

UC San Diego

UC San Diego Electronic Theses and Dissertations

Title

Detecting Access Control Misconfigurations with Change Validation

Permalink

<https://escholarship.org/uc/item/4685h4v1>

Author

Xiang, Chengcheng

Publication Date

2021

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA SAN DIEGO

Detecting Access Control Misconfigurations with Change Validation

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Chengcheng Xiang

Committee in charge:

Professor Yuanyuan Zhou, Chair
Professor Pamela C. Cosman
Professor George Porter
Professor Deian Stefan
Professor Geoffrey M. Voelker

2021

Copyright

Chengcheng Xiang, 2021

All rights reserved.

The Dissertation of Chengcheng Xiang is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2021

TABLE OF CONTENTS

| | |
|--|------|
| Dissertation Approval Page | iii |
| Table of Contents | iv |
| List of Figures | vii |
| List of Tables | viii |
| Acknowledgements | ix |
| Vita | xii |
| Abstract of the Dissertation | xiii |
| Chapter 1 Introduction | 1 |
| 1.1 Motivation | 1 |
| 1.2 Contribution | 4 |
| 1.2.1 Testing Access-control Configuration Changes with ACTEST | 5 |
| 1.2.2 Inferring Access-control Behavior Changes with P-DIFF | 6 |
| Chapter 2 Testing Access-control Configuration Changes with ACTEST | 8 |
| 2.1 Introduction | 8 |
| 2.2 Motivating Examples | 10 |
| 2.3 ACTEST Overview | 12 |
| 2.3.1 ACTEST Definitions | 13 |
| 2.3.2 Testing Principle | 14 |
| 2.3.3 Usage and Deployment Model | 15 |
| 2.4 ACTEST Generation | 15 |
| 2.4.1 Transforming Programs into <i>Safe</i> ACTESTs | 17 |
| 2.4.2 Making ACTEST Performance-efficient | 20 |
| 2.4.3 Minimizing Sysadmin Involvement | 25 |
| 2.4.4 Presenting ACTEST Results | 26 |
| 2.5 Evaluation | 27 |
| 2.5.1 Methodology and Testing Environment | 27 |
| 2.5.2 Detecting Real-world Vulnerabilities | 27 |
| 2.5.3 Detecting the Impacts of Injected Changes | 29 |
| 2.5.4 Performance | 34 |
| 2.6 Discussions | 35 |
| 2.7 Acknowledgement | 36 |
| Chapter 3 Inferring Access-control Behavior Change with P-DIFF | 37 |
| 3.1 Introduction | 37 |
| 3.1.1 Motivation | 37 |

| | | |
|-----------|---|----|
| 3.1.2 | Contributions | 39 |
| 3.2 | Observations of Real-World Access Control Systems | 41 |
| 3.2.1 | Access-Control Configurations | 41 |
| 3.2.2 | Access Logs | 42 |
| 3.2.3 | Access-Control Policies | 43 |
| 3.3 | Design Decisions | 44 |
| 3.3.1 | Inferring Policy from Access Logs | 44 |
| 3.3.2 | Using Decision Tree Based Models | 45 |
| 3.3.3 | Dealing with Sparse Logs | 46 |
| 3.4 | Threat Model | 46 |
| 3.5 | P-DIFF Overview | 49 |
| 3.6 | Policy Representation | 50 |
| 3.7 | Policy Inference | 54 |
| 3.7.1 | Parsing Access Logs | 54 |
| 3.7.2 | Policy Learning Algorithm | 55 |
| 3.7.3 | Namespace Inference | 56 |
| 3.8 | Policy Change Management | 57 |
| 3.8.1 | Algorithm | 57 |
| 3.8.2 | Optimizations | 60 |
| 3.9 | Use Cases | 61 |
| 3.9.1 | Change Validation | 61 |
| 3.9.2 | Forensic Analysis | 64 |
| 3.10 | Evaluation | 64 |
| 3.10.1 | Systems and Datasets | 64 |
| 3.10.2 | Experimental Design | 65 |
| 3.10.3 | Overall Results | 67 |
| 3.10.4 | Precision, Recall, and F-Score | 69 |
| 3.10.5 | False Positive and False Negative | 70 |
| 3.10.6 | Execution Time | 72 |
| 3.10.7 | Scalability | 72 |
| 3.10.8 | Validation Overhead | 73 |
| 3.11 | Discussions and Limitations | 75 |
| 3.12 | Acknowledgement | 76 |
| Chapter 4 | Related Work | 77 |
| 4.1 | Access-control Misconfiguration | 77 |
| 4.1.1 | Access-control Configuration Testing and Verification | 77 |
| 4.1.2 | Access-control Misconfiguration Detection | 78 |
| 4.1.3 | Characteristic Study of Access-deny Issues | 78 |
| 4.2 | Other Types of Misconfiguration | 79 |
| 4.2.1 | Configuration Testing | 79 |
| 4.2.2 | Misconfiguration Detection | 79 |
| 4.3 | Other Related Techniques | 80 |
| 4.3.1 | Access-control Bug Detection | 80 |

| | | |
|-----------|------------------------------------|----|
| 4.3.2 | Intrusion Detection | 80 |
| 4.3.3 | Decision Tree Algorithms | 81 |
| 4.3.4 | Execution Acceleration | 81 |
| 4.4 | Acknowledgement | 82 |
| Chapter 5 | Conclusion | 83 |
| | Bibliography | 86 |

LIST OF FIGURES

| | | |
|--------------|---|----|
| Figure 2.1. | A new vulnerability detected by ACTESTs in a Docker image of MediaWiki. | 12 |
| Figure 2.2. | A new vulnerability detected by ACTESTs in a Docker image of Drupal. . . | 13 |
| Figure 2.3. | Request handling code pattern..... | 21 |
| Figure 2.4. | Advanced trimming by comparing allowed and denied execution traces. . . | 22 |
| Figure 2.5. | Normalized test running-time by different trim methods. | 35 |
| Figure 3.1. | Examples of access logs (the inputs of P-DIFF) and the configurations and code implementation of access control in the Wikipedia system..... | 38 |
| Figure 3.2. | Decision tree representations of three access control models. | 46 |
| Figure 3.3. | Example of sparse accesses to a course website and the decision tree inferred from them. | 47 |
| Figure 3.4. | The workflow of P-DIFF..... | 48 |
| Figure 3.5. | An example of access-control policies in configuration files, access logs, and a decision tree. | 52 |
| Figure 3.6. | A traditional decision tree and a Time-Changing Decision Tree (TCDT) generated from the logs in Figure 3.4..... | 53 |
| Figure 3.7. | Examples that demonstrate splitting events in TCDT-based policy learning. | 58 |
| Figure 3.8. | An example that demonstrates TCDT learning algorithm can infer rules even there is no rule change..... | 60 |
| Figure 3.9. | Two optimizations to efficiently calculate change-count. | 62 |
| Figure 3.10. | Two use cases with P-DIFF: (a) change validation and (b) forensic analysis. | 63 |
| Figure 3.11. | Precision, recall, and F-score of TCDT classifying access results for the Center and Course datasets..... | 71 |
| Figure 3.12. | Scalability analysis in terms of execution time for training, validation, and forensics with the increasing numbers of log entries from the Wikipedia dataset. | 74 |

LIST OF TABLES

| | | |
|------------|--|----|
| Table 1.1. | Example of recent security incidents caused by access-control misconfigurations. | 1 |
| Table 2.1. | New vulnerabilities detected by ACTESTs in public Docker images of popular web systems..... | 28 |
| Table 2.2. | Common types of vulnerabilities detected..... | 30 |
| Table 2.3. | Real-world systems and access logs collected for evaluation. | 30 |
| Table 2.4. | Different types of access-control configuration changes injected in our experiments..... | 31 |
| Table 2.5. | Detecting change impacts with ACTESTs generated by different trimming methods. | 32 |
| Table 2.6. | Detecting change impacts with different ways of generating test inputs. ... | 33 |
| Table 3.1. | Information encoded in access logs of different software. | 43 |
| Table 3.2. | Annotations of the log format. | 54 |
| Table 3.3. | Datasets used in our evaluation..... | 66 |
| Table 3.4. | Different types of access control policy changes used in the experiments. ... | 67 |
| Table 3.5. | Policy changes detected by P-DIFF..... | 68 |
| Table 3.6. | Effectiveness of forensic analysis. | 69 |
| Table 3.7. | Performance of P-DIFF: training time, and time for validation and forensics per access. | 72 |

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor Professor Yuanyuan (YY) Zhou for her invaluable guidance, generous support and profound inspiration on both my research and life. This dissertation is not possible without YY's insight and wisdom. YY taught me what scientific truth is and guided me to pursue truth by thinking, trying and creating. YY's wisdom and inspiration always gave me direction and confidence to travel through mist and reach the frontiers that are less travelled. Beyond research, YY also encouraged me to pursue the best I can be. YY's spirit on questing for perfection makes me see the world differently. YY's emphasis on thinking, communicating and being tough has been life-changing to me. I may not be smarter than what I was five years ago, but I believe I have become more independent-minded, more outgoing and tougher when facing life. Thank you, YY!

I would like to thank Professor Geoffrey M. Voelker. Geoff encouraged me many times to do presentations on SysLunch as well as CSE294, and he gave me a lot of helpful feedback on my research. Geoff also maintained the sysnet website, which provides important data for experiments presented in this dissertation. The sysnet hiking Geoff leads always gave me a chance to recharge myself and make many new friends.

I would also like to thank my thesis committee members Professor Pamela C. Cosman, Professor George Porter, Professor Deian Stefan and Professor Geoffrey M. Voelker for always being supportive and giving me their insightful feedback on this dissertation.

I was very lucky to have chance to work with my academic brother, Professor Tianyin Xu. Tianyin set up a great example to me when he was still at UCSD during my first year. Tianyin shared a lot of his experience on doing research and inspired me to think more by asking many in-depth questions. Even when he moved to UIUC, Tianyin still generously gave me a lot of help on the two projects in this dissertation.

I learned a lot during my internship at NetApp working with Shankar Pasupathy and Anita Jinda. They gave me best guidance and support on studying enterprise data as well as the freedom to explore. This invaluable experience gave me insight and sense on configuration

problems that enterprises are facing.

I want to thank Cindy Moore, who provides both valuable data and sysadmins' perspectives for this dissertation. Cindy spent a lot of time on collecting dataset from the sysnet websites and helping me understand the data. Cindy also shared a lot of her wisdom as an experienced sysadmins, which is invaluable to this dissertation.

I am also thankful to my collaborators from the Whova company: Tianwei Sheng, Xinxin Jin and Simon Ninon. They helped collect industrial data that provide indispensable insight for this dissertation. Tianwei and Xinxin also shared a lot of their own thinking and experience on the problems this dissertation studies.

I was very lucky to work with all the members in the Opera group. They are both collaborators and friends. Yudong Wu and Bingyu Shen provided me extremely generous help when I was a junior student. They stayed up with me to fight for my first paper deadline. Haochen Huang and Li Zhong selflessly collaborated with me on multiple projects. Their thinking and hard-work contributed a lot to projects presented in this dissertation and several other projects I worked on. I was also very privileged to work with Mingyao Shen, Andrew Yoo, Tianyi Shan, Nathaniel Nguyen and Eric Mugnier.

I would also like to thank all the members in the Sysnet group, including all the faculties, staffs and students. Sysnet is a big family that is full of both inspiration and fun. Special thanks to Vector Li, who shared with me his own experiences and stories to encourage me during my lowest points. Thanks Shelby Thomas, Stewart Grant, Rajdeep Das, Keegan Ryan, Lixiang Ao, Nishant Bhaskar and Alex Liu for helping me proofread my papers and giving me a lot of useful feedback on my projects.

Last but not the least, I am deeply indebted to my parents. They gave me both the perseverance to never give up and also the unconditional love that I can always give up. They do not know much about research, but their conscientious attitude towards work always motivate me to pursue perfection with hardworking.

Chapter 2 and 4, in part, is a reprint of the material under submission. Xiang, Chengcheng;

Mugnier, Eric; Nguyen, Nathaniel; Huang, Haochen; Zhong, Li; Zhou, Yuanyuan; Xu, Tianyin. The dissertation author was the primary investigator and author of this paper.

Chapter 3 and 4, in part, is a reprint of the material as it appears in Proceedings of the 26th ACM Conference on Computer and Communications Security, 2019. Xiang, Chengcheng; Wu, Yudong; Shen, Bingyu; Shen, Mingyao; Huang, Haochen; Xu, Tianyin; Zhou, Yuanyuan; Moore, Cindy; Jin, Xinxin; Sheng, Tianwei. The dissertation author was the primary investigator and author of this paper.

VITA

- 2013 Bachelor of Engineering, Shanghai Jiao Tong University
- 2016 Master of Engineering, Shanghai Jiao Tong University
- 2021 Doctor of Philosophy, University of California San Diego

PUBLICATIONS

“Detecting Risky Access-control Changes with Automatic Configuration Testing”. **Chengcheng Xiang**, Eric Mugnier, Nathaniel Nguyen, Haochen Huang, Li Zhong, Yuanyuan Zhou, Tianyin Xu. Under submission.

“PYLIVE: On-the-Fly Code Change for Python-based Online Services”. Haochen Huang*, **Chengcheng Xiang*** (co-first), Li Zhong, Yuanyuan Zhou. 2021 USENIX Annual Technical Conference (ATC’21), July 2021.

“PracExtractor: Extracting Configuration Good Practices from Manuals to Detect Server Misconfigurations”. **Chengcheng Xiang**, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, Shankar Pasupathy. 2020 USENIX Annual Technical Conference (ATC’20), July 2020.

“Towards Continuous Access Control Validation and Forensics”. **Chengcheng Xiang**, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, Tianwei Sheng. The 26th ACM Conference on Computer and Communications Security (CCS’19), November 2019.

“Can Systems Explain Permissions Better? Understanding Users’ Misperceptions under Smartphone Runtime Permission Model”. Bingyu Shen, Lili Wei, **Chengcheng Xiang**, Yudong Wu, Mingyao Shen, Yuanyuan Zhou, Xinxin Jin. the 30th USENIX Security Symposium (Security’21), August 2021.

ABSTRACT OF THE DISSERTATION

Detecting Access Control Misconfigurations with Change Validation

by

Chengcheng Xiang

Doctor of Philosophy in Computer Science

University of California San Diego, 2021

Professor Yuanyuan Zhou, Chair

Access-control misconfigurations are among the main causes of today’s security incidents. One main reason is that access-control configurations need to be frequently changed by system administrators (sysadmins) to accommodate dynamic information sharing. Unfortunately, to err is human—sysadmins often make mistakes (e.g., over-granting privileges) when changing access control configurations. Such mistakes can stay unnoticed for a long time until eventually being exploited by attackers, causing catastrophic security incidents.

This dissertation explores two validation approaches to detect access-control misconfigurations at different life-cycle stages of systems. The first approach is to test access-control configuration changes before they are deployed to production. This can help sysadmins detect

access-control misconfigurations before they bring any real harm to production systems. The second approach is to monitor access-control behavior changes after the configuration changes are deployed to production. This can help sysadmins detect and diagnose potential data leaks caused by access-control misconfigurations quickly so that they can be fixed timely.

First, this dissertation presents a new type of test programs, ACTESTs, to test access-control configuration changes and a new approach to generate such test programs from existing program code. ACTESTs output the impacts of access-control changes—what requests were denied, but will be allowed after a change, and vice versa. With this, sysadmins can validate if the changed requests are intended or not and identify potential security vulnerabilities. The key challenges this dissertation addressed include making ACTESTs *safe* to run in production environments and making them *performance-efficient*. ACTESTs help detect 168 new misconfigurations from 72 Docker images.

Second, this dissertation presents P-DIFF, a practical tool for monitoring access-control behavior to help sysadmins early detect unintended access-control configuration changes and perform postmortem forensic analysis upon security attacks. P-DIFF continuously monitors access logs and infers access-control behavior changes from them. This dissertation devises a novel time-changing decision tree to effectively represent access-control behavior changes, coupled with a new learning algorithm to infer the tree from access logs. Evaluation shows that P-DIFF can detect 76%–100% of access control behavior changes.

Chapter 1

Introduction

1.1 Motivation

In recent years, access-control misconfigurations have been the culprit of many security incidents, including data leakage, website compromises and ransomware [113, 101, 66, 27]. For example, in December 2019, a wrong configuration change to a database’s network security rules at Microsoft leaked 250 million entries of email, IP address and support case details[36]. In July 2020, 23000 misconfigured MongoDB databases are wiped by hackers and the victims are asked to pay ransom[135]. Table 1.1 displays more recently-reported security incidents caused by access-control misconfigurations.

