Science and Software Engineering*. ACM, 109–113.

[5] ALALFI, M., CORDY, J. R. et DEAN, T. R. (2012). Automated verification of role-based access control security models recovered from dynamic web applications. *Proceedings of the 14th International Symposium on Web Systems Evolution*. IEEE Computer Society, 1–10.

[6] ALALFI, M. H., CORDY, J. R. et DEAN, T. R. (2008). SQL2XMI: Reverse engineering of UML-ER diagrams from relational database schemas. *Proceedings of the 15th Working Conference on Reverse Engineering*. IEEE Computer Society, 187–191.

[7] ALALFI, M. H., CORDY, J. R. et DEAN, T. R. (2009). Automated reverse engineering of UML sequence diagrams for dynamic web applications. *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation*. IEEE Computer Society, 287–294.

[8] ALALFI, M. H., CORDY, J. R. et DEAN, T. R. (2009). Modelling methods for web application verification and testing: state of the art. *Software Testing, Verification and Reliability*, 19, 265–296.

[9] ALALFI, M. H., CORDY, J. R. et DEAN, T. R. (2009). WAFA: Fine-grained dynamic analysis of web applications. *Proceedings of the 11th IEEE International Symposium on Web Systems Evolution*. IEEE Computer Society, 141–150.

[10] ALALFI, M. H., CORDY, J. R. et DEAN, T. R. (2010). Automating coverage metrics for dynamic web applications. *Proceedings of the 14th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 51–60.

[11] ALALFI, M. H., CORDY, J. R. et DEAN, T. R. (2012). Recovering role-based access control security models from dynamic web applications. *Web Engineering*, Springer. 121–136.

[12] ALSTRUP, S., HAREL, D., LAURIDSEN, P. W. et THORUP, M. (1999). Dominators in linear time. *SIAM Journal on Computing*, 28, 2117–2132.

[13] ANDERSON, R., BARTON, C., BÖHME, R., CLAYTON, R., VAN EETEN, M. J., LEVI, M., MOORE, T. et SAVAGE, S. (2013). Measuring the cost of cybercrime. *The Economics of Information Security and Privacy*, Springer. 265–300.

[14] ARTZI, S., KIEZUN, A., DOLBY, J., TIP, F., DIG, D., PARADKAR, A. et ERNST, M. (2010). Finding bugs in web applications using dynamic test generation and explicit state model checking. *IEEE Transactions on Software Engineering*.

[15] ASUNCION, H., ASUNCION, A. et TAYLOR, R. (2010). Software traceability with topic modeling. *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*. ACM, 95–104.

[16] AUGUST, T. et TUNCA, T. I. (2011). Who should be responsible for software security? a comparative analysis of liability policies in network environments. *Management Science*, 57, 934–959.

[17] BALL, T., MAJUMDAR, R., MILLSTEIN, T. et RAJAMANI, S. K. (2001). Automatic predicate abstraction of C programs. *ACM SIGPLAN Notices*. ACM, vol. 36, 203–213.

[18] BALL, T. et RAJAMANI, S. K. (2000). Bebop: A symbolic model checker for boolean programs. *SPIN Model Checking and Software Verification*, Springer. 113–130.

[19] BAUER, L., GARRISS, S. et REITER, M. K. (2011). Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)*, 14, 2:1–2:28.

[20] BAUER, L., LIANG, Y., REITER, M. K. et SPENSKY, C. (2012). Discovering access-control misconfigurations: New approaches and evaluation methodologies. *Proceedings of the second ACM Conference on Data and Application Security and Privacy*. ACM, 95–104.

[21] BAUMGRASS, A., BAIER, T., MENDLING, J. et STREMBECK, M. (2012). Conformance checking of RBAC policies in process-aware information systems. *Business*

Process Management Workshops, Springer Berlin Heidelberg, vol. 100 de *Lecture Notes in Business Information Processing*. 435–446.

[22] BAXTER, I., YAHIN, A., MOURA, L., SANT'ANNA, M. et BIER, L. (1998). Clone detection using abstract syntax trees. *Proceedings of the 1998 International Conference on Software Maintenance*. IEEE, 368–377.