Table 1.1. Example of recent security incidents caused by access-control misconfigurations.

| Time | Incident | Organization |
|-------------|--|---------------------|
| 2019.7 | 100 million customer data leaked [20] | CapitalOne |
| 2019.9 | 20 million citizen records exposed [134] | Novaestrat |
| 2019.12 | 250 million customer data leaked [36] | Microsoft |
| 2020.3 | 10.88 billion records leaked [120] | CAM4 |
| 2020.7 | 23,000 MongoDB are ransomed [135] | Many Orgs |
| 2020.10 | Sensitive patient data exposed [98] | Pfizer |
| 2021.1 | 300 million customer data leaked [73] | Astoria Company |
| 2021.3 | 300,000 customer data leaked [92] | Hobby Lobby |
| 2021.4 | 2.5 million customer file leaked [37] | Bizongo |

Access-control misconfigurations cause severe outcomes. First, they open doors for

unauthorized users to access private data. As shown by the incidents in Table 1.1, attackers can easily download a large amount of private data and sell them on the black market or ask for ransom. To make the matter worse, these doors are often opened for a long time. Unlike service down, opened data access would not block anyone's work and so users have less incentive to report it. An IBM report shows that it takes 207 days on average for companies to identify a data breach [51]. In extreme cases, it takes years to discover a data breach [93].

A major challenge of managing access-control configuration is to accommodate constant configuration changes. Access-control configuration is never a one-time effort but requires continuous changes to accommodate the dynamics of data and resource sharing, as well as high churn and updates of new protection domains and organizational roles [127, 23, 99]. The following lists a few common cases when sysadmins need to modify access-control configurations to accommodate user, data, functionality and domain changes.

- *User change*: Users may join, depart or change roles within an organization or a project.
- *Data change*: Some data may become sensitive or start to contain sensitive information.
- *Functionality change*: New features, accesses or services are added for the public or a certain group of user to access.
- *Domain change*: Data need to be reorganized into different domains or subdomains.

Unfortunately, access-control configuration changes are error-prone for at least three reasons: 1) Access-control configurations are complex and heterogeneous. The configurations are often not "What You See Is What You Get" [86, 65], but are encoded in app-specific directives and rules with different formats. For example, file systems use bit vector to represent permissions, web servers take regular expressions to match URLs for deny/allow, and databases use customized roles to define privileges. 2) Sysadmins are under time pressure to make a change when important requests are incorrectly denied. One common case is about handling access-denied issues: when a user complains about being denied to something that she is supposed to have access, sysadmins

need to address the issue quickly for her. Because of the time pressure, sysadmins may perform some quick changes as workarounds without carefully checking if the changes grant the least requested permissions or grant additional unexpected permissions. That is, sysadmins may prioritize availability over security without precisely understanding the consequences [115, 57].

3) Diagnosis logs for access-control are often vague and not actionable [127, 17]. As a result, it is hard for sysadmins to precisely understand the access-control behavior without looking into the code. In reality, sysadmins often perform a trial-and-error process to workaround problems [127].

To eliminate errors, verification and validation are two major methodologies that are widely applied in many areas, including software and hardware systems. Verification tries to ensure “*building systems right*”, namely building systems that comply with given specifications and requirements. And validation aims to assure “*building the right systems*”, namely building systems that meet the need of users or stakeholders. In the context of access-control configuration, verification tries to ensure that a set of configurations can be correctly interpreted by the target systems or can fulfill human-provided properties; validation, instead, aims to assure that the configurations are what users/sysadmins need, namely the configured systems will behave as they expected.

Previous work on detecting access-control misconfigurations mostly lies in the verification direction [97, 35, 48, 54, 71, 53]. These techniques usually require humans to translate actual system configurations into formal models and rely on humans to provide specifications to verify. However, due to the diversity and complexity of access-control implementations, it takes significant human effort to encode system configurations using modeling languages, let alone maintain consistent models when system configurations change. In addition, it is also hard for humans to provide a complete set of specifications for verification. As a result, verification cannot practically eliminate all errors, and validation is still needed for sysadmins to detect errors that cause unintended results.

The key missing piece of today’s access-control management is tools that can help sysadmins *validate* their configuration changes. Validating configuration changes need to know

the actual impacts on the access-control behavior of running systems. Without tooling support, sysadmins can either infer access-control behavior statically from configurations or running tests against running systems, of which both have many challenges. First, it is hard to manually infer the access-control behavior of running systems from the static access-control configurations. As discussed before, access-control configurations are encoded in app-specific directives with different formats, including bit vector, regular expressions and database tables. It is non-trivial (if not impossible) to reason about the access-control behavior by inspecting the configurations statically. Second, it is also difficult to manually evaluate access-control behavior by running tests against running systems. Such tests need to be system tests and cannot cause side effects to actual system data. Therefore, running such tests need a dedicated testing environment, which is costly and not easy to manage. In addition, such tests rely on a good coverage of possible data requests so that access allow and deny are thoroughly exposed. These cannot be easily generated manually, as the number of possible requests can be large and the requests can be complex.

1.2 Contribution

This dissertation presents two validation methods to help sysadmins validate their access-control configuration changes. These two validation methods target for *two different life-cycle stages of systems*. First, before an access-control configuration change is deployed to production systems to take effect, we present ACTEST to test the changed configuration and assist sysadmins understand its impacts on access-control behavior so that sysadmins can decide if the impacted behavior is intended or not. Second, after an access-control configuration change has been deployed to production systems, we present P-DIFF to infer the behavior changes from access logs so that sysadmins can detect an unintended data exposure early and avoid further data leak.

Thesis Statement: *By validating access-control configuration changes with testing and machine learning techniques at different system life-cycle stages, sysadmins can effectively detect access-control misconfigurations.*

1.2.1 Testing Access-control Configuration Changes with ACTEST

Chapter 2 presents ACTEST. ACTEST is a type of test programs to help sysadmins test their access-control configuration changes before they are deployed. Every time sysadmins change their access-control configurations, they can run ACTEST to understand the impacts on the access-control behavior, namely what requests were denied before become allowed and vice versa. With this, sysadmins can validate if the change impacts on behavior is intended or not. If it is not intended, they can further modify their configurations to make them more secure.

ACTEST has three properties. First, ACTEST can be safely run with production data and configuration. This saves sysadmins' efforts from setting up and maintaining dedicated testing environments, which is prohibitively costly [18, 5]. Second, ACTEST can generate a good coverage of test requests to expose access-control behavior changes. This frees sysadmins from learning to write test code or specifying all test requests manually. Third, ACTEST is performance-efficient. ACTEST only runs the code logic to generate the access-control results (i.e. allow/deny) but eliminates the part for unrelated computation and I/O. Therefore, running ACTEST is much faster than testing with production system directly.

ACTEST can be particularly built by developers or generated from the original programs. To minimize developers' efforts, Chapter 2 presents a general approach to turn an original program into an ACTEST. First, the approach runs the original program in a virtualized environment and uses copy-on-write to avoid it causing any side effects to production data. Second, the approach uses two ways to generate test requests: 1) using historical access logs to generate test requests; 2) letting sysadmins specifying the target subjects (e.g., users), objects (e.g., files) and actions (e.g. GET, PUT) and generating the Cartesian product of them to synthesize possible requests for testing. Third, our approach trims the original program to speed up its execution. This is based on an observation that the access-control checks are typically called before the major computation and I/O are executed. Only executing the access-control checks is enough to get the access-control results and the original program can be trimmed to speedup the whole

execution.

Chapter 2 evaluates ACTESTS for various systems with configurations in public Docker images from DockerHub. ACTESTS have helped detect 168 new vulnerabilities from 72 Docker images. They have been reported to the image maintainers, and so far 25 of them have been confirmed and 19 of them have been fixed by the maintainers. Chapter 2 also evaluates ACTESTS with five real-world systems, including Wikipedia and the web proxy of a commercial company. The results show that: 1) with replaying access logs, ACTESTS can detect up to 80% of all the behavior impacts of injected changes; 2) by synthesizing requests, ACTESTS can detect all the behavior impacts of injected changes. For performance, ACTESTS saves 9.09%–98.61% execution time from testing with the original programs.

1.2.2 Inferring Access-control Behavior Changes with P-DIFF

Chapter 3 presents P-DIFF, a practical tool that can infer access-control behavior changes from *access logs*. P-DIFF takes solely existing systems’ access logs as input and requires sysadmins to record no configuration changes. Recorded configuration changes are not sufficient for humans to reason about behavior changes as they are not “What You See Is What You Get”. In addition, sometimes it is also hard to record as file permissions and network-level firewalls can be modified by users and other superusers without sysadmins’ awareness.

P-DIFF can be used to assist sysadmins in two use cases: 1) Change validation: P-DIFF can be used as a monitoring tool that continuously analyzes the access logs to detect behavior changes. When a behavior change is detected, P-DIFF can warn sysadmins about the change and ask sysadmins to validate if the change is intended or a misconfiguration. 2) Forensic analysis: when a security incident has been discovered, P-DIFF can be used for postmortem analysis. P-DIFF can detect all behavior changes happened in the history and let sysadmins examine *when* and *which* change opened up the access.

The key contribution of P-DIFF is a novel learning algorithm for inferring behavior changes from access logs. The algorithm uses a decision-tree to encode the access-control

behavior of a system and stores time series in the tree leaves to record behavior changes. This dissertation refers the decision tree as Time-Changing Decision Tree (TCDT). TCDT cannot be inferred with traditional learning algorithms as traditional algorithms do not consider changes in the learning data. This dissertation presents a new TCDT learning algorithm that analyzes access logs as a sequence of access events ordered by time to capture changes.

Chapter 3 evaluates P-DIFF with access logs collected from five real-world deployed systems, including *two from industrial companies*. For change validation, P-DIFF can effectively detect 76%–100% of the behavior changes with an average precision of 89%. For forensic analysis, P-DIFF can detect the root-cause changes for 85%–98% of the evaluated cases.

Chapter 2

Testing Access-control Configuration Changes with ACTEST

2.1 Introduction

With the error-proneness of configuration changes, a key missing piece of current access-control management is tools that can help sysadmins test their configuration changes. Without tooling support, sysadmins face two challenges to test their configurations manually. First, testing access-control requires system-level tests to evaluate end-to-end system behavior; however, organizations may not maintain a dedicated testing environment because it is costly and is not easy to manage, as shown in forum discussions [95, 96]. This restricts sysadmins to only perform limited tests on their production systems. The tests can only be the ones without side effects (e.g. read data) but cannot be the opposite (e.g. modify data). Second, testing access-control requires a good coverage of possible requests. These cannot be easily generated manually, as the number of possible requests can be large and the requests can be complex. In addition, running the test requests may need to tackle other technical challenges, like mimicking a source IP address.

This chapter presents a new type of test programs that software vendors can release for sysadmins to validate their access-control configuration changes. Such test programs can be released along with the main product programs so that sysadmins can easily download and use for test. Vendors can choose to develop such test programs from scratch as an individual tool. To help software vendors create such test programs, this chapter presents a *general* approach to

generate such test programs from *existing* system code. We term the generated test programs as ACTESTS.

With ACTESTS, every time sysadmins change access-control configurations, they can run the test programs to detect the impacts of the change—what requests are changed from deny to allow and vice versa. Such testing enables sysadmins to early detect risky changes to prevent unexpected access that results in security incidents.

ACTESTS can be run in production. It saves sysadmins from setting up dedicated testing environments—setting up such environments requires replicating the production systems including production data and dependencies, which is prohibitively costly [18, 5] and thus is not a common practice for many organizations [95, 96]. ACTESTS require minimal effort from sysadmins. It does not require sysadmins to implement the test code or to specify every testing inputs manually.

To generate ACTESTS, we need to address three major challenges: 1) As ACTESTS need to be run in production, how to avoid the target programs making side effects to production systems? 2) How to generate testing requests without much effort from sysadmins? 3) To achieve comprehensive tests with a high coverage of test requests, how to make ACTESTS performance-efficient to run large-scale tests efficiently?

To address the first challenge, we leverage sandboxing techniques to avoid any unexpected side effects. First, an ACTEST encapsulates itself inside a isolated, virtualized environment with resource caps to avoid interference with production processes. Second, an ACTEST uses copy-on-write for data access to avoid modifying production data. Third, an ACTEST is run in a virtualized network with firewall rules to block the outgoing traffic so that it cannot affect production network traffic.

For the second challenge, we develop two approaches to generate test inputs. First, ACTESTS can take historical access logs specified by sysadmins to generate test requests. As shown in [125], most server programs record requests in access logs. Second, if historical access logs are not available or have a low coverage on possible requests, ACTESTS allow sysadmins to

specify target subjects (e.g., the database table of user names), objects (e.g., the root directory of all HTML files) and actions (e.g. GET, PUT). Then ACTESTs synthesize testing requests based on the Cartesian product of the subjects, objects and actions.

To address the third challenge, we propose an approach to modify the target system to speed up its request handling. This is based on an observation that access-control checks are usually executed in an early phase of the data paths, before major disk I/O and network that take up most the request handling time. Since these early-phase access-control checks have already evaluated the access-control results, the subsequent procedures can be skipped without affecting the test results. We present two trimming methods to accelerate ACTESTs by skipping the data-related procedures.

We evaluate ACTESTs by using them to detect the change impacts in public Docker images of five widely-used web systems. With the detected change impacts, 168 new vulnerabilities are identified from 72 Docker images. We report them to the image maintainers. So far 25 of them have been confirmed and 19 of them have been fixed by the maintainers.

We also evaluate ACTESTs with five real-world deployed systems, including Wikipedia and the web proxy of a commercial company with millions of customers. The results show 1) by synthesizing testing requests, ACTESTs can detect all the impacts of randomly injected changes to these systems; 2) by replaying historical requests, ACTESTs can detect up to 80% of all the injected impacts. ACTESTs also perform efficiently. Our trimming methods can reduce 9.09%–98.61% running time over testing with the original programs.

2.2 Motivating Examples

We present how change impacts detected by ACTESTs can help sysadmins validate their access-control configuration changes and identify security vulnerabilities. We use two real-world access-control misconfigurations ACTESTs detected from Docker images as examples.

Dangerous web interfaces exposed by installing PHP extensions.

ACTESTs can detect unexpected interfaces and resources introduced by configuration changes. This helps sysadmins validate a change when they install third-party plugins or extensions, a common practice for web applications [121, 116, 33]. As exemplified by both real-world incidents [108] and our evaluation on Docker images, installing extensions could incur security risks.

Figure 2.1 shows a new vulnerability detected by ACTESTs in a Docker image of MediaWiki [117], a popular open-source wiki system. In this example, the sysadmin installed a third-party PHP extension “MW-0Auth2Client” that introduced several dangerous PHP files, including “eval-stdin.php”. These files exposed web interfaces that should only be used for testing environments and can be exploited for remote code execution attacks in production. However, sysadmins were not aware of these files and failed to apply any access-control rules to limit the access, making the dangerous interfaces in the files publicly accessible. ACTESTs effectively detect that these files were not accessible before the installation, but become accessible after. It warns sysadmins so they can double check if the exposed interfaces in the files are safe and make appropriate configurations to prevent requests to the dangerous interfaces before they are exploited.

Openly-accessible database dump due to rule misconfigurations.

ACTESTs can also detect access-control misconfigurations that lead to unexpected impacts. Oftentimes, access-control configurations are complex and error-prone—they are not “What You See Is What You Get,” but use bit vector (e.g. file permissions), regular expressions (e.g. URL match) and database roles. Therefore, it is necessary to test the actual impacts in terms of system behavior for sysadmins to validate if the behavior change meets their intention.

Figure 2.2 shows a new vulnerability detected by ACTESTs in a Docker image of Drupal (a widely-used CMS system). In a change, the sysadmin adds a few database files, such as “vov_500.sql” and “db/light.sql.gz”. She adds a customized access-control rule to prevent

System configuration Change:

Installed a third-party extension “MW-0Auth2Client”

Vulnerability:

Failed to block public accesses to dangerous web interfaces introduced by the extension, such as:

- /extensions/MW-0Auth2Client/phpunit/eval-stdin.php

Consequence:

Enabled remote code execution attack that may cause a full compromise of the web server.

Changed accesses detected by our tool

| Before | After |
|---|--|
| anonyuser, GET, phpunit/eval-stdin.php, NotFound | anonyuser, GET, phpunit/eval-stdin.php, Allowed |

Figure 2.1. A new vulnerability detected by ACTESTs in a Docker image of MediaWiki [117]. This vulnerability has been confirmed by the image maintainer. ACTESTs detect that anonymous users cannot access phpunit/eval-stdin.php by default, but can access it after the change.

these files from being accessed publicly. The rule is in the form of a regular expression “*sql”. However, the added rule only blocks files with “.sql” but not with “.sql.gz”. This makes the file “db/light/sql.gz” open to public access. Note that this file is a database dump, including both users data and admins account info. ACTESTs effectively detect only the “.sql” files are blocked but the “.sql.gz” files are still accessible. It can warn sysadmins so that they can further change their configurations to block requests to the “.sql.gz” files.

2.3 ACTEST Overview

ACTESTs are test programs for helping sysadmins analyze the *impact* of access-control changes in terms of end-to-end system behavior. A sysadmin can run ACTESTs for each change of access-control configurations. ACTESTs output the impact of the configuration change, in the form of the difference of access-control behavior, before and after the target change. Based on ACTESTs’ output, the sysadmin can examine whether the configuration change results in intended

System configuration change:

- Add db files: db/vov_500.sql, db/light.sql.gz
- Add access rules to block accesses to db files

```
- <Files "\.(engine|inc|install|make|profile)
  (~|\.sw[op]|\.bak|\.orig)?$|...">
+ <Files "\.(engine|inc|install|make|profile|.*sql)
  (~|\.sw[op]|\.bak|\.orig)?$|...">

  require all denied
</FilesMatch>
```

Vulnerability:

Only blocked files “*.sql” but not “*.sql.gz” to public access.