[23] BECK, K. (1999). Embracing change with extreme programming. *Computer*, 32, 70–77.

[24] BERNDL, M., LHOTÁK, O., QIAN, F., HENDREN, L. et UMANEE, N. (2003). Points-to analysis using BDDs. *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 38, 103–114.

[25] BINKLEY, D. et HARMAN, M. (2004). A survey of empirical results on program slicing. *Advances in Computers*, 62, 105–178.

[26] BLEI, D., NG, A. et JORDAN, M. (2003). Latent dirichlet allocation. *Journal of machine Learning research*, 3, 993–1022.

[27] BOEHM, B. W. (1988). A spiral model of software development and enhancement. *Computer*, 21, 61–72.

[28] BOLLIG, B. et WEGENER, I. (1996). Improving the variable ordering of OBDDs is NP-complete. *Computers, IEEE Transactions on*, 45, 993–1002.

[29] BRAVENBOER, M. et SMARAGDAKIS, Y. (2009). Exception analysis and points-to analysis: Better together. *Proceedings of the 18th International Symposium on Software testing and analysis*. ACM, 1–12.

[30] CHANDRA, A. et HAREL, D. (1985). Horn clause queries and generalizations. *The Journal of Logic Programming*, 2, 1–15.

[31] CHEN, H. et WAGNER, D. (2002). MOPS: an infrastructure for examining security properties of software. *Proceedings of the 9th ACM Conference on Computer and communications security*. ACM, 235–244.

[32] CHRISTENSEN, A., MøLLER, A. et SCHWARTZBACH, M. (2003). Precise analysis of string expressions. *Static Analysis*, 1076–1076.

[33] CLARKE, E. (1997). Model checking. *Foundations of Software Technology and Theoretical Computer Science*, Springer Berlin / Heidelberg, vol. 1346 de *Lecture Notes in Computer Science*. 54–56.

[34] CLAUSE, J., LI, W. et ORSO, A. (2007). Dytan: a generic dynamic taint analysis framework. *Proceedings of the 2007 International Symposium on Software Testing and Analysis*. ACM, 196–206.

[35] COLANTONIO, A., DI PIETRO, R. et OCELLO, A. (2008). A cost-driven approach to role engineering. *Proceedings of the 2008 ACM Symposium on Applied computing*. ACM, 2129–2136.

[36] CORDY, J. R., HALPERN-HAMU, C. D. et PROMISLOW, E. (1991). TXL: A rapid prototyping system for programming language dialects. *Computer Languages*, 16, 97–107.

[37] CORMEN, T. H., STEIN, C., RIVEST, R. L. et LEISERSON, C. E. (2009). *Introduction to Algorithms*. McGraw-Hill Higher Education, troisième édition.

[38] COVA, M., BALZAROTTI, D., FELMETSGER, V. et VIGNA, G. (2007). Swaddler: An approach for the anomaly-based detection of state violations in web applications. *Recent Advances in Intrusion Detection*. 63–86.

[39] CUSUMANO, M. A. (2004). Who is liable for bugs and security flaws in software? *Communications of the ACM*, 47, 25–27.

[40] CWE/SANS (2011). CWE/SANS top 25 most dangerous software errors. `http://cwe.mitre.org/top25`.

[41] DALTON, M., KOZYRAKIS, C. et ZELDOVICH, N. (2009). Nemesis: preventing authentication & access control vulnerabilities in web applications. *USENIX Security Symposium*. 267–282.

[42] DANG, Y., ZHANG, D., GE, S., CHU, C., QIU, Y. et XIE, T. (2012). XIAO: tuning code clones at hands of engineers in practice. *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 369–378.

[43] DANIAL, A. (2012). Count lines of code. `http://cloc.sourceforge.net/`.

[44] DAS, M. (2000). Unification-based pointer analysis with directional assignments. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*. ACM, vol. 35, 35–46.

[45] DAS, T., BHAGWAN, R. et NALDURG, P. (2010). Baaz: a system for detecting access control misconfigurations. *USENIX Security Symposium*. 161–176.

[46] DAU, F. et KNECHTEL, M. (2009). Access policy design supported by FCA methods. *Conceptual Structures: Leveraging Semantic Technologies*. Springer, 141–154.

[47] DUCASSE, S., NIERSTRASZ, O. et RIEGER, M. (2006). On the effectiveness of clone detection by string matching. *International Journal on Software Maintenance and Evolution: Research and Practice*, 18, 37–58.

[48] DWYER, M. B., HATCLIFF, J., JOEHANES, R., LAUBACH, S., PĂSĂREANU, C. S., ZHENG, H. et VISSER, W. (2001). Tool-supported program abstraction for finite-state verification. *Proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 177–187.

[49] ESPARZA, J. et SCHWOON, S. (2001). A BDD-based model checker for recursive programs. *Computer Aided Verification*. Springer, 324–336.

[50] FELDTHAUS, A., SCHAFER, M., SRIDHARAN, M., DOLBY, J. et TIP, F. (2013). Efficient construction of approximate call graphs for javascript ide services. *Proceedings of the 35th International Conference on Software Engineering*. IEEE Computer Society, 752–761.

[51] FELMETSGER, V., CAVEDON, L., KRUEGEL, C. et VIGNA, G. (2010). Toward automated detection of logic vulnerabilities in web applications. *USENIX Security Symposium*. 143–160.

[52] FISLER, K., KRISHNAMURTHI, S., MEYEROVICH, L. A. et TSCHANTZ, M. C. (2005). Verification and change-impact analysis of access-control policies. *Proceedings of the 27th International Conference on Software Engineering*. ACM, 196–205.

[53] FOWLER, M. et BECK, K. (1999). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

[54] GALLAGHER, K. B. et LYLE, J. R. (1991). Using program slicing in software maintenance. *IEEE Transactions on Software Engineering*, 17, 751–761.

[55] GAMMA, E., HELM, R., JOHNSON, R. et VLISSIDES, J. (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[56] GAMMA, E., RICHARD, H., RALPH, J. et JOHN, V. (2007). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

[57] GANAPATHY, V., JAEGER, T. et JHA, S. (2006). Retrofitting legacy code for authorization policy enforcement. *Proceedings of the 2006 Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 15 pp.–229.

[58] GANAPATHY, V., KING, D., JAEGER, T. et JHA, S. (2007). Mining security-sensitive operations in legacy code using concept analysis. *Proceedings of the 29th International Conference on Software Engineering*. IEEE/ACM, 458–467.

[59] GANTER, B., WILLE, R. et WILLE, R. (1999). *Formal concept analysis*. Springer Berlin.

[60] GAUTHIER, F., LETARTE, D., LAVOIE, T. et MERLO, E. (2011). Extraction and comprehension of Moodle's access control model: A case study. *Proceedings of the 9th Annual International Conference on Privacy, Security and Trust*. IEEE Computer Society, 44–51.

[61] GAUTHIER, F. et MERLO, E. (2012). Alias-aware propagation of simple pattern-based properties in PHP applications. *Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE Computer Society.

[62] GAUTHIER, F. et MERLO, E. (2012). Fast detection of access control vulnerabilities in PHP applications. *Proceedings of the 19th Working Conference on Reverse Engineering*. IEEE Computer Society, 247–256.

[63] GAUTHIER, F. et MERLO, E. (2013). Semantic smells and errors in access control models: A case study in PHP. *Proceedings of the 35th International Conference on Software Engineering*. IEEE/ACM.

[64] GEORGIADIS, L. et TARJAN, R. E. (2004). Finding dominators revisited. *Proceedings of the fifteenth annual ACM-SIAM Symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 869–878.

[65] GÖDE, N. et KOSCHKE, R. (2009). Incremental clone detection. *Proceedings of the 13th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 219–228.

[66] GODEFROID, P., NORI, A., RAJAMANI, S. et TETALI, S. (2010). Compositional may-must program analysis: unleashing the power of alternation. *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, vol. 45, 43–56.