Consequence:

Sensitive data, including admin account and user information, in the database dump file “db/light.sql.gz” is exposed to public.

Changed accesses detected by our tool:

| Before | After |
|---|--|
| anonyuser, GET, db/light.sql.gz, NotFound | anonyuser, GET, db/light.sql.gz, Allowed |

Figure 2.2. A new vulnerability detected by ACTESTS in a Docker image of Drupal [32]. ACTESTS detect that anonymous users cannot access the database dump file db/light.sql.gz before but can access it after the change. ACTESTS can warn sysadmins so they can restrict the access before the data is leaked.

access-control behavior. ACTESTS can help sysadmins to identify incorrect or unintended access-control that either leads to security vulnerabilities such as data breaches (misconfigurations that grant more access than what is intended) or accessibility issues (misconfigurations that do not grant sufficient access for the desired functionality).

2.3.1 ACTEST Definitions

For a target system, an ACTEST is a system-level test program, denoted as $t(C, D, \Delta C)$. It takes inputs: 1) the current system configuration C , 2) the configuration change ΔC , and 3) the current system data, D . Note that C and D are the entire set of system configurations and data, not limited to access-control related ones. In a typical deployment scenario, C and D are production configurations and data of the system is operating on, and the ACTEST runs before

ΔC is rolled out to production. As discussed above, the entire set of configurations and data are needed to achieve high-fidelity of the testing results and ease the deployment.

After testing a configuration change ΔC , the ACTEST outputs the change impact as the set of requests that have different results before the change and after the change—namely, requests that were denied by C , but become allowed after applying ΔC to C , and vice versa. A request can be defined as a tuple $\langle s, o, a, r \rangle$, which represents a subject s (e.g., a user) performs an action a (e.g., an HTTP GET/PUT request) to an object o (e.g., a file) and gets a result r (either ALLOW or DENY). Therefore, the impact of the change is a set of tuples $\langle s, o, a, r, r' \rangle$, where r is the access results with C , r' is the result with ΔC and $r \neq r'$. Our definition of access-control change impact is consistent with prior work [125].

2.3.2 Testing Principle

Conceptually, an ACTEST calculates the change impact by testing and comparing the access-control behavior with C and $C + \Delta C$. The ACTEST sets up the system with the production configuration C and data D and the system with $C + \Delta C$ and D , respectively. Then, the ACTEST issues a set of access requests to both of the setups and observes their access-control behavior. The access-control behaviors are recorded in two separate logs L and L' , each of which records a sequence of access tuples $\langle s, o, a, r \rangle$ as defined in §2.3.1. Each line l in L and L' should have $s_i = s'_i$, $o_i = o'_i$, and $a_i = a'_i$, but may differ in $r_i \neq r'_i$. ACTEST then computes a “diff”, $\Delta L = L' - L$, which contains all line l with $r_i \neq r'_i$, as the change impact.

ACTESTs provide two ways to generate the test inputs, in the form of $\langle s, o, a \rangle$, i.e., requests. First, ACTESTs allow sysadmins to provide access logs which record the historical requests. The access logs help ACTESTs cover the common requests. Second, ACTESTs also allow sysadmins to specify the subjects, objects, and actions, which will be used to generate all possible combinations to achieve a complete coverage.

2.3.3 Usage and Deployment Model

ACTESTs can be effectively generated for any software system. We discuss how to generate ACTESTs in §2.4.

ACTESTs target to be run in production. As discussed in §2.1, this is necessary because both production environments and data are needed to decide the end-to-end access-control behavior. Without in-production test, it then requires to maintain a full copy of production environments as well as data and make sure the test environment in sync with the production environment, which is known as impractical and prohibitively costly [18, 5].

To run ACTESTs in production, sysadmins only need to specify a few inputs for ACTESTs: (1) the location of the production configuration C and data D ; (2) the target configuration change ΔC ; (3) a set of access logs, and/or a specification of target subjects, actions and objects (e.g., the root directory of all HTML files to test); and (4) optionally, the limit of CPU and memory that ACTESTs are expected to take.

Multiple ACTEST instances may need to be deployed to test different programs. For these cases, all the ACTEST instances can be placed in a single virtualized subnet so they can communicate with each other but do not affect production processes (cf. §2.4.1).

ACTESTs target on testing a configuration change ΔC to help sysadmin reason about the *impact* of ΔC . ACTESTs do not evaluate the absolute security of a system configuration (e.g., C or C')—absolute security is hard to measure [15, 69]. Moreover, given the common distribution practice of secure and deployable images (e.g., those on DockerHub [28]), sysadmins can use ACTESTs to evaluate their configurations against the vendor-distributed configurations.

2.4 ACTEST Generation

We present a general technique to help developers build ACTESTs for their programs. The idea is to transform an original program into an ACTEST. To achieve this, there are four challenges to address:

- How to transform an original program into a *safe* ACTEST to run in production? Since the ACTEST will be run in production, it cannot incur side effects to the production system or the environment.
- How to make ACTESTs performance-efficient? ACTESTs may need to test a large number of requests (the combination space of subjects, objects and actions can be large) and so take a long time to run.
- How to generate ACTESTs with minimal input from sysadmins to increase usability?
- How to present the test results in a way that is easy for sysadmins to validate? A small configuration change may lead to a large number of impacted requests.

To generate safe ACTESTs, we adopt virtualization technologies to isolate the test program from the production environment while using file system copy-on-write to allow safe data sharing. As many of today's systems have been deployed with containers, we build an automated tool that can transform the widely-used container specifications, Dockerfiles, into ACTESTs in the form of Docker images.

To improve test performance, we propose trimming methods to accelerate the program execution. The key idea is that access-control checks are usually run before the main task in a program execution. Therefore, the main task execution can be removed without impacting access-control checks. We developed a dynamic analysis tool to trace the program execution path and extract the candidate code to trim.

To minimize sysadmins' effort and improve usability, we design two methods for sysadmins to specify how to generate test requests, namely using access logs and enabling sysadmins to specify subjects, objects and actions.

To ease validation, we present an aggregation technique to group impacted requests by attributes of subjects, objects and actions, so the validation efforts are largely reduced.

2.4.1 Transforming Programs into *Safe* ACTESTS

The best way to achieve high fidelity is to reuse the code logic of the target system program to test access-control configurations. The test program can take the configurations and data of the deployed system and then evaluate the system behavior. However, the original program cannot be directly used because it is unsafe to run it in the production environment. First, running a copy of the original program could have side effects to the production instances of the same program. It may write to the same data files or content for the same network ports. Second, the tests may have side effects to the production instances of other programs. For example, it may write to a database or make an RPC call to a remote server. Third, running such tests in production may have unbounded performance impact on the production instances on the same physical machine.

To transform an original program into a safe test case, one could take two possible approaches. The first one is an extracting approach—starting with the access-control related code in the original program and extract only the code necessary for access-control but not other unrelated functionalities. This can be done either manually or using automated program analysis. We conducted a feasibility study of three popular programs: `httpd`, `lighttpd` and `HBase` and found it difficult to do either manually or with a program analysis tool. From these programs, we manually identified 9-30 access-control check functions, these check functions take in total 53-76 parameters and the parameters can be defined in up to 321 program locations. The depended code path can be exponential to the parameter definition locations. Therefore, it would take large human effort to extract all the depended code manually. It would also require accurate and scalable slicing techniques for an automated tool to do the extraction. As pointed in [31], such static slicing usually cannot precisely extract the depended code path and cannot scale to a large code region as in real-world programs.

The second approach is to trim the original program down. Developers can take the whole original program as a starting point and trim the dangerous side effects it may cause to

production systems. We take this approach and build a tool to apply transformations to make the original program safe for in-production test.

Isolation

Three types of isolation are necessary to make ACTESTs safe to run in production:

- *Namespace isolation:* ACTESTs should only see their own namespace for PID, file system, network and IPC. This can avoid ACTESTs from interfering with production processes, like writing to the same file, contending for the same port and sending a signal to a production process. Namespace isolation is necessary based on the assumption that ACTESTs may collocate with production processes on the same host machine. As discussed before, allocating host machine exclusively for testing can be prohibitively costly.
- *Network isolation:* An ACTEST may need to communicate with another process to obtain the data it needs. However, it is problematic to let the ACTEST talk with a production process as this may cause an expected state change of the production process, like writing to a database. To address this, the ACTEST should only communicate with another ACTEST for the target production process. This makes it necessary to place all ACTESTs into an isolated subnetwork so that they can only communicate with each other but can neither send nor receive packets from production processes. Note different ACTESTs may run on different host machines and so it is not enough to have the aforementioned namespace isolation for the network on a single machine.
- *Performance isolation:* Running ACTESTs can occupy a large amount of CPU time and memory space, which can affect largely the performance of the collocated production processes. To control the performance impact to an expected level, it is necessary to allow sysadmins to set a usage limit of CPU and memory for ACTESTs.

We adopt existing system deployment techniques in cloud environment to achieve isolation. In cloud environment, systems are usually packed into virtualized environments (e.g.

virtual machine or container images) to be deployed. We adopt a similar approach to generate ACTESTS. First, ACTESTS are run in either a virtual machine or container to make namespace isolation. Second, multiple ACTESTS are placed in a single virtualized subnetwork (i.e. Virtual LAN) so that they can communicate with each other but not with production processes out of the subnetwork. Third, the resource allocation functionality of virtual machine and container is leveraged for sysadmins to allocate resources for ACTESTS.

Safe Data Sharing

ACTESTS need to access both production configurations and data to evaluate access-control decisions. Many systems have access-control configurations coupled with production data. For instance, in a web server system, the web page's file permissions are stored together with the web page content. In a database system, user permissions are stored in multiple tables.

Since it is prohibitively costly to maintain a whole copy of the production data, our design lets ACTESTS directly access production data. Therefore, we have to make sure ACTESTS do not pollute the production data.

We use Copy-On-Write techniques [61, 88] to address the challenge. The production configurations and data are directly shared to ACTESTS for all read requests, while a copy is made when ACTESTS try to modify any production configurations and data. The modification is only applied to the copied version, which is invisible to production systems, so it does not affect the execution of production systems. Any future read of ACTESTS will also go to the copied version so ACTESTS can always work with the newest data they generated.

The copy-on-write mechanism may slow down ACTESTS' execution as write to files may need to copy them first. Luckily, read operations dominate most server workloads [74, 7, 13, 43, 109, 34]. In addition, we will present in §2.4.2 our method for removing I/O related code that is irrelevant to access-control from the original program.

Implementation

We develop a tool to transform existing containers into ACTESTs for container-based production environments [24, 25, 87, 14]. Our tool takes a container specification, i.e. Dockerfile, as the input and generates an ACTEST in the form of a Docker image. This takes two steps:

First, an overlay network [29] is set up for the image to make it run in an isolated subnet. Similar techniques, like VXLAN [52], exist for virtual machines. The name and IP address of the overlay network are generated as configurable parameters that allow sysadmins to set later – this is necessary for sysadmins to flexibly place multiple ACTESTs in a single subnet.

Second, an overlayfs file system [61] is configured to enable Copy-on-Write of production configurations and data. Overlayfs is a union mount file system provided in Linux. It combines two file systems – an upper layer and a lower layer, into one. Every read to the combined file system first goes to the upper layer and then the lower layer. Every write only goes to the upper layer. And if the write target does not exist in the upper layer but only in the lower layer, it will be copied from the lower layer to the upper layer to be written. The lower layer would never be modified. In our implementation, the source production configurations and data are mount as a lower layer and a temporary file system is mount as an upper layer.

Our tool focuses on containers, but similar tools can be built for transforming existing virtual machines. The relied techniques are also available in virtual machines: VXLAN [52] can be set for network isolation and overlayfs [61] can also be set for file system Copy-on-Write.

2.4.2 Making ACTEST Performance-efficient

Running the whole original program for testing can take a long time and consume a large number of resources. We propose a new technique to trim the program to accelerate the execution. The main idea is that most part of the target program is not about access-control checking. Therefore, these unrelated parts can be trimmed to accelerate the execution.

```

//entry handler of requests                                httpd-2.4.46
void ap_process_async_request(request_rec *r)
{
    ...
    // do access checks
    access_status = ap_process_request_internal(r);
    if (access_status == OK) {
        //sub-handler that executes the request task
        access_status = ap_invoke_handler(r);
    }
    ...
}
    access_status = ap_run_access_checker(r);

```

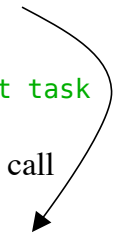


Figure 2.3. Request handling code pattern. Each request is processed by an entry handler, which first checks access and then calls a sub-handler to execute the specific task. Basic trimming removes the sub-handler call entirely. Note: it may not be always correct because the sub-handler could do additional task-specific checks.

Basic Idea

Intuitively, a server program has a request handler that is executed for every incoming request. Inside the request handler, it should perform access-control checks in the beginning and then only execute the actual request task when the checks pass. We validate this intuition with five widely used server programs, including Httpd, Nginx, Mediawiki, vsftpd and MySQL and it holds with all of them. Specifically, all these programs share the same code pattern as shown in Figure 2.3. Every time a request comes, the entrance request handler first performs some preprocessing as well as access-control checks and then dispatches the request to a specific sub-handler.

Following the intuition, our first attempt is to find the function call to the sub-handler and then removing it entirely. The sub-handler call is easy to find and remove for all five programs we studied. However, removing the sub-handler could lead to wrong access-control results. The reason is that the sub-handler also performs additional task-specific access-control checks. Our

| Source code of Httpd-2.4.46 | Allowed trace | Denied trace | Diff | Trimmed code in AC'TEST |
|--|--|---|--|--|
| <pre> 4722 int default_handler(request_rec *r) 4723 { ... 4804 if ((status = file_open(filename,...)) 4805 != APR_SUCCESS) { 4806 log_rerror("file permissions deny: "...); 4807 return HTTP_FORBIDDEN; 4808 } 4809 ap_update_mtime(r, r->finfo.mtime); 4810 ap_set_last_modified(r); ... 4831 apr_brigade_insert_file(fd,...); ... 4849 return OK; 4850 } </pre> | <pre> 4722 4723 ... 4804 4805 ... 4808 4809 4810 ... 4831 ... 4849 4850 </pre> | <pre> 4722 4723 ... 4804 4805 4806 4807 4808 </pre> | <pre> > 4806 > 4807 < 4809 < 4810 < ... < 4831 < ... < 4849 </pre> | <pre> int default_handler(request_rec *r) { ... if ((status = file_open(filename,...)) != APR_SUCCESS) { return HTTP_FORBIDDEN; } return OK; } </pre> |

Figure 2.4. Advanced trimming by comparing allowed and denied execution traces. Allowed trace: execution trace of an allowed access; Denied trace: execution trace of a denied access. The trimmed code excludes code lines in the diff except return statements to make sure the function returns correctly.

study shows that there are two types of task-specific checks: 1) file permissions—programs typically check file permissions only when they try to open files; 2) task-specific configurations – depending on what kind of tasks to execute, a sub-handler may have further access-control checks differing from the global check in the entrance request handler. For example, different HTTP requests in Apache need to check for different permissions like for proxy redirection, cgi script execution and directory listing.

We experimented with this approach. As we will present in §2.5.3, this approach only works for one system and for three other systems, the trimmed programs can generate false access results so that only 46.4% - 52.7% of the result changes can be detected. For the remaining one system, the trimmed programs can detect no result changes.

Advanced Trimming

In order to have the trimmed program to generate the correct access-control results, it is necessary to delve into each sub-handler to find out the last access-control checks. One possible way to do this is to statically analyze the code either manually or with static analysis tools. We investigated this approach and found both difficult if not impossible. First, the sub-handlers perform the major functionality of a program and so the execution path explodes after the sub-handlers. It requires large efforts and is not scalable for developers to manually analyze. Second, there are many usages of function pointers and polymorphism inside the sub-handlers, which makes it hard to build an effective static analysis tool. The static analysis technique needed is interprocedural and whole-program-wide, which brings challenges on both accuracy and scalability.

Therefore, we propose a dynamic analysis approach to help developers trim their programs for ACTESTS. The idea is to run two instances of a program with two different configurations: one allowed access and one denied access to the same request. This request is issued to each of the instances and two execution traces are collected: one for access allow (allowed trace) and one for access deny (denied trace). Figure 2.4 shows an example of an allowed and denied

trace. By comparing the two execution traces, we can find code snippets that are only executed in one of the traces but not both. Most of these code snippets contain no access-control checks because they are only executed in one trace while access-control checks need to be executed in both traces. Therefore, the found code snippets can be safely trimmed without affecting access-control results.