[67] GODIN, R. et VALTCHEV, P. (2005). Formal concept analysis-based class hierarchy design in object-oriented software development. *Formal Concept Analysis*, Springer. 304–323.

[68] GOPAL, R. (1991). Dynamic program slicing based on dependence relations. *Proceedings of the Conference on Software Maintenance*. IEEE Computer Society, 191–200.

[69] GRANT, S., CORDY, J. et SKILLICORN, D. (2011). Reverse engineering co-maintenance relationships using conceptual analysis of source code. *Reverse Engineering, 2011. WCRE'11. 18th Working Conference on*. IEEE Computer Society, 87–91.

[70] GUELEV, D. P., RYAN, M. et SCHOBBENS, P. Y. (2004). Model-checking access control policies. *Information Security*, Springer. 219–230.

[71] GUIGUES, J. et DUQUENNE, V. (1986). Familles minimales d'implications informatives résultant d'un tableau de données binaires. *Math. Sci. Humaines*, 95, 5–18.

[72] HALFOND, W. et ORSO, A. (2005). AMNESIA: analysis and monitoring for NEutralizing SQL-injection attacks. *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 174–183.

[73] HALLÉ, S., ETTEMA, T., BUNCH, C. et BULTAN, T. (2010). Eliminating navigation errors in web applications via model checking and runtime enforcement of navigation state machines. *Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 235–244.

[74] HAMMER, C., KRINKE, J. et SNELTING, G. (2006). Information flow control for java based on path conditions in dependence graphs. *IEEE International Symposium on Secure Software Engineering*. 87–96.

[75] HAMMER, C., SCHAADE, R. et SNELTING, G. (2008). Static path conditions for java. *Proceedings of the third ACM SIGPLAN Workshop on Programming languages and analysis for security*. ACM, 57–66.

[76] HANSEN, F. et OLESHCHUK, V. (2005). Conformance checking of RBAC policy and its implementation. *Information Security Practice and Experience*. Springer, 144–155.

[77] HARDEKOPF, B. et LIN, C. (2009). Semi-sparse flow-sensitive pointer analysis. *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, vol. 44, 226–238.

[78] HAREL, D. (1985). A linear algorithm for finding dominators in flow graphs and related problems. *Proceedings of the seventeenth annual ACM Symposium on Theory of computing*. ACM, 185–194.

[79] HARMAN, M. et HIERONS, R. (2001). An overview of program slicing. *Software Focus*, 2, 85–92.

[80] HORWITZ, S., REPS, T. et BINKLEY, D. (1990). Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12, 26–60.

[81] HP (2013). Hp 2012 cyber risk report. `http://www.hpenterprisesecurity.com/collateral/whitepaper/HP2012CyberRiskReport_0313.pdf`.

[82] HU, H. et AHN, G. (2008). Enabling verification and conformance testing for access control model. *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*. ACM, 195–204.

[83] HUANG, Y., YU, F., HANG, C., TSAI, C., LEE, D. et KUO, S. (2004). Securing web application code by static analysis and runtime protection. *Proceedings of the 13th International Conference on World Wide Web*. ACM, 40–52.

[84] JAEGER, T., EDWARDS, A. et ZHANG, X. (2004). Consistency analysis of authorization hook placement in the linux security modules framework. *ACM Transactions on Information and System Security (TISSEC)*, 7, 175–205.

[85] JANG, D. et CHOE, K.-M. (2009). Points-to analysis for javascript. *Proceedings of the 2009 ACM Symposium on Applied Computing*. ACM, 1930–1937.

[86] JANG, J., AGRAWAL, A. et BRUMLEY, D. (2012). ReDeBug: Finding unpatched code clones in entire OS distributions. *Proceedings of the 2012 Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 48–62.

[87] JHALA, R. et MAJUMDAR, R. (2009). Software model checking. *ACM Computing Surveys*, 41, 1–54.

[88] JOVANOVIC, N., KRUEGEL, C. et KIRDA, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. *2006 IEEE Symposium on Security and Privacy*. IEEE Computer Society, 6–pp.