We build a tool to implement this idea. Our tool first instruments a target program and runs it with different configurations provided by developers to obtain an allowed and a denied traces. Then our tool compares the two traces to find the trace diff that only exists in the allowed trace but not the denied trace. Our tool maps the trace back to source code to generate the candidate code snippet for trimming.

Our tool adopts code coverage analysis for tracing. Specifically, our current implementation uses gcov [40] for tracing C/C++ programs and uses Xdebug [123] for tracing PHP programs. Our tool can be extended to other languages as code coverage analysis is a general technique available for most programming languages.

Our tool relies on developers for two pieces of information. First, it relies on developers to provide a few test requests as well as the corresponding configurations that allow and deny it. The provided test requests may not get all sub-handlers executed and so some of them may not be trimmed. However, this is acceptable because this only misses some acceleration opportunities and brings no other harm to the access results. Second, it also needs developers to validate and tune the generated code snippet to trim. The generated code snippets may still contain some access checks if the denied trace does not include the last check in the execution path. In this case, it requires developers' knowledge on the code semantics to prevent the remaining access check from being trimmed. From our experience, the generated code snippets are usually short (less than 100 LOC) and it is easy to tune.

2.4.3 Minimizing Sysadmin Involvement

ACTEST needs to generate test requests as test input to exercise access-control configurations. We divide this challenge into generation policy and mechanism.

- The generation policy decides what requests $\langle s, o, a \rangle$ should be generated, which can vary with concrete system deployments and configuration changes made. We provided policy templates for sysadmins to specify the requests they want to test.
- The generation mechanism is the implementation that turns an $\langle s, o, a \rangle$ into an actual access request and sends it to the testing program, which is usually program specific. Therefore, we provide a mechanism skeleton and leave developers to implement the specific request generation logic.

Policy Template

ACTESTs provide two policy templates for sysadmins to specify what requests to generate. First, the first policy template accepts a set of previous access logs along with an annotation of the subject, object and action fields. The template then parses access logs into $\langle s, o, a \rangle$ tuples and uses them as the test requests. As shown in previous work [125], existing systems' access logs usually record the essential information of subject, object and action. Such an access log policy then can help ACTESTs cover the common requests. Second, the second policy template accepts the locations to collect subjects, objects and actions. Such locations include database tables to collect users (S), directories to collect all file (URL) (O) paths and self-written lists to collect all actions (A). The template then traverses the Cartesian product of them $S \times O \times A$ to synthesize the test requests. Depending on sysadmins' specification, the synthesized requests can be either the one related to a change or be all possible requests.

Mechanism Skeleton

Our mechanism skeleton provides two general features for developers. The first one is simulating requests from various IP addresses. This is commonly needed as many access-controls

are based on IP addresses, like blocking certain ones. We implement this with the AnyIP [60] feature provided by Linux Kernel. This feature enables our testing container binding its the whole IPv4 or IPv6 address space to its loopback device. In this case, a request sent to any third-party IP address will be routed back to itself and will be recognized as from the third-party IP address. This enables the server program to do access-control with the correct source IP. The second one is concurrent requests. Sending concurrent requests is essential for accelerating the testing, besides the trimming method we proposed. We implement a concurrency skeleton which adopts multiple processes and multiple threads to sends requests. We take into account the potential dependencies between requests from the same IP or the same user so that these requests will not be concurrently sent.

2.4.4 Presenting ACTEST Results

The number of impacted requests can be large even with a small configuration change. Such examples include allowing access to a directory with many files. The configuration change may only be a small change to the directory permission, but the impacted requests include the ones to all the files under the directory. Asking sysadmins to validate every access is time-consuming.

ACTESTs provide two ways to present the impacted requests in an efficient way for sysadmins to validate. First, ACTESTs aggregate impacted requests based on the common attributes of subject (user name and group), object (directory name and file suffix) and action (action type). For example, if requests to all files under a directory are impacted, ACTESTs only show the directory name to sysadmins. ACTESTs also show the file suffix impacted under the directory, which may raise sysadmins' attention to dig more. The aggregation can be easily extended to other attribute types and makes the validation efficient. Second, ACTESTs also rank the impacted requests based on their likelihood to be vulnerable. This allows sysadmins to only validate top ranked impacts if they have limited time. Currently, ACTESTs only use simple heuristics, like if a file name starts with a dot (i.e. hidden system files), to calculate the likelihood.

We keep better ranking methods as our future work.

2.5 Evaluation

2.5.1 Methodology and Testing Environment

We conduct three sets of experiments to measure ACTESTs’ effectiveness and efficiency:

1. Detecting Real-world vulnerabilities: We use ACTESTs to analyze the impacts of configuration changes in public Docker images and inspect how many of them are vulnerable.
2. Controlled experiments: We replicate five real-world system environments, including Wikipedia and the web proxy of a commercial company. We inject access-control configuration changes to them and use ACTESTs to detect the change impacts.
3. Performance: We run ACTESTs with different trimming options to test the real systems as in the controlled experiments and compare their performance.

We run our experiments on an Intel Xeon 2.84GHZ machine with 4 cores, 16 GB memory and 2TB HDD.

2.5.2 Detecting Real-world Vulnerabilities

This experiment aims to evaluate how effective ACTESTs are when being used to detect misconfiguration vulnerabilities. We choose five widely-used web systems and download their container images from Dockerhub [28] as the evaluation targets. As shown in Table 2.1, the chosen systems include Wordpress [122], Drupal [32], Joomla [55], Dokuwiki [30] and Mediawiki [117], of which each was downloaded for more than 10 million times on Dockerhub. We mainly choose web systems as they are the only publicly available systems we can find that have access-control configurations.

For each system, we download the container images updated recently (in one year) and extract their configurations as well as data as the testing targets. These configurations include

Table 2.1. New vulnerabilities detected by ACTESTs in public Docker images of popular web systems. 25 of the detected vulnerabilities have been confirmed and 19 of them have been fixed by the image maintainers.

| Systems | # of Images | Vulnerable images | Vulnerabilities |
|-----------|-------------|-------------------|-----------------|
| Dokuwiki | 57 | 19 | 67 |
| Mediawiki | 47 | 23 | 56 |
| Wordpress | 42 | 18 | 28 |
| Drupal | 29 | 8 | 11 |
| Joomla | 18 | 4 | 6 |
| Total | 193 | 72 | 168 |

Apache configurations, Nginx configurations and file permissions of web pages. We use the official images’ configurations as the initial version and the third-party images’ configurations as the changed version. We generate ACTESTs for each systems and use the generated ACTESTs to test the change impact of each changed version against the initial version. As we do not have historical logs for these systems, we set ACTEST to synthesize requests regarding to all the combinations of users, actions and resources.

These experiments focus on risky change impacts—requests that are not allowed in the official images but are allowed in the third-party images. As not all reported access changes are vulnerabilities, we manually validate if they are vulnerabilities or not. We use a conservative standard to make the decision: if the same type of allowed requests is identified as a vulnerability in *other* systems or in public vulnerability databases, we count the corresponding change as a vulnerability. When multiple requests are impacted by the same change (e.g. multiple file requests are allowed because of a change in their parent directory), we only count it once. We also report our detected vulnerabilities to image maintainers for confirmation and for them to do a fix timely.

Results

With the risky changes detected by ACTESTs, we identified 168 vulnerabilities from 72 public Docker images. We reported 135 vulnerabilities to image maintainers with their contacts

available. So far, 25 of the vulnerabilities have been confirmed and 19 of them have been fixed by the image maintainers.

In total, ACTESTs detected 874 risky access changes from 193 tested images. 706 of the changes are not identified as vulnerabilities by us but they can be potentially vulnerable. These changes expose resources that are newly-added and system-specific, and we cannot decide if they expose sensitive information or not. As discussed before, our identification of vulnerabilities is conservative. We identify access changes as vulnerabilities only if the same type of allowed requests is identified as vulnerabilities in *other* systems before. In practical usage, the changes that we do not identify as vulnerabilities are also worth reporting to sysadmins for validation. These requests are exposed for the first time. As we have presented in §2.2 Figure 2.1, the newly-introduced resources can be dangerous to expose and sysadmins may not be aware of the exposure.

Table 2.2 shows the common types of our detected vulnerabilities. The most common type is the exposure of dangerous web interfaces. Such examples include the exposure of PHP interfaces under directory “phpunit” and “vendor” to public access. These interfaces may disclose sensitive data and allow arbitrary code execution [42, 41]. Our experiment results show that these interfaces are located in third-party extensions and are introduced when sysadmins install the extensions. As sysadmins have no knowledge of what a third-party extension may introduce, it is hard for them to be aware of the exposure of these interfaces. However, with ACTESTs reporting the change impacts, sysadmins can learn about the changed requests when they install an extension and can disable the access before they are exploited by attackers.

2.5.3 Detecting the Impacts of Injected Changes

This experiment aims to evaluate how effectively ACTESTs can detect the impacted requests of various access-control configuration changes in real-world deployed systems. We collect configurations and data from five real-world deployed systems, including Wikipedia, a web proxy for a commercial company with millions of users, a department course website, a

Table 2.2. Common types of vulnerabilities detected.

| Types of vulnerability | Examples | # of vulnerability |
|------------------------------------|---|--------------------|
| Dangerous interface exposure | Internal PHP interfaces for testing such as <code>phpunit/*</code> are exposed. | 46 |
| Sensitive system settings exposure | System settings like <code>composer.json</code> and <code>.htaccess</code> are exposed. | 40 |
| Sensitive metadata exposure | Version control data like <code>.git/*</code> are exposed | 16 16 |
| Dangerous access method enabled | HTTP diagnostic method TRACE is enabled for public access. | 14 |
| Database files. exposure | *.sql files are allowed to public download. | 10 10 |

Table 2.3. Real-world systems and access logs collected for evaluation. For Wikipedia, only edit requests are collected but read requests are not (see Wikipedia’s logging policy [119]). This does not affect ACTESTS as all Wikipedia pages are publicly readable.

| Real-world systems | System type | Access log time span | # Requests in logs |
|--------------------|-------------------|----------------------|--------------------|
| Wikipedia | Wiki Application | 18 years | 217 thousand |
| Comp-A | Nginx proxy | 1.4 days | 10 million |
| Course Center | Nginx web server | 1 year | 3.8 million |
| Group | Apache web server | 3 years | 9 million |
| | | 1 year | 2.5 million |

research center website and a research group website. We also collect access logs from them to use as input for ACTESTS’ test request generation. Table 2.3 shows the time span of the logs as well as the number of requests in them for each system. For Wikipedia, because the whole website is too big for our testing environment, we use a part of it, called cowiki, as our testing target. Based on the collected configurations, we randomly inject configuration changes and then run ACTESTS to detect their impacts.

The injected configuration changes are generated based on the access-control mechanisms each system is configured with. Table 2.4 shows the three types of access-control mechanisms used in the target systems and the corresponding types of changes injected to them. Note these change types are also used in previous work [125]. We use two change types from previous work

Table 2.4. Different types of access-control configuration changes injected in our experiments. These changes are also used in previous work [125].

| Access-control | Injected Changes | Systems Applied |
|--------------------|-------------------------|-----------------------|
| File permission | allow/block file access | Course, Group, Center |
| Web server | allow/block URL | Course, Group, Center |
| access-control | allow/block IP | Comp-A |
| App access-control | allow/block pages edit | Wikipedia |

and add a new change type for web application permissions. For each change type, a number of the change targets (e.g. file, directory, URL) are randomly chosen to apply the change. The number of the change targets is set to be 10% of all the available resources.

Two sets of experiments are conducted to measure ACTESTS’ effectiveness under different settings: different trimming methods and different ways to generate test inputs.

Different trimming methods

We use basic trimming and advanced trimming methods as discussed in §2.4.2 to generate ACTESTS for different systems. Then we use randomly injected changes to evaluate their effectiveness on detecting the impacted requests.

To better understand the limit of trimming methods alone and avoid the disturbance of historical logs, synthesizing requests is used to generate test requests with good coverage of subjects, objects and actions. For Wikipedia, users and pages in database tables are automatically extracted to synthesize requests. For Comp-A, URLs and IPs in the configuration file are automatically extracted to generate requests. For Course, Center and Group, the combination of all web pages and HTTP access methods are iterated to synthesize all requests.

Results

The advanced trimming works effectively on all systems in terms of detecting impacted requests. Table 2.5 shows that with advanced trimming ACTESTS can detect all the 303-4861 impacted requests of injected changes in different systems. This indicates that the advanced trimming keeps all the necessary access-control checks while removing the computation and

Table 2.5. Detecting change impacts with ACTESTs generated by different trimming methods.

| Real-world systems | # of Total injections | # (%) of detection | |
|--------------------|-----------------------|--------------------|---------------|
| | | Basic trim | Advanced trim |
| Wikipedia | 930 | 0 (0) | 930 (1) |
| Comp-A | 4861 | 4861 (1) | 4861 (1) |
| Course | 303 | 142 (0.47) | 303 (1) |
| Center | 710 | 329 (0.46) | 710 (1) |
| Group | 178 | 94 (0.53) | 178 (1) |

I/O unrelated to access checks. Therefore, the generated ACTESTs can always return correct access-control results as well as detect correct impacts.

The basic trimming’s effectiveness varies on different systems. As shown in Table 2.5, Wikipedia’s ACTEST with basic trimming can detect none of the impacted requests, while Comp-A’s ACTEST with basic trimming can detect all the impacted requests. For Course, Center and Group, the ACTESTs can detect 46%-53% of the impacted requests. The reason is that different systems are configured with different access-control mechanisms. Wikipedia’s access-control mechanism is implemented by PHP code, which is done mostly in request sub-handlers that are removed by the basic trimming (cf. Section 2.4.2). Comp-A only uses Nginx’s access-control, which is performed out of request sub-handlers and so is not removed by the basic trimming. For Course, Center and Group, they rely on both Nginx/Apache web server’s access-control and file permissions checks. Nginx/Apache’s access checks are kept with the basic trimming but file permission checks are all removed by the basic trimming.

Different test input generations

We evaluate ACTESTs’ effectiveness with test inputs generated by replaying historical logs and synthesizing requests separately. For replaying logs, we use the one we collected for the five real-world systems, as shown in Table 2.3. For synthesizing requests, we specify the corresponding source of subjects, objects and actions (e.g., database tables and file directories) for ACTESTs to automatically extract the possible value and iterate all the combinations, as

Table 2.6. Detecting change impacts with different ways of generating test inputs.

| Real-world systems | # of total injections | # (%) of detection | |
|--------------------|-----------------------|--------------------|---------------------|
| | | Replay logs | Synthesize requests |
| Wikipedia | 930 | 739(0.80) | 930(1) |
| Comp-A | 4861 | 0(0) | 4861(1) |
| Course | 303 | 242(0.80) | 303(1) |
| Center | 710 | 106(0.15) | 710(1) |
| Group | 178 | 19(0.11) | 178(1) |

discussed in Section 2.5.3. Advanced trimming is used to avoid the inaccuracy introduced by trimming method.

Results

By synthesizing requests, ACTESTs detect all the impacts of the injected changes, as shown in Table 2.6. This illustrates that access synthesis can generate a comprehensive set of test requests. We admit that this depends on sysadmins' knowledge on the subjects, objects and actions. From our experience, most of them can be specified with database tables and file directories, and ACTESTs can automatically extract the possible values for synthesis.

By replaying historical logs, ACTESTs can detect up to 80% of the injected change impacts, as shown in Table 2.6. ACTESTs work best for Wikipedia and Course, because their logs have a good coverage of possible requests. For Wikipedia, the logs are of 18-years long and cover requests to most pages. For Course, the logs are 1-year span and the web pages host is also about classes in the last one year; so the logs have a good coverage of web pages. For Center and Group, the logs fail to cover many obsolete web pages that were never modified in the last five years. For Comp-A, replaying log can detect no change, as the time span is only 1.4 days. This illustrates that for systems with actively-accessed resources and complete historical logs, sysadmins may choose access logs to generate test requests for ACTESTs; otherwise, they may choose to synthesize requests to have a better coverage.

2.5.4 Performance

We measured how long time it takes to run ACTESTs against the five real systems. Synthesizing requests are used to make sure ACTESTs have a good coverage of the possible requests. The results show that for Comp-A, Course, Center and Group, it only takes less than 10 minutes to finish all the tests. For Wikipedia, it takes a relatively longer time, 1.2 hours. The main bottleneck for Wikipedia is the executions and database queries that are not trimmed. We plan to further optimize the performance in the future.

We compared ACTESTs' performance with different trimming methods: no trim, basic trim and advanced trim. No trim means the original program runs the whole request handling code to perform the test, which is our baseline. Basic trim only keeps the global access checking and removes the majority of request handling code. Basic trim does not ensure correct access-control results all the time but can be treated as the upper bound performance that may be achieved by trimming. Advanced trim precisely removes the code for I/O and computation that are not related to access checking and always ensures correct access-control results.

As shown in Figure 2.5, advanced trim can achieve comparable speedup as basic trim (upper bound). Advanced trim reduced 9.09%-98.61% running time compared with no trim, while basic trim reduced 9.09%-98.64%. Both advanced trim and basic trim work best (98.61% and 98.64% reduction) on the Course system. The reason is that Course has more large files in PDF and MP4 formats. This takes a lot of time to do I/O with no trim, while advanced trim and basic trim eliminate the necessity to do I/O. Both advanced trim and basic trim have only a small speedup on the Comp-A system. This is because the tested system is a web proxy, which only redirects requests but does not handle the requests. As a result, each request handling in it is already simple and fast, and there is not much code for I/O or computation to trim.

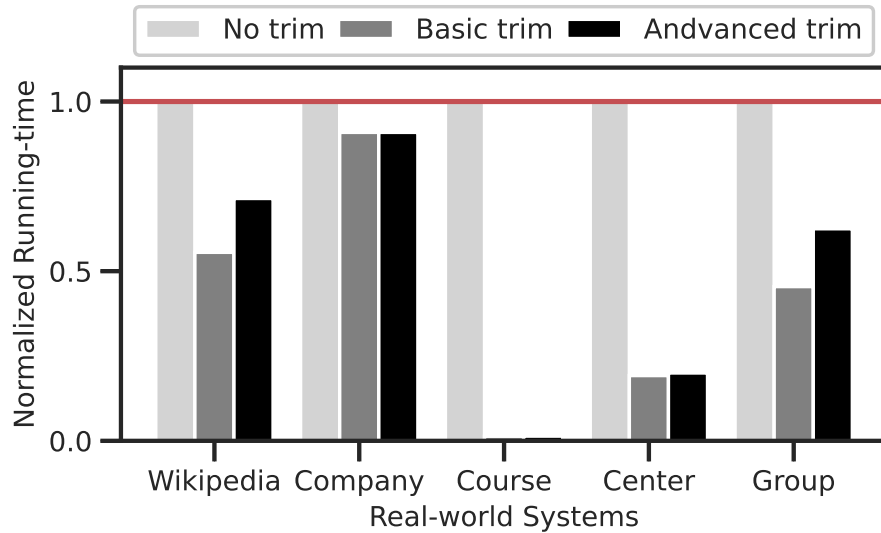


Figure 2.5. Normalized test running-time by different trim methods.

2.6 Discussions

The generation and distribution of ACTESTs can be merged into software vendors’ release process of their main program. When a new version of the main program is ready to release, vendors take two steps to generate a new ACTEST for it. First, they provide a new Dockerfile and use our transformation tool to turn it into an ACTEST container image. Note that most of the time previous Dockerfile can be reused and it has been a common practice for vendors to compose Dockerfiles to build Docker images for release [24, 25, 87, 14]. Second, they run our trimming tools to generate a trimmed program and validate if the trimmed program keeps the correct access-control checking. This process is not fully automated. However, from our experience, it does not take much effort to validate if access checks are mistakenly trimmed or not. And this needs not to be done frequently as access-control checks do not evolve often. Some access-control checks in Apache httpd have not been changed for 19 years [6].

We plan to further automate our trimming method (§2.4.2). Our current trimming tool is based on dynamic analysis, which is neither sound nor complete and thus needs developers’ validation. We plan to further incorporate static analysis [138] to make it sound. With a dynamic

analysis extracting the candidate code to trim that likely calls no access check, a static analysis can be adopted to make sure the candidate code does not call any access checks. The incompleteness of trimming is acceptable as it only affects testing performance, but not correctness. Developers can provide more inputs (e.g. configurations and test requests) to make the dynamic analysis cover more execution paths.

For test request generation, the choice between using access logs and synthesizing requests can be made based on the specific scenario. Access logs provide a convenient way to generate test requests, but the effectiveness of testing depends on the comprehensiveness and coverage of historical requests, On the other hand, test request synthesis provides a complete set of test inputs. It requires sysadmins to specify the subjects, objects and actions.

2.7 Acknowledgement

This chapter, in part, is a reprint of the material under submission. Xiang, Chengcheng; Mugnier, Eric; Nguyen, Nathaniel; Huang, Haochen; Zhong, Li; Zhou, Yuanyuan; Xu, Tianyin. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Inferring Access-control Behavior Change with P-DIFF

3.1 Introduction

3.1.1 Motivation

Despite a number of efforts on testing and verifying access control configurations [97, 35, 63, 23, 10, 11, 70], it is still prohibitively difficult to eliminate all access control misconfigurations in real-world systems (§4). Specifically, state-of-the-art detection tools for access control misconfiguration [23, 10, 11] mostly detect inconsistencies of configurations and cannot understand configurations at a higher policy level. As noted in [23], given frequent configuration changes and their ad-hoc, one-off nature, it is very difficult for automated tools to deduce the exact and complete list of access control misconfigurations.

As a consequence, without continuous validation on access control behavior changes, access control misconfigurations often stay unnoticed for a long time until being exploited by malicious users on an attack. The reason is unlike other types of misconfigurations [130, 132], access control misconfigurations cannot be manifested through observable symptoms (e.g., crashing behavior, dysfunctions, or performance degradation). According to a recent report [62], it takes 206 days on average for US companies to detect a data breach, which is too late for any remedies.

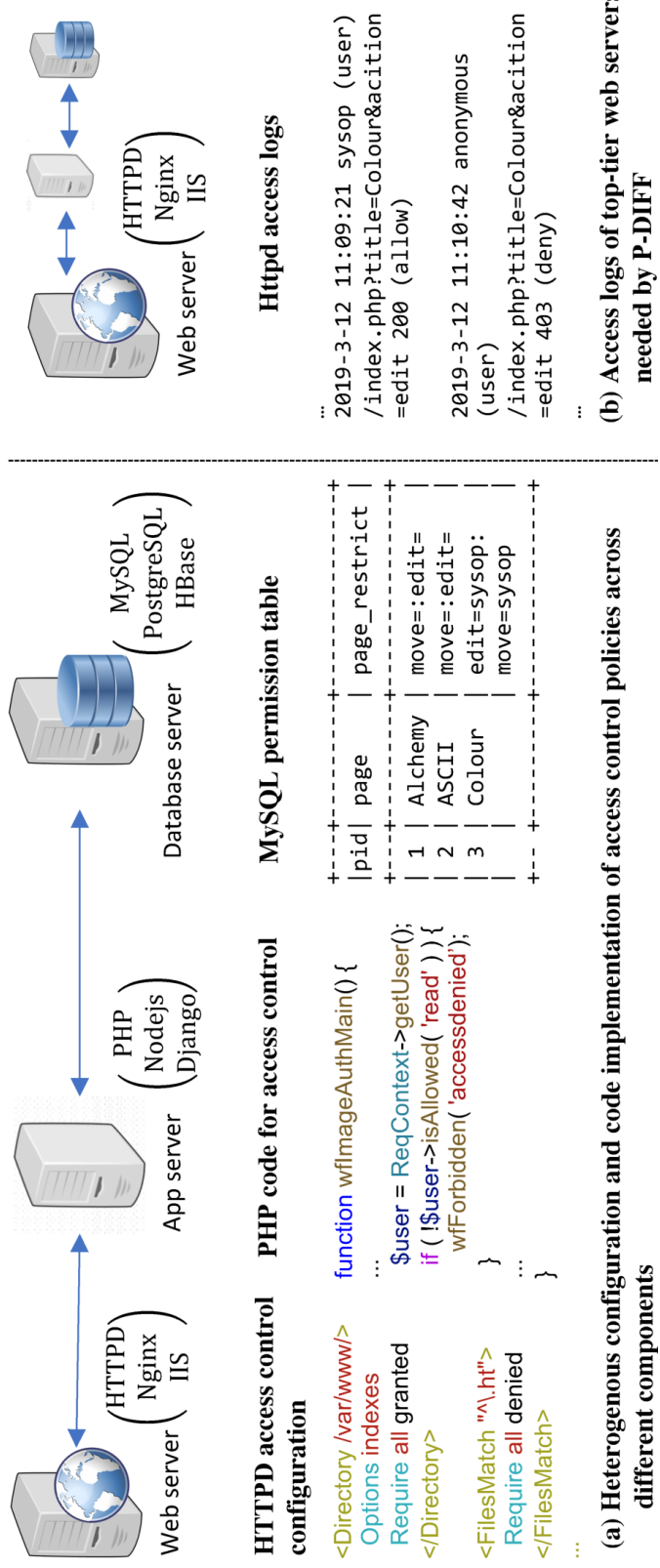


Figure 3.1. Examples of access logs (the inputs of P-DIFF) and the configurations and code implementation of access control in the Wikipedia system. (a) Each component has its own implementation of access control, either in configuration or in code: the heterogeneity makes comprehension and analysis challenging; (b) The access logs record the *end-to-end* access control behavior and has simple and clear semantics. We build our solution, P-DIFF, using only access logs.

Unfortunately, there is little tooling support for access control behavior validation. One potential approach is to track all the configuration changes with a version control system, and let sysadmins validate all the changes. However, there is still a gap between the static access control configurations and the actual running system behavior. In many systems, the access control behavior is determined by multiple heterogeneous components and their configurations. As shown in Figure 3.1, access control is typically implemented in heterogeneous configurations and code across multiple different components in large-scale, complex systems. It is non-trivial (if not impossible) to reason about the end-to-end access control behavior by inspecting the configurations and code statically. This chapter proposes to infer the access control behavior and behavior changes from the access logs that record the end-to-end access results (typically the output by the top-tier components). Once a behavior change is detected, sysadmins will be notified to validate if the change is intended. Once an unintended change is detected, sysadmins can fix the access control timely and avoid potential security incidents (such as data leakage) in the future.

3.1.2 Contributions

This chapter presents P-DIFF, a practical tool for inferring access control behavior and behavior changes from *access logs*. As we will show in §3.2, existing access logs generated by most software systems contain enough information for inferring the changes. Therefore, P-DIFF does not require any modifications to existing systems other than enabling access logs. In addition, P-DIFF also does not require sysadmins to record access control configuration changes, which can be tedious and also sometimes impossible (some changes, e.g., file permissions and network-level firewalls can be done by users or other superusers without sysadmins' awareness).

By detecting access control behavior changes, P-DIFF effectively assists sysadmins in the following two important tasks:

- *Change validation.* When P-DIFF observes changes of access control behavior, it notifies sysadmins with the observed changes. This enables sysadmins to examine the changes to

identify and fix access control misconfigurations that open up security vulnerabilities.

- *Forensic analysis.* For postmortem analysis upon a security incident, P-DIFF can backtrack all the behavior changes related to a malicious access. This provides clues for sysadmins to understand *when* and *what* changes opened up the access. Those clues can help sysadmins narrow down the changes record they need to investigate in their logbooks or the version control systems.

There are two major challenges for designing and implementing P-DIFF. The first challenge is to represent access control behavior in a generic and informative way. As different systems take different access attributes to control access (such as IP, user, and URL), it is necessary to provide the access-decisive attributes together with the behavior changes so that sysadmins can make informative validations. However, it is non-trivial to provide a general abstraction for representing different attributes.

To handle the first challenge, we adopt a decision-tree representation to encode access control behaviors in an organized and condensed rule-like format (referred to as inferred policies). This design decision is made based on two reasons. First, we observed that access control decisions are made by a set of binary decisions (cf. §3.2). Therefore, a decision-tree structure is capable of encoding them. Second, the decisive attributes of access control may inherently have a hierarchical structure, such as the hierarchical namespace of active directory domains, files and directories, IP addresses, and URLs. A tree-based structure is a natural fit to effectively encode those hierarchical attributes.

The second challenge is to handle behavior changes while inferring the decision tree. Existing decision-tree inference algorithms all rely on an assumption that the encoded rules always have constant results (e.g. ALLOW or DENY for access control). However, this assumption is not true in the case of access control rules. For example, a web server administrator disabled public access to a directory “ABC” on May 1st, thus accesses to this directory before May are allowed, and accesses after May 1st are denied. In this case, existing inference algorithms cannot

decide whether “ABC” is a decisive attribute in the rules because of the related result changes (cf. §3.8).

To address the second challenge, we extend the traditional decision tree to support time-series information, referred in this chapter as Time-Changing Decision Tree (TCDT). In TCDT, each rule result is represented as a time series instead of a single binary value (ALLOW or DENY). We design a new TCDT learning algorithm to infer the new decision tree by treating access logs as a sequence of access events ordered by access time instead of an unordered set of events (cf. §3.8). TCDT not only can precisely model access control rules at any given time, but also can capture the evolution of access control rule changes.

We evaluate P-DIFF with datasets collected from five real systems, including *two from industrial companies*. For change validation, P-DIFF can detect 76%–100% of the rule changes with an average precision of 89%. For forensic analysis, P-DIFF can pinpoint the root-cause change that is responsible for permitting the target access in 85%–98% of the evaluated cases.

3.2 Observations of Real-World Access Control Systems

The design of P-DIFF is driven by a few important observations of real-world access control implementations. This section discusses these observations and explains the rationales behind our design decisions.

3.2.1 Access-Control Configurations

Despite the uniform model of access control (e.g., *access-control matrix* [58]), real-world access-control implementations are highly customized to specific applications, resulting in distinct access-control configurations in terms of syntax, semantic, and schema.

First, different software systems implement various access control models. For instance, the Unix file system adopts discretionary access control (DAC) [82] to restrict access to files based on the identity of users and groups. The Apache web server adopts *attribute*-based access control (ABAC) [50], e.g., any access from a certain IP address should be denied (the address is

treated as an attribute). MySQL uses *role*-based access control (RBAC) [89] where privileges are granted by assigning one or more roles to each user. Different access control implementations require distinct access control configurations.

Second, even for the same access control model, different software systems often implement the model differently with customized syntax and formats. For instance, many web servers (e.g., Apache, Nginx, and IIS) adopt the attribute-based access control model; however, each of them implements its own configuration directives and parameters.

The heterogeneity of access control configurations imposes significant challenges for building generic, automated tools to directly interpret and validate configurations. Implementing specific parsers or interpreters for every target software project requires significant engineering and maintenance effort.

3.2.2 Access Logs

We observe that access logs of different software systems tend to have a unified format and are easy to parse. No matter how complex the configurations are, access logs record identical information—the *results* (either ALLOW or DENY) of an access request—represented as a tuple $\langle S, O, A, R \rangle$ where S , O , A , and R denote *subject*, *object*, *action*, and *result* respectively. Within a system, access logs are typically generated by a few unified logging statements.

Table 3.1 shows access log formats of nine different software systems of various types [110, 19, 4, 76, 94, 81]. It shows that most access logs of the studied systems encode the required information. Therefore, it is straightforward to build a uniform parser that takes a few simple format annotations to work with different systems.

Furthermore, the access results (ALLOW or DENY) recorded in the access logs of one software component reflect the *end-to-end* access control behavior which includes the access control of all the downstream components. For example in Figure 3.1, the access logs of the web server reflect the end-to-end access control behavior of the entire Wikipedia system including the web server itself as well as the downstream app server and database server. If the request is

Table 3.1. Information encoded in access logs of different software. S = subject, O = object, A = action, R = result.

| Software | Type | S | O | A | R |
|-------------|-------------|---|---|---|---|
| Apache2 | Web server | Y | Y | Y | Y |
| Jetty | Web server | Y | Y | Y | Y |
| Squid Proxy | Web cache | Y | Y | Y | Y |
| MySQL | Database | Y | Y | Y | Y |
| HDFS | File system | Y | Y | Y | N |
| SELinux | Kernel sec. | Y | Y | Y | Y |
| pureftpd | FTP server | Y | Y | Y | N |
| iptables | Firewall | Y | Y | Y | Y |
| openssh | SSH server | Y | N | N | Y |

denied at the app server or the database server, a DENY will also be recorded in the access log of the top-level web server.

In order to make our solution practical, we explore the feasibility of building a solution on top of the end-to-end access logs only, instead of attempting to understand the complex, heterogeneous access control configurations of every single component in the system (which may not be feasible for closed-source, proprietary software or hardware components).

3.2.3 Access-Control Policies

An access-control *policy* is composed of a set of *rules*. Each rule can be represented by an IF-THEN statement that evaluates a *subject attribute* and an *object attribute*, and the concrete action in order to make a decision (ALLOW or DENY). Within the same subject/object attribute, all the subjects/objects are treated as identical. For instance, an access control rule for a web server could be:

```

1 IF ($method is "GET") THEN
2     IF ($url is "/confidential/*") THEN
3         IF (group($user) is "admin") THEN
4             ALLOW

```

The observation drives the following two design decisions of P-DIFF: (1) We are able to encode the access control policy using a decision tree based on the IF-THEN representation. Certainly, traditional decision trees cannot deal with time-series sequences and cannot encode policy changes. Therefore, we design a novel decision tree named Time-Changing Decision Tree in §3.6; (2) The policy inference should work at the granularity of subject/object *attributes* rather than each individual subject/object for efficiency and scalability.

3.3 Design Decisions

3.3.1 Inferring Policy from Access Logs

There are two information sources from which access control policy changes can be inferred: (1) configuration change history and (2) access logs. We decide to build P-DIFF on top of the access logs for the following considerations.