[89] JOVANOVIC, N., KRUEGEL, C. et KIRDA, E. (2010). Static analysis for detecting taint-style vulnerabilities in web applications. *Journal of Computer Security*, 18, 861–907.

[90] JR., E. M. C., GRUMBERG, O. et PELED, D. A. (1999). *Model Checking*. The MIT Press.

[91] KHOR, S. et GROGONO, P. (2004). Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically. *Proceedings of the 19th International Conference on Automated Software Engineering*. IEEE Computer Society, 346–349.

[92] KIEYZUN, A., GUO, P., JAYARAMAN, K. et ERNST, M. (2009). Automatic creation of SQL injection and cross-site scripting attacks. *Proceedings of the 31st ACM/IEEE International Conference on Software Engineering*. IEEE Computer Society, 199–209.

[93] KOLOVSKI, V., HENDLER, J. et PARSIA, B. (2007). Analyzing web access control policies. *Proceedings of the 16th International Conference on World Wide Web*. ACM, 677–686.

[94] KOREL, B. et LASKI, J. (1988). Dynamic program slicing. *Information Processing Letters*, 29, 155–163.

[95] KOREL, B. et RILLING, J. (1998). Dynamic program slicing methods. *Information and Software Technology*, 40, 647–659.

[96] KOVED, L., PISTOIA, M. et KERSHENBAUM, A. (2002). Access rights analysis for Java. *ACM SIGPLAN Notices*, 37, 359–372.

[97] LAVOIE, T. et MERLO, E. (2012). An accurate estimation of the Levenshtein distance using metric trees and Manhattan distance. *Proceedings of the 6th International Workshop on Software Clones*. 1 –7.

[98] LENGAUER, T. et TARJAN, R. E. (1979). A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1, 121–141.

[99] LETARTE, D., GAUTHIER, F. et MERLO, E. (2011). Security model evolution of php web applications. *Proceedings of the 4th International Conference on Software Testing, Verification and Validation*. IEEE Computer Society, 289–298.

[100] LETARTE, D. et MERLO, E. (2009). Extraction of inter-procedural simple role privilege models from PHP code. *Proceedings of the 16th Working Conference on Reverse Engineering*. IEEE, 187–191.

[101] LEVY, H. M. (1984). *Capability-based computer systems*, vol. 12. Digital Press Bedford.

[102] LHOTÁK, O. et HENDREN, L. (2004). Jedd: a BDD-based relational extension of Java. *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*. ACM, vol. 39, 158–169.

[103] LI, J. et ERNST, M. D. (2012). CBCD: cloned buggy code detector. *Proceedings of the 34th International Conference on Software Engineering*. IEEE/ACM, 310–320.

[104] LI, K. (2010). Towards security vulnerability detection by source code model checking. *Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation*. IEEE Computer Society, 381–387.

[105] LIVSHITS, B., NORI, A., RAJAMANI, S. et BANERJEE, A. (2009). Merlin: specification inference for explicit information flow problems. *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 75–86.

[106] LIVSHITS, V. et LAM, M. (2005). Finding security vulnerabilities in Java applications with static analysis. *USENIX Security Symposium*. USENIX Association, 18–18.

[107] MADSEN, M., LIVSHITS, B. et FANNING, M. (2013). Practical static analysis of javascript applications in the presence of frameworks and libraries. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. ACM, 499–509.

[108] MARTIN, E. et XIE, T. (2007). A fault model and mutation testing of access control policies. *Proceedings of the 16th International Conference on World Wide Web*. ACM, 667–676.

[109] MARTIN, R. C. (2003). *Agile software development: principles, patterns, and practices*. Prentice Hall PTR.

[110] MASOOD, A., BHATTI, R., GHAFOOR, A. et MATHUR, A. (2009). Scalable and effective test generation for role-based access control systems. *IEEE Transactions on Software Engineering*, 35, 654–668.

[111] MASOOD, A., GHAFOOR, A. et MATHUR, A. (2010). Conformance testing of temporal role-based access control systems. *IEEE Transactions on Dependable and Secure Computing*, 7, 144–158.