In our experience, inferring the policy changes from the configuration change history is difficult with non-technical barriers. First, as access control policy is implemented and enforced by multiple components as exemplified in Figure 3.1, the configurations of each component could be managed by different teams (e.g., web server administrators, app developers, and database administrators) without a holistic authoring system [99, 100]. It is technically challenging to keep track of every single configuration change in a large-scale, complex system, not to mention the cultural challenges of enforcing the practice of tracking everything. On the other hand, access logs are the output of the running systems and can be collected without much extra overhead.

Second, as discussed in §3.2.2, access logs reflect the *precise end-to-end access control behavior* of the entire system. We only need to collect the access logs of the top-tier components. Instead, a configuration based solution requires to understand the interactions among multiple components which could be challenging in large-scale, complex systems. Furthermore, due to misconfigurations and bugs in the configuration handling code, the configuration settings may not be consistent with the policy or the mental model of sysadmins [129, 128]. Access logs, as

the output of the access control system, *precisely* records the end-to-end behavior.

Third, comprehension and analysis of various configuration and code are challenging, especially for closed-source, proprietary software and hardware components. As shown in [114], reverse engineering of an application’s access-control configurations is challenging and requires non-trivial human efforts. Oftentimes, understanding the access control configurations of a single component is non-trivial [39], let alone analyzing the interactions between multiple components. On the other hand, access logs have simple and clear semantics, as discussed in §3.2.2.

3.3.2 Using Decision Tree Based Models

As discussed in §3.2.3, an access control policy is essentially a “classifier” that labels an access to be either allowed or denied. There are many machine learning algorithms that can infer such classifiers from data, such as Decision Tree [83], Association rule learning [1], Logistic Regression [45] and Neural Networks [44]. We choose Decision Tree for two reasons:

- Decision trees are easy to interpret. As our goal is to inform human administrators of policy changes for validation, it is important to use a machine learning algorithm that generates human-understandable classifiers. Many algorithms, such as Logistic Regression and Neural Networks, generate classifiers with hard-to-understand weights and thus are not suitable for our use case.
- Decision trees can effectively represent access control policies. As discussed in §3.2.3, an access control policy consists of multiple steps of decision-making and may contain decisions on hierarchical attributes, which can naturally be represented by a decision tree. Figure 3.2 gives an example of how decision trees can effectively represent policies implemented in different access control models, including DAC, RBAC, and ABAC. Some other algorithms, such as Association Rule Learning can only infer correlations between attributes but cannot deal with hierarchical relations of them.

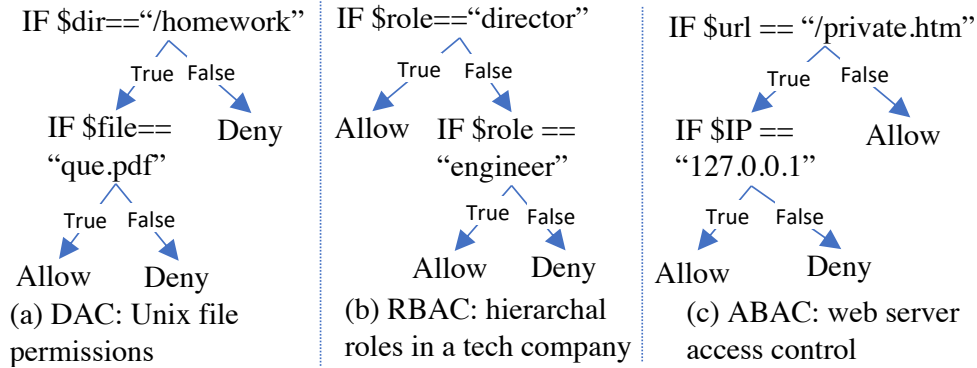


Figure 3.2. Decision tree representations of three access control models. The tree structure can effectively represent access control hierarchies for (a) files, (b) roles, and (c) URLs with multiple decision steps based on different attributes.

3.3.3 Dealing with Sparse Logs

A key challenge of inferring access control policies from access logs is to deal with *sparse logs* that only contain a fraction of all possible requests. Access logs are often sparse because users typically do not request every resource in a system in a short period, which has been reported in prior studies [21] as well as the access logs we collected from real-world deployments (used for evaluation).

Given that access logs are often sparse, one *cannot* assume every possible tuple of $\langle S, O, A, R \rangle$ (§3.2.2) can be observed from historical logs. In other words, one cannot train a classifier with the complete dataset of every $\langle S, O, A, R \rangle$ tuple. To address this challenge, we design a decision tree learning algorithm to infer the result R of unobserved requests from observed access records. As shown in Figure 3.3, our learning algorithm groups observed and unobserved requests and uses the observed request results to infer the group policies. The detailed algorithm is described in §3.7.

3.4 Threat Model

P-DIFF targets on detecting the attacks that aim to steal data by exploiting access control vulnerabilities. The typical cases include that a sysadmin mistakenly over-grant the permissions

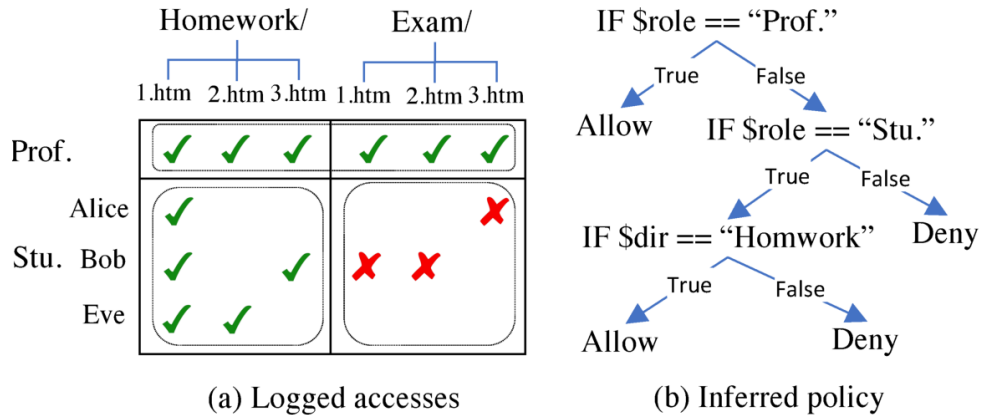


Figure 3.3. Example of sparse accesses to a course website and the decision tree inferred from them. In (a), a green—red mark means the access was allowed—denied. A vacancy means the access did not happen, and thus the policy is not reflected in the accesses. To address it, our learning algorithm groups vacancy with green—red marks and infers a group policy as shown in (b).

of resources, e.g., make a private web page accessible to unexpected users (e.g. anonymous) and then the attackers steal the data by acting as those users. As reported by a number of recent studies [127, 100, 99, 11, 10, 9, 91, 23], such misconfigurations of access control are among the most common and severe security risks in modern information systems.

However, P-DIFF does not target on password attacks or spoofing attacks in which attackers try to guess a password and pretend to be a normal user. In those cases, P-DIFF cannot differentiate a malicious access from a normal access because they have the same access-decisive attribute (i.e. user name) in the log.

The correctness and effectiveness of P-DIFF rely on that the access logs faithfully reflect the system behavior. This is based on three specific assumptions. First, we assume the sysadmins enable access logs in the system settings. This is a reasonable assumption because the default settings of most programs (e.g. web server) have access logs enabled. In addition, since only one log entry needs to be recorded for each access, this will not cause too much performance or storage overhead. Second, we assume that the monitored system can always correctly generate access logs. If the system is modified by attackers so that no log or fake logs are generated, P-DIFF may not be able to detect the behavior changes. Third, we assume that there is no rootkit

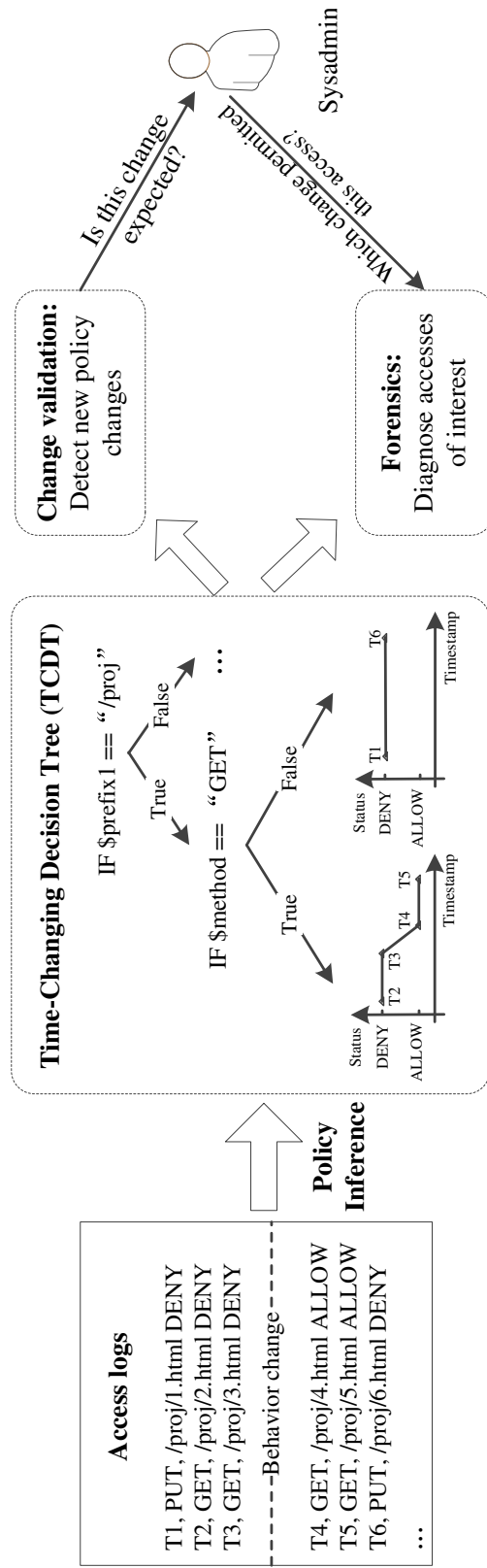


Figure 3.4. The workflow of P-DIFF. P-DIFF infers access control policies from access logs. It maintains the inferred policies in a data structured named Time-Changing Decision Tree which records the entire change history. P-DIFF supports two uses cases, change validation and forensics analysis, as elaborated in §3.5.

or malware at the storage layer which can modify the generated logs.

3.5 P-DIFF Overview

P-DIFF is a tool that infers access control policies from access logs. P-DIFF is able to detect policy changes, when it observes deviation of access results from its known policy. P-DIFF supports two use cases, change validation and forensics, by (1) detecting new policy changes and (2) extracting historical changes.

Figure 3.4 illustrates the end-to-end workflow of P-DIFF. P-DIFF infers access control policies and maintains the policy change history in internal decision-tree like data structure. When P-DIFF observes a new access result, it checks whether or not the result adheres to the latest known policy. If not, P-DIFF treats the violation as a policy change.

Change validation is done by asking sysadmins to validate the policy change whenever a change is detected. To avoid over-alarming, by default, P-DIFF only notifies system admins when an access that previous was denied is now granted. This is the common pattern of illegal accesses caused by over-granting misconfigurations, as discussed in §3.1.1. P-DIFF presents both the changed rules, together with the accesses that were affected by the rule changes to make the validation effective.

For forensic analysis, given an access of interest (e.g., illegal access that caused security incidents like data breaches), P-DIFF searches the change history and identifies the rule change that causes a previously denied access to be granted. If the access is allowed from the beginning, P-DIFF outputs the initial state as the root cause.

P-DIFF needs to address three main challenges: (1) How to effectively maintain the evolution of access control policies? (2) How to accurately learn access control policies from access logs? and (3) How to manage the policy changes?

To address the first challenge, we design a novel data structure named Time-Changing Decision Tree (TCDT) to encode rule changes as time series. In a TCDT, each leaf node of the

tree is no longer associated with a percentage of ALLOW/DENY as in the traditional decision tree, but with the history of all the access results related to the rule. In this way, TCDT not only can precisely model access-control policies at any given time, but also can capture the evolution of policy changes.

For the second challenge, we design a decision-tree-based learning algorithm to automatically infer policies from access logs. As discussed in §3.2.3, access-control policies have an IF-THEN form and inherent namespaces hierarchies, which can be encoded in a decision tree model with each internal node representing a condition associated with an attribute and each path from the root node to a leaf node representing an access control rule.

Addressing the third challenge requires a learning algorithm that can infer access control rules alone with its evolution history. Traditional decision tree learning cannot deal with time-series sequences and thus cannot be used by P-DIFF (cf. §4 for details). We design a new data structure named Time-Changing Decision Tree (TCDT) and the learning algorithm which is capable of learning access control policy changes over time and encoding the change history in a TCDT.

P-DIFF is implemented with Python based on the NumPy and Pandas data analysis libraries [75, 77]. It can be deployed on different platforms that support Python.

3.6 Policy Representation

Access control policies can be naturally represented using a series of IF-THEN statements and be maintained in decision-tree-based data structures (cf. §3.2.3). In this section, we first present how to use traditional decision trees to encode static access control policies. We then present a novel data structure called Time-Changing Decision Tree (TCDT) to encode the evolution history of access control policies.

Decision Tree (DT)

A decision tree is a predictive model that generates a result value based on the observed attributes of an item [83]. It encodes the result generation rule in each tree path that walks from the root node to a leaf node. A DT has two types of nodes: internal nodes and leaf nodes. An internal node encodes a pair of $(AttributeName, AttributeValue)$ and has two outgoing edges corresponding to a predicate whether or not an item with an attribute name has the corresponding attribute value. A leaf node encodes (r, p_r) where $r \in \{\text{True}|\text{False}\}$ is the final decision result and $p_r \in [0, 1]$ is the probability of the result.

A decision tree can encode access control policies by treating subjects (e.g., IP), objects (e.g., file), and actions (e.g., GET or SET) as *access attributes*, and access results (ALLOW or DENY) as result values. Each internal node encodes $(Access\ Attribute\ Name, Access\ Attribute\ Value)$, each leaf node encodes the access result with the probability, and each path from the root node to a leaf node represents an access rule. An access attribute name refers to “IP”, “file” and “GET”, etc., and the corresponding access attribute value is a binary value deciding whether an access is allowed or denied at the point. Figure 3.5 illustrates how an access rule is encoded in a decision tree based on Apache web server’s access control implementation.

The IF-THEN structure of DT can also effectively encode rules with regular expressions. For example, a rule that allows access to “/proj/*/1.htm” can be encoded as:

```
1 IF ($prefix1 is "/proj") THEN
2     IF ($prefix2 is "1.htm") THEN
3         ALLOW
```

Time-Changing Decision Tree (TCDT)

One key limitation of the traditional decision tree is that it cannot work with time-series data where policies change over time, and thus cannot represent access control policy changes. As a result, P-DIFF cannot be built using traditional DT techniques.

In order to maintain the evolution history of access control policies, we design TCDT. A

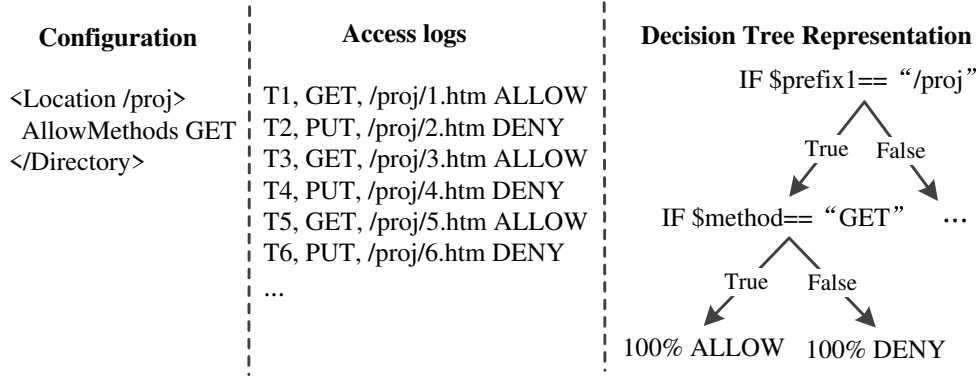


Figure 3.5. An example of access-control policies in configuration files, access logs, and a decision tree. T1-T6 are the timestamps.

TCDT has a similar exterior structure as a traditional DT. Differently, TCDT makes each leaf node encode a time series T :

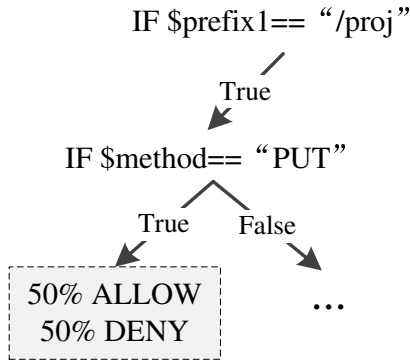
$$T = ((\tau_1, r_1), (\tau_2, r_2), \dots, (\tau_n, r_n)) \quad (3.1)$$

which represents the result values during the time period (r_i represents the result value of the interval $[\tau_i, \tau_{i+1}]$).

Figure 3.6 shows a TCDT generated from access logs in Figure 3.4 and compares it with a traditional DT generated from the same logs. We can see that with the time series in each leaf node, historical rule changes can be easily represented. For applications like continuous access control monitoring, the precision can be largely improved by learning from recent results instead of aggregating all the results, as shown in our evaluation result in §3.10.4. Note that the application of TCDT is not limited to access control monitoring and forensics. The TCDT data structure and the TCDT learning algorithm can be used in other works that need to infer rules from continuously changing time-series data.

TCDT is fundamentally different from Time-Series Decision Tree in machine learning literature [131]. A time-series decision tree classifies a sequence (attributes of a period) into different categories. However, TCDT classifies a “point” (attributes at a single time) into different

Traditional Decision Tree Representation



Time-Changing Decision Tree Representation

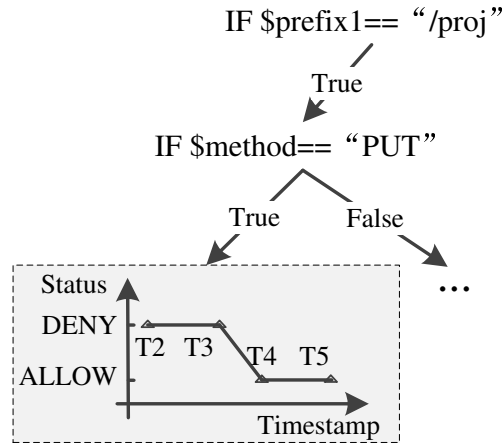


Figure 3.6. A traditional decision tree and a Time-Changing Decision Tree (TCDT) generated from the logs in Figure 3.4. Both decision trees have the same internal nodes; however, in a traditional decision tree, the leaf nodes are associated with proportional results; in a TCDT, the leaf nodes are associated with the time-series results which can be used to represent policy changes.

categories. We position TCDT in the machine learning literature and discuss the related work in detail in §4.

Unknown attributes and values

The attributes and values in both DT and TCDT are limited to the one observed in the access logs. However, when a decision-tree is adopted to classify the access result of a new-coming access, the related attributes and values may not be seen before. In a traditional decision tree, classification will be done with the probability in a leaf node of the False branches. This may cause false classifications and thus miss changes. We address this problem by adjusting our TCDT to explicitly classify such an access as UNKNOWN. When an access is detected as UNKNOWN, P-DIFF will conservatively notify system admins to validate if there is a change. Then P-DIFF will build a new TCDT so that those unknown attributes and values can be encoded. We show in §3.10.6 that building a new TCDT is efficient. It takes 19 minutes to build a TCDT from 320 million log entries collected from the Wikipedia website (cf. Figure 3.12 in §3.10.7).

Table 3.2. Annotations of the log format. P-DIFF requires users to annotate the access log format, which is a one-time effort for a given system.

| Field | Annotation | Semantics |
|----------------------|------------|---|
| Timestamp | %t | Timestamp of each access |
| Hierarchical feature | %h(*) | Features with hierarchical namespace, such as IP address, URL, etc. * is a delimiter character. |
| Normal feature | %n | Non-hierarchical features |
| Access result | %l | ALLOW or DENY |
| Irrelevant | %o | Irrelevant fields |

3.7 Policy Inference

This section describes the algorithms and mechanisms for inferring access control policies from access logs. Note that we do not consider policy changes in this section and thus do not differentiate a traditional decision tree versus a TCDT. We discuss policy change management in §3.8.

3.7.1 Parsing Access Logs

P-DIFF parses logs based on sysadmins’ annotations on the log format. To reduce the manual work and to make it general, P-DIFF does not require detailed annotation of each field’s semantics, such as URL, IP, user and group etc. Instead, P-DIFF abstracts fields into five types: timestamp, hierarchical features, normal features, access results, and other non-related fields. Table 3.2 shows the meaning of each type. P-DIFF recognizes the timestamp for time-series ordering and access results as a label of each access. P-DIFF differentiates hierarchical features and normal features to further exploit the inherent hierarchical namespace of access rules (cf. §3.7.3).

3.7.2 Policy Learning Algorithm

P-DIFF uses a classic decision tree learning algorithm [83] to build the tree structure based on the access logs, described in Algorithm 1. Before starting the algorithm, the access log needs to be transformed into the algorithm’s input format

$$L = \{(x_{i_1}, \dots, x_{i_n}, y) | i \in [1, m]\} \quad (3.2)$$

where $(x_{i_1}, \dots, x_{i_n})$ is the feature vector generated from the access attributes (subject, object, action), y is the prediction label from access result r , and m is the number of log entries. Each subject, object and action attribute could have more than one feature, respectively and each feature is transformed into a unique field in $(x_{i_1}, \dots, x_{i_n})$. For instance, a subject can have features of both username and group, so two fields are created in the feature vector. P-DIFF expands the hierarchical features using the methods described in §3.7.3 and transforms the expanded features into a feature vector with one-hot encoding [118].

Algorithm 1 takes L as input and grows the tree recursively. In each recursive step, the algorithm splits one node into two child nodes, by selecting a feature j and its value x_{i_j} that split L into two subsets with the purest labels, i.e. subsets with as large proportion of ALLOW or DENY as possible. The two generated subsets are

$$L_l = \{(x_{k_1}, \dots, x_{k_n}, y) | k \in [1, m] \wedge x_{k_j} = x_{i_j}\} \quad (3.3)$$

$$L_r = L - L_l \quad (3.4)$$

To find the feature j and its value x_{i_j} , a metric function is adopted to measure the label purity of a set. Traditional DT learning algorithms use either entropy or Gini Impurity [12, 83] as the metric. We will show that those metrics cannot handle policy changes in §3.8, and the new metric we design for P-DIFF to learn TCDDTs.

Algorithm 1. Decision Tree Learning

```
1: function DTL( $L$ ) a
2:    $root \leftarrow treenode()$ 
3:    $i, x_{i_j} \leftarrow best\_split(L)$  b
4:    $L_l, L_r \leftarrow split(L, i, x_{i_j})$  c
5:    $mg \leftarrow metric\_gain(L, L_l, L_r)$  d
6:   if  $mg \neq 0$  then
7:      $root.left \leftarrow DTL(L_l)$ 
8:      $root.right \leftarrow DTL(L_r)$ 
9:   return  $root$ 
```

^a $L = \{(x_{i_1}, \dots, x_{i_n}, y) | i \in [1, m]\}$, the training data.

^bFind the feature j and its value x_{i_j} that split L into two purest subsets, i.e. subsets with as large proportion of ALLOW or DENY as possible.

^cSplit L into $L_l = \{(x_{k_1}, \dots, x_{k_n}, y) | k \in [i, m] \wedge x_{k_j} = x_{i_j}\}$ and $L_r = L - L_l$.

^dCalculate $metric(L_l) + metric(L_r) - metric(L)$, where metric is a function measures the label purity of a set, e.g. entropy or Gini Impurity.

3.7.3 Namespace Inference

Decision trees inherently have the capability of representing hierarchical namespaces in access-control rules. Unfortunately, traditional decision tree learning algorithms (e.g., Algorithm 1) do not recognize hierarchical features well and thus cannot generate the inherent hierarchical structure. For instance, given a file path as a feature, such as `"/projects/proj1.html"`, it is treated as a single string; therefore, a node may be generated with a decision condition `"path==/proj/1.html"` but not with `"prefix1==/proj"`. To extend that, P-DIFF generates rules not only for the path, but also for its parent directory `"/proj"`.

P-DIFF makes two efforts to generate hierarchical rules. First, P-DIFF adopts Quinlan-encoding [3] to expand the hierarchical features. P-DIFF takes the annotations of hierarchical features (Table 3.2) with a delimiter and expands a string with all its prefixes. In the case of file path, once the feature is annotated as hierarchical and delimited with `"/"`, then prefix features will be generated, such as `"prefix0==/"` and `"prefix1==/projects"`. Note that the annotation is a one-time effort.

Second, P-DIFF adopts a hierarchy aware mechanism [137] for the best-split step (Algo-

rithm 1, Line 3) during decision tree learning. P-DIFF follows the hierarchical order to choose a feature that best-splits the input data in the tree learning phase. Let us assume that there exist three attributes ["user", "method", "file path"]. P-DIFF will expand the three attributes to five features in the feature vector: ["user", "method", "prefix0", "prefix1", "file name"].

P-DIFF first tries to choose a best-splitting feature from ["user", "method", "prefix0"] and if no feature results in change reduction, it tries ["prefix1"] and ["file name"] in order. Once P-DIFF finds a feature with metric gain, it ensures that features at a higher level of the hierarchy are considered before features at a lower level.

3.8 Policy Change Management

3.8.1 Algorithm

Algorithm 1 and the other traditional decision tree (DT) learning algorithms cannot deal with policy changes for two reasons. First, traditional DTs cannot encode changes over time. P-DIFF addresses this by using TCDT (§3.6).

Moreover, traditional algorithms (e.g., CART, ID3 and C4.5 [12, 83, 84]) cannot directly work with TCDT. This is because the splitting metrics (Gini Impurity and entropy) employed by traditional algorithms do not consider rule changes over time—both Gini Impurity and entropy are calculated based on the aggregated results and fail to take the time information into account. Figure 3.7 shows two cases that the splitting metrics in traditional DT learning cannot decide whether to split or not in Algorithm 1’s split step, because all the “purity metrics” are same before and after the split (Row 3). On the other hand, the correct split decision can be made if the “time series” information is taken into account (Row 4).

Therefore, we design a TCDT learning algorithm. Learning a TCDT requires different *training input* and *splitting metric* from Algorithm 1. For TCDT, the *training input* is a time-series

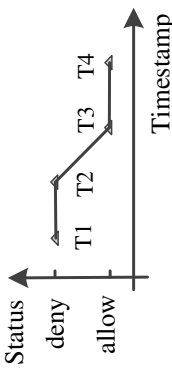
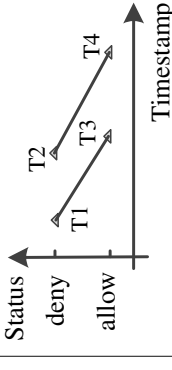
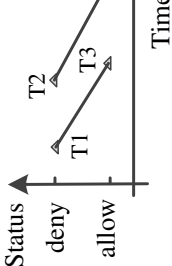
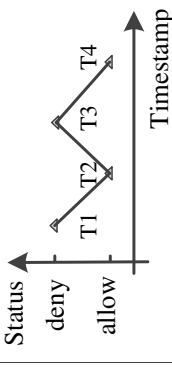
| | Case 1: A single rule change (should not split) | Case 2: Multiple rule changes (should split) |
|--------------------------|--|--|
| Policy Change | GET, /proj/* DENY \rightarrow ALLOW | GET, /proj/1.htm DENY \rightarrow ALLOW GET, /proj/2.htm DENY \rightarrow ALLOW |
| Access log subset | T1, GET, /proj/1.htm DENY T2, GET, /proj/2.htm DENY T3, GET, /proj/1.htm ALLOW T4, GET, /proj/2.htm ALLOW | T1, GET, /proj/1.htm DENY T2, GET, /proj/1.htm ALLOW T3, GET, /proj/2.htm DENY T4, GET, /proj/2.htm ALLOW |
| Purity metrics | If not split $P_{allow}: 0.5, P_{deny}: 0.5$ Gini Impurity: 0.5 Entropy: 1 | If not split $P_{allow}: 0.5, P_{deny}: 0.5$ Gini Impurity: 0.5 Entropy: 1 |
| Time Series | If split $P_{allow_left}: 0.5, P_{allow_right}: 0.5$ Gini Impurity: 0.5 Entropy: 1 | If split $P_{allow_left}: 0.5, P_{allow_right}: 0.5$ Gini Impurity: 0.5 Entropy: 1 |
| | ChangeCount: 1  | ChangeCount: 2  |
| | ChangeCount: 2  | ChangeCount: 3  |

Figure 3.7. Examples that demonstrate splitting events in TCDDT-based policy learning (cf. §3.8). Case 1 does not require splitting, while Case 2 does due to the condition: `if prefix2=="proj/1.htm"`. Traditional splitting metrics cannot decide whether to split if a change is involved, because the possibility of ALLOW or DENY is always 0.5 in each subset (Gini Impurity: $1 - (p_{allow})^2 - (p_{deny})^2 = 0.5$, Entropy: $-p_{allow}\log(p_{allow}) - p_{deny}\log(p_{deny}) = 1$). The time-series change counts differ in the subsets and can guide correct splitting events.

sequence:

$$\mathcal{L} = ((\tau_1, x_{1_1} \dots x_{1_n}, y_1), \dots, (\tau_m, x_{m_1} \dots x_{m_n}, y_m)) \quad (3.5)$$

where $\tau_i < \tau_j$ for $i < j$, x_{i_1}, \dots, x_{i_n} is the feature vector generated from timestamped access attributes denoted as (*timestamp, subject, object, action*), and $y_i \in \{0, 1\}$ is the prediction label from the access result r . For *splitting metric*, we propose a new metric, *change-count*, to effectively differentiate multiple unchanged rules from one changed rule only based on logs, as shown in Figure 3.7.

Intuitively, the change-count of a time-series sequence is how many times the end result is changed. Mathematically, for the time series \mathcal{L} , the change-count is defined as:

$$CC(\mathcal{L}) = \sum_{i=1}^{n-1} |y_{i+1} - y_i| \quad (3.6)$$

When splitting \mathcal{L} into two sequences \mathcal{L}_l and \mathcal{L}_r , the algorithm tries to find the feature j and its value x_{i_j} in the way:

$$i, j = \underset{i \in [1, m], j \in [1, n]}{\operatorname{argmax}} (CC(\mathcal{L}) - \sum_{\mathcal{L}_k \in \operatorname{split}(\mathcal{L}, j, x_{i_j})} CC(\mathcal{L}_k)) \quad (3.7)$$

where $\operatorname{split}(\mathcal{L}, j, x_{i_j})$ is a function that splits a sequence \mathcal{L} by examining whether $x_{i'_j} = x_{i_j}$, where $i' \in [1, m]$.

We set the splitting goal to be generating the least changes possible—generating a new rule should reduce the total change-count as many as possible. The goal can effectively decide when to split in both cases of a single rule change and multiple rule changes, as shown in Figure 3.7. Also, in the case that there is no rule change, the goal can also make the correct splitting so that different rules are generated, as shown in Figure 3.8.

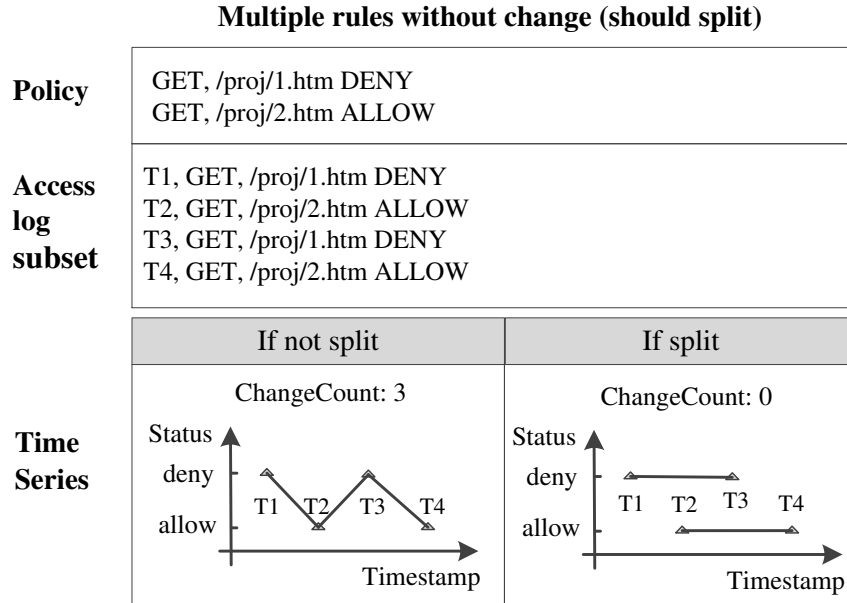


Figure 3.8. An example that demonstrates TCDT learning algorithm can infer rules even there is no rule change. In this case, a splitting is required on the condition: `if prefix2=="/proj/1.htm"`. The time-series change count favors this splitting as the change count decreases from 3 to 0 after splitting.

3.8.2 Optimizations

Calculating change-counts, $CC(\mathcal{L})$, defined in Equation (6) has a significant impact on the training time for model generation, as it needs to be calculated *many times* for every feature j and value x_{i_j} in Equation (7). Note that the change-counts are different for different features and values, and the results cannot be directly reused. Therefore, without an efficient implementation of change-counts, P-DIFF cannot build the model in a short amount of time for large volumes of access logs (e.g., the Wikipedia dataset evaluated in §3.10 has more than 300 million log entries).

Our initial change-count implementation is to loop through the entire *access result array* that stores the access result (with 0 to represent ALLOW and 1 to represent DENY), as shown in Figure 3.9. However, we find that this straightforward implementation is inefficient. It takes 51 minutes to train a model from 20 million log entries, and more than 2 hours for 40 million log entries. Hence, it cannot work with datasets at the similar scales of our Wikipedia dataset (369 million entries).