[112] MAYRAND, J., LEBLANC, C. et MERLO, E. (1996). Experiment on the automatic detection of function clones in a software system using metrics. *Proceedings of the 1996 International Conference on Software Maintenance*. IEEE, 244–253.

[113] MCGRAW, G. (2006). *Software security: building security in*, vol. 1. Addison-Wesley Professional.

[114] MEDINI, S., ANTONIOL, G., GUEHENEUC, Y., DI PENTA, M. et TONELLA, P. (2012). Scan: an approach to label and relate execution trace segments. *Proceedings of the 19th Working Conference on Reverse Engineering*. IEEE Computer Society, 135–144.

[115] MERLO, E., LETARTE, D. et ANTONIOL, G. (2007). Automated protection of PHP applications against SQL-injection attacks. *Proceedings on the 11th European Conference on Software Maintenance and Reengineering*. IEEE Computer Society, 191–202.

[116] MINAMIDE, Y. (2005). Static approximation of dynamically generated Web pages. *Proceedings of the 14th International Conference on World Wide Web*. ACM, 432–441.

[117] MOLLOY, I., CHEN, H., LI, T., WANG, Q., LI, N., BERTINO, E., CALO, S. et LOBO, J. (2008). Mining roles with semantic meanings. *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies*. ACM, 21–30.

[118] MOODLE (2011). Moodle documentation. `http://docs.moodle.org/`.

[119] MOODLE (2011). Moodle statistics. `http://moodle.org/stats/`.

[120] MOODLE FORUM (2012). Capabilities mod/lesson:edit and mod/lesson:manage. `https://moodle.org/mod/forum/discuss.php?d=89315`.

[121] MOODLE TRACKER (2012). MDL-36139. `http://tracker.moodle.org/browse/MDL-36139`.

[122] MOODLE TRACKER (2012). MDL-36140. `http://tracker.moodle.org/browse/MDL-36140`.

[123] MOUELHI, T., FLEUREY, F., BAUDRY, B. et TRAON, Y. (2008). A model-based framework for security policy specification, deployment and testing. *Model Driven Engineering Languages and Systems*, Springer Berlin Heidelberg, vol. 5301 de *Lecture Notes in Computer Science*. 537–552.

[124] NAUMOVICH, G. et CENTONZE, P. (2004). Static analysis of role-based access control in J2EE applications. *SIGSOFT Softw. Eng. Notes*, 29, 1–10.

[125] NGUYEN-TUONG, A., GUARNIERI, S., GREENE, D., SHIRLEY, J. et EVANS, D. (2005). Automatically hardening web applications using precise tainting. *Security and Privacy in the Age of Ubiquitous Computing*, 295–307.

[126] OBIEDKOV, S., KOURIE, D. et ELOFF, J. (2009). Building access control models with attribute exploration. *Computers & Security*, 28, 2–7.

[127] OTTENSTEIN, K. J. et OTTENSTEIN, L. M. (1984). The program dependence graph in a software development environment. *ACM SIGPLAN Notices*. ACM, vol. 19, 177–184.

[128] OWASP (2007). Owasp top 10 most critical web application security risks. `https://www.owasp.org/index.php/Top_10_2007`.

[129] OWASP (2012). Owasp top 10 most critical web application security risks. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[130] OWASP (2014). Owasp secure coding practices. `https://www.owasp.org/index.php/OWASP_Secure_Coding_Practices_-_Quick_Reference_Guide`.

[131] OWASP (2014). Owasp top 10 most critical web application security risks. `https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project`.

[132] PALMER, S. R. et FELSING, M. (2001). *A practical guide to feature-driven development*. Pearson Education.

[133] PEARCE, D., KELLY, P. et HANKIN, C. (2007). Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30, 4.

[134] PHAM, T.-H., TRUONG, N.-T. et NGUYEN, V.-H. (2009). Analyzing RBAC security policy of implementation using AST. *Knowledge and Systems Engineering, 2009. KSE '09. International Conference on*. 215–219.

[135] PHP (2012). The PHP manual. `http://www.php.net/manual/en/index.php`.

[136] PISTOIA, M., FLYNN, R., KOVED, L. et SREEDHAR, V. (2005). Interprocedural Analysis for Privileged Code Placement and Tainted Variable Detection. *Proceedings of the 19th European Conference on Object-Oriented Programming*. Springer, 362–386.

[137] POSHYVANYK, D. et MARCUS, A. (2007). Combining formal concept analysis with information retrieval for concept location in source code. *Proceedings of the 15th IEEE International Conference on Program Comprehension*. IEEE Computer Society, 37–48.

[138] POWER, D., SLAYMAKER, M. et SIMPSON, A. (2011). Automatic conformance checking of role-based access control policies via Alloy. *Engineering Secure Software and Systems*, Springer Berlin Heidelberg, vol. 6542 de *Lecture Notes in Computer Science*. 15–28.

[139] REPS, T., HORWITZ, S. et SAGIV, M. (1995). Precise interprocedural dataflow analysis via graph reachability. *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 49–61.

[140] ROBSCHINK, T. et SNELTING, G. (2002). Efficient path conditions in dependence graphs. *Proceedings of the 24th International Conference on Software Engineering*. ACM, 478–488.

[141] ROY, C. K. et CORDY, J. R. (2007). A survey on software clone detection research. Rapport technique Technical Report 2007-541, School of Computing, Queen's University.

[142] ROYCE, W. W. (1970). Managing the development of large software systems. *Proceedings of the 1970 IEEE WESCON Conference*. Los Angeles, vol. 26.

[143] SANDHU, R., COYNE, E., FEINSTEIN, H. et YOUMAN, C. (1996). Role-based access control models. *Computer*, 29, 38–47.

[144] SATYAM (2011). PHP grammar. `http://java.net/downloads/javacc/contrib/grammars/php.jj`.

[145] SCHMIDT, D. A. (1998). Data flow analysis is model checking of abstract interpretations. *Proceedings of the 25th ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, New York, NY, USA, 38–48.

[146] SCHWABER, K. (1997). Scrum development process. *Business Object Design and Implementation*, Springer. 117–134.

[147] SECURITY, A. (2013). 2013 global application security risk report. https://www.aspectsecurity.com/uploads/downloads/2013/06/Aspect-2013-Global-AppSec-Risk-Report.pdf.

[148] SECURITY, M. (2013). Real life vulnerabilities statistics: an overview. http://blog.mindedsecurity.com/2013/02/real-life-vulnerabilities-statistics.html.

[149] SECURITY, W. (2013). Website security statistics report. https://www.whitehatsec.com/assets/WPstatsReport_052013.pdf.

[150] SINGH, R., GIANNAKOPOULOU, D. et PĂSĂREANU, C. (2010). Learning component interfaces with may and must abstractions. *Computer Aided Verification*. Springer, 527–542.

[151] SOFTTEK (2013). State-of-the-art software security statistical report. https://www.softtek.com/webdocs/special_pdfs/WP-State-of-the-art-2013.pdf.

[152] SON, S., MCKINLEY, K. et SHMATIKOV, V. (2011). RoleCast: finding missing security checks when you do not know what checks are. *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 1069–1084.

[153] SON, S. et SHMATIKOV, V. (2011). SAFERPHP: Finding semantic vulnerabilities in PHP applications. *Proceedings of the ACM SIGPLAN 6th Workshop on Programming Languages and Analysis for Security*. ACM, 8.

[154] SRIDHARAN, M. et BODíK, R. (2006). Refinement-based context-sensitive points-to analysis for Java. *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*. ACM, vol. 41, 387–400.

[155] SRIDHARAN, M., DOLBY, J., CHANDRA, S., SCHÄFER, M. et TIP, F. (2012). Correlation tracking for points-to analysis of javascript. *ECOOP 2012–Object-Oriented Programming*, Springer. 435–458.

[156] SUN, F., XU, L. et SU, Z. (2011). Static detection of access control vulnerabilities in web applications. *USENIX Security Symposium*. 155–170.

[157] SUVÉE, D., VANDERPERREN, W. et JONCKERS, V. (2003). JAsCo: an aspect-oriented approach tailored for component based software development. *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*. ACM, 21–29.