To accelerate the training time, we design and implement the following two optimizations as illustrated in Figure 3.9:

1. We observed that in typical cases, ALLOW is much more frequent than DENY. Therefore, looping through the entire access result array is unnecessary. To improve it, our first optimization only loops through all the DENYs and checks for possible changes next to each DENY. Note that given that the change-count needs to be calculated many times, we generate an index of all the 1 values after the first change-count calculation and use the index in the subsequent ones.
2. We further adopt discrete convolution [85] to calculate the sum of every two adjacent result numbers: if the sum is 1, there is a change. We use the implementation of discrete convolution as efficient vectorized operations in the Numpy library [75].

With these two optimizations, our implementation of P-DIFF is able to handle the entire Wikipedia dataset. Our evaluation in §3.10.7 shows P-DIFF only takes 19 minutes to train a model from 320 million log entries.

3.9 Use Cases

P-DIFF supports two use cases, change validation and forensics diagnosis, on top of its TCDT-based access control policy learning described in §3.6–§3.8. In this section, we show how to use the learned TCDT as a policy evolution representation for supporting the two use cases.

3.9.1 Change Validation

P-DIFF continuously monitors the new-coming access results from the access logs. For each access, P-DIFF calculates the expected access results (ALLOW or DENY) based on the policy maintained in the TCDT. When P-DIFF observes that the access results deviate from the policy it currently maintains, P-DIFF treats the deviation as the result of a policy change. P-DIFF

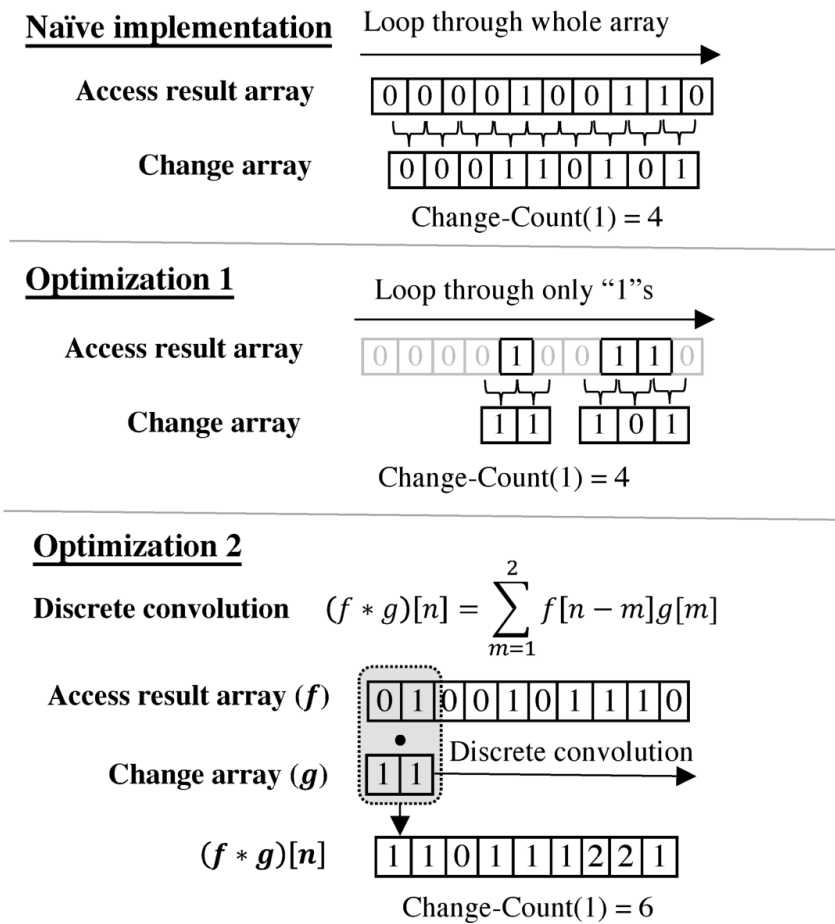


Figure 3.9. Two optimizations to efficiently calculate change-count. The naïve implementation is to loop through the whole access result array and count the changes. Optimization 1 improves it by only looping through the seldom occurred DENYS (value 1). Optimization 2 uses discrete convolution [85] $(f * g)[n]$ to calculate the sum of every two adjacent numbers (if the sum is 1, there is a change).

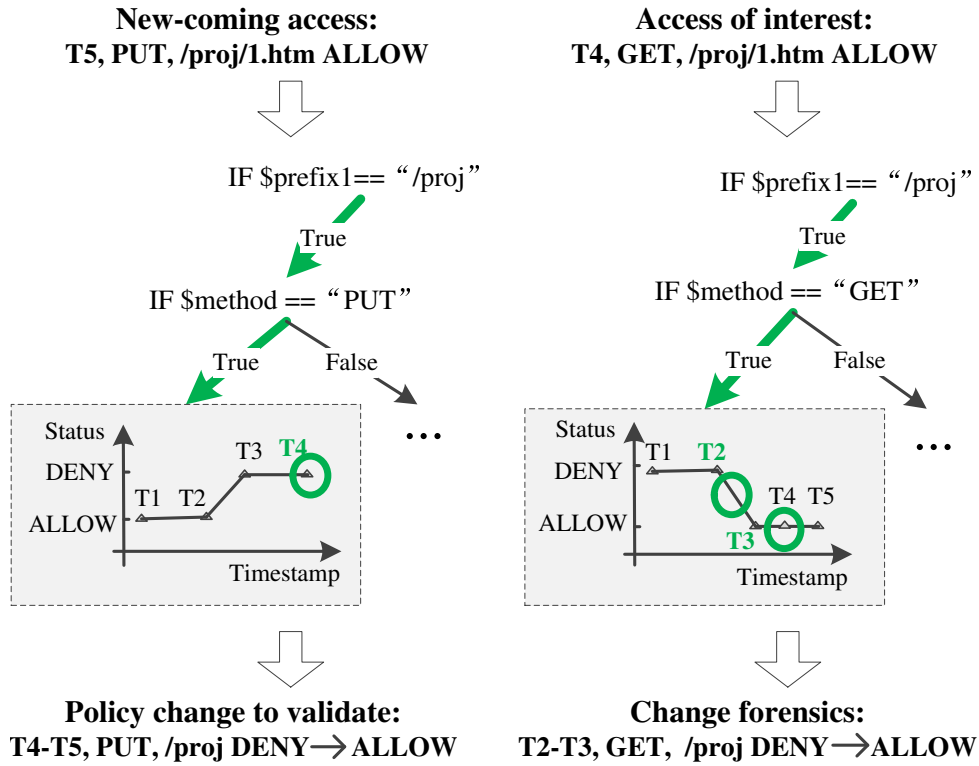


Figure 3.10. Two use cases with P-DIFF: (a) change validation and (b) forensic analysis. For (a), P-DIFF detects a policy change based on the deviation of the access results at T5 and the policy maintained in TCDT. P-DIFF then notifies the sysadmins with the policy change for validation. For (b), P-DIFF backtracks the time series at the leaf node to pinpoint the root-cause change between T2 and T3 that permitted the access.

then notifies sysadmins with the changed access control rules and asks sysadmins to validate the changes. If the sysadmins confirm the change to be expected, P-DIFF updates the TCDT to incorporate the policy changes. Otherwise, P-DIFF detects access control misconfigurations. It discards the access results and keeps monitoring new access results (after the misconfigurations get fixed by the sysadmins).

P-DIFF presents the change policy by extracting it from the corresponding path in the TCDT. Figure 3.10 (left) shows an example of change validation. When monitoring a new access at timestamp T5, P-DIFF calculates its access result based on the TCDT (which is DENY); however, P-DIFF finds that the access was actually ALLOWed in the access log. The deviation leads to the validation request.

3.9.2 Forensic Analysis

Given an access of interest (e.g., an illegal access that steals confidential information), P-DIFF can pinpoint the policy change that permitted the access by searching the policy evolution history maintained in the TCDT. This is achieved by finding the path in the TCDT that determines the result of the access and searching for the change that started permitting the target access in time series encoded at the leaf node. Figure 3.10 (right) shows an example of forensic analysis. Given a target access at T4, P-DIFF finds the corresponding leaf node in TCDT and backtracks through the time series to find out the root-cause policy change happened between T2 and T3.

Note that forensic analysis can be done in-situ or serve as an independent tool postmortem to any security incidents in which P-DIFF reads historical access logs and builds the TCDT by analyzing the history.

3.10 Evaluation

We evaluate P-DIFF using controlled experiments based on datasets collected from five real-world deployments of various systems with different scales, including firewalls of a software company, an online Wikipedia service, and three websites hosted in academic organizations. Table 3.3 describes these datasets.

3.10.1 Systems and Datasets

- **Wikipedia.** An online encyclopedia website for public access and has 33 million registered users. We collect access logs from a public dataset of request traces to Wikipedia in September 2007 [112] (which is the only online dataset available). Access control is implemented by the MediaWiki application. The protected resources including protected pages of different protection levels, such as full-protected and semi-protected pages, which can only be modified by sysadmins and registered users respectively.

- **Center A** multi-node web server hosting home page, online tools and personal pages for a research center with more than 10 faculty members and 50 graduate students in a research university. Resource protection is done with Apache HTTPD web server's access control configurations (Figure 3.1). The protected resources include a public website for news and personal pages, and an internal website for group-internal resources. The protection policies of the whole server are maintained by a sysadmin, but each member can modify the protection policies of their own pages.
- **Course A** department-wide course web server hosting 200+ courses in the past 5-6 years for more than 2000 undergraduate and graduate students. Resources are mainly protected by file permissions based on operating system level access control mechanisms. The protected resources include public and private web pages of course materials. The protection policies of different course pages are maintained by the corresponding instructors and teaching assistants. Course materials can be changed from private to public during the semester and changed back to private after the semester.
- **Company Firewall logs** used by a software company serving millions of users. The policies specify blocking IPs and IP ranges to protect the company network against Internet-based attacks.
- **Group A** website hosting group pages and personal pages for a research group with more than 20 researchers. Resource protection (mostly for private web pages) is done through file permissions. The access policies are maintained by one sysadmin.

3.10.2 Experimental Design

To evaluate the effectiveness of P-DIFF requires two pieces of information: (1) the access logs recording access requests and access results, and (2) the access control policy changes that resulted in the access results (the ground truth).

Table 3.3. Datasets used in our evaluation. The datasets cover a variety of systems, protection mechanisms, access control configurations at different scales (§3.10.1).

| Dataset | Configuration | Time Span | # Access |
|-----------|--------------------------|-----------|----------|
| Wikipedia | Application logic | 2 weeks | 369M |
| Center | Web server configuration | 11 months | 5.9M |
| Course | File permission | 11 months | 3.8M |
| Company | Firewall | 3 hours | 100K |
| Group | File permission | 1 month | 32K |

The Wikipedia dataset [112] includes both of the two pieces of information. The policy changes can be obtained based on Wikipedia’s page protection change history [67], which records the protection changes of each page (e.g., changed from publicly editable to only accessible by specific users).

For Center, Course, Company and Group, we do not have the policy change history (configuration changes in these systems were not tracked). Therefore, we only use the requests recorded in the access logs and generate new access results by synthesizing the access control policies (the original access results are ignored). As shown in Table 3.4, we generate different types of policy changes to cover different scenarios; for each type, different attributes (subject, object, and actions) are selected to be changed. We also selectively change policies that affect objects with different access frequencies, categorized as “rare”, “normal”, and “frequent” based on the frequency they are accessed. The experiment design allows us to study how access frequency affects P-DIFF’s policy learning. We randomly choose a time to make a policy change for a given dataset. In total, each dataset contains 60 policy changes across its time span, with 15 changes in each type, as detailed in Table 3.4. With the policies and their changes, all the access results before and after the change are set accordingly.

All the experiments are conducted on an AWS instance, with Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz CPU (16 cores), 64GB memory, and Ubuntu 16.04.

Table 3.4. Different types of access control policy changes used in the experiments for the Center, Course, Company, and Group datasets. The detailed experiment design can be referred to in §3.10.2.

| Category | Change Types (# Changes) | Dataset Applied |
|-----------------|-------------------------------------|--------------------|
| File permission | Type 1: allow file access (15) | Course Group |
| | Type 2: block file access (15) | |
| | Type 3: allow directory access (15) | |
| | Type 4: block directory access (15) | |
| Web server ACL | Type 5: allow user access (15) | Center |
| | Type 6: block user access (15) | |
| | Type 7: allow GET/PUT method (15) | |
| | Type 8: block GET/PUT method (15) | |
| Iptable ACL | Type 9: allow ip access (15) | Company |
| | Type 10: block ip access (15) | |
| | Type 11: allow subnet (15) | |
| | Type 12: block subnet (15) | |

3.10.3 Overall Results

Change Validation.

We evaluate P-DIFF’s effectiveness in detecting policy changes. For each dataset, we divide the time span (cf. Table 3.3) of the access logs into two parts: the first part is used for training (observed accesses) and the second part is used for testing (upcoming accesses). The first part takes $\frac{7}{10}$ of the time span, and the second part takes the rest $\frac{3}{10}$. If a real policy change is not detected, we count it as a false negative. If a detected policy change is incorrect, we count it as a false positive.

Table 3.5 shows the number and percentage of policy changes P-DIFF detects from each dataset. In total, P-DIFF detects 93 (94%) out of 99 rule changes with 12 false positives and 6 false negatives. For each dataset, P-DIFF generates less than 6 false positives and less than 3 false negatives. For Wikipedia and Group datasets, P-DIFF generates 0 false positives and negatives. This shows that P-DIFF works effectively on small education systems (Group, Course, Center), medium commercial systems (Company with millions of users) as well as large-scale popular

Table 3.5. Policy changes detected by P-DIFF. FP stands for false positive and FN stands for false negative.

| Dataset | # Total changes | # Detected changes | Precision (FP) | Recall (FN) |
|-----------|-----------------|--------------------|----------------|-------------|
| Wikipedia | 25 | 25 (100%) | 1.0 (0) | 1.0 (0) |
| Center | 18 | 16 (89%) | 0.76 (5) | 0.89 (2) |
| Course | 18 | 17 (94%) | 0.85 (3) | 0.94 (1) |
| Company | 21 | 18 (86%) | 0.81 (4) | 0.86 (3) |
| Group | 17 | 17 (100%) | 1.0 (0) | 1.0 (0) |
| Total | 99 | 93 (94%) | 0.89 (12) | 0.94 (6) |

websites (Wikipedia ranked the 7th popular website in the world [2]).

Forensic Analysis.

To evaluate the effectiveness of forensic analysis, we select an *access of interest* after each policy change. The access of interest is an access that was supposed to be denied based on the policy before the change, but is allowed after the policy change. In other words, if the policy change is misconfigured, the access could be illegal. We feed the access of interest into P-DIFF and evaluate whether P-DIFF can pinpoint the root-cause policy change that permits the access.

As shown in Table 3.6, P-DIFF pinpoints the root-cause policy change for 283 (93%) out of 303 accesses of interest. For the Wikipedia and Center datasets, TCDT correctly encodes 114 out of 123 changed rules on the normal objects (i.e. user and method). For the Course, Company and Group datasets, TCDT correctly encodes 169 out 180 changed rules on the hierarchical objects (i.e. directory and subnet). Once the rule is correctly encoded, P-DIFF can always correctly backtrack the time series.

We investigate the 18 accesses of interest for which P-DIFF fails to pinpoint the root-cause policy changes. In 50% of the cases (9 out of 18), the objects being accessed are rarely accessed in the history (refer to §3.10.2 for the experiment design). In these cases, P-DIFF fails to generate any rules and thus cannot pinpoint the rules. In the remaining 9 cases, P-DIFF generated inaccurate rules (i.e. on the parent directory instead of on the child directory), and therefore fails

Table 3.6. Effectiveness of forensic analysis. P-DIFF can pinpoint the root-cause policy changes that permit the access of interest in the evaluation.

| Dataset | # Access of interest | Pinpointing root-cause changes |
|----------------|-----------------------------|---------------------------------------|
| Wikipedia | 63 | 61 (97%) |
| Center | 60 | 53 (88%) |
| Course | 60 | 59 (98%) |
| Company | 60 | 51 (85%) |
| Group | 60 | 59 (98%) |
| Total | 303 | 283 (93%) |

to pinpoint the precise root cause change.

Effectiveness Discussion.

For policy change detection, the goal of P-DIFF is to detect as many true changes as possible while minimizing the reports of false changes. Our evaluation result in Table 3.5 shows P-DIFF detects 93 out of 99 changes, which means only six (7%) changes are missed. P-DIFF generates 12 positives which increase sysadmins’ validation overhead. Overall, the validation overhead is reasonably small. Taking Wikipedia, one of the most popular online services, as an example, the sysadmins of Wikipedia only need to validate 25 changes in 4.2 days (which is the testing period).

For forensic analysis, as shown in Table 3.6, P-DIFF successfully pinpoints root-cause changes of 283 out 303 accesses of interest, This means that sysadmins can use P-DIFF to efficiently diagnose 93% of the target access event. Without P-DIFF, the sysadmins may have to go through either a large number of access logs or various configuration and code in different components as discussed in §3.2.

3.10.4 Precision, Recall