[158] TILLEY, T., COLE, R., BECKER, P. et EKLUND, P. (2005). A survey of formal concept analysis support for software engineering activities. *Formal Concept Analysis*, Springer. 250–271.

[159] TIP, F. (1995). A survey of program slicing techniques. *Journal of Programming Languages*, 3, 121–189.

[160] TONELLA, P. (2004). Formal concept analysis in software engineering. *Proceedings of the 26th International Conference on Software Engineering*. IEEE Computer Society, 743–744.

[161] TONELLA, P. et CECCATO, M. (2004). Aspect mining through the formal concept analysis of execution traces. *Proceedings of the 11th Working Conference on Reverse Engineering*. IEEE Computer Society, 112–121.

[162] TRAON, Y. L., MOUELHI, T. et BAUDRY, B. (2007). Testing security policies: Going beyond functional testing. *Proceedings of the 18th IEEE International Symposium on Software Reliability*. 93–102.

[163] TRIPP, O., PISTOIA, M., FINK, S., SRIDHARAN, M. et WEISMAN, O. (2009). TAJ: effective taint analysis of web applications. *PLDI 2009, Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 87–97.

[164] ULLMAN, J. (1989). *Principles of Database and Knowledge-Base Systems*, vol. II. computer science press.

[165] VISSER, W., HAVELUND, K., BRAT, G., PARK, S. et LERDA, F. (2003). Model checking programs. *Automated Software Engineering*, 10, 203–232.

[166] VON AHN, L., BLUM, M., HOPPER, N. J. et LANGFORD, J. (2003). CAPTCHA: Using hard AI problems for security. *Advances in Cryptology—EUROCRYPT 2003*, Springer. 294–311.

[167] WANG, R., WANG, X., ZHANG, K. et LI, Z. (2008). Towards automatic reverse engineering of software security configurations. *Proceedings of the 15th ACM Conference on Computer and Communications Security*. ACM, 245–256.

[168] WASSERMANN, G. et SU, Z. (2007). Sound and precise analysis of web applications for injection vulnerabilities. *Proceedings of the 2007 Conference on Programming Language Design and Implementation*. ACM, 32–41.

[169] WASSERMANN, G. et SU, Z. (2008). Static detection of cross-site scripting vulnerabilities. *Proceedings of the 2008 ACM/IEEE International Conference on Software Engineering*. ACM/IEEE, 171–180.

[170] WEI, S. et RYDER, B. G. (2013). Practical blended taint analysis for javascript. *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 336–346.

[171] WEISER, M. (1981). Program slicing. *Proceedings of the 5th International Conference on Software Engineering*. IEEE Press, 439–449.

[172] WHALEY, J. et LAM, M. (2004). Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, 39, 131–144.

[173] XIE, Y. et AIKEN, A. (2006). Static detection of security vulnerabilities in scripting languages. *USENIX Security Symposium*. 179–192.

[174] XUAN-HIEU, P. et CAM-TU, N. (2007). GibbsLDA++: A C/C++ implementation of latent Dirichlet allocation. `http://gibbslda.sourceforge.net/`.

[175] YAMAGUCHI, F., LOTTMANN, M. et RIECK, K. (2012). Generalized vulnerability extrapolation using abstract syntax trees. *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 359–368.

[176] YAMAGUCHI, F., WRESSNEGGER, C., GASCON, H. et RIECK, K. (2013). Chucky: exposing missing checks in source code for vulnerability discovery. *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 499–510.

[177] YEVTUSHENKO, S. A. (2000). Concept explorer. *Proceedings of the 7th National Conference on Artificial Intelligence (Russia)*. 127–134.

[178] ZHANG, X., EDWARDS, A. et JAEGER, T. (2002). Using CQUAL for static analysis of authorization hook placement. *USENIX Security Symposium*. 33–48.

[179] ZHOU, J., ZHANG, H. et LO, D. (2012). Where should the bugs be fixed? More accurate information retrieval-based bug localization based on bug reports. *Proceedings of the 34th International Conference on Software Engineering*. IEEE Computer Society, 14–24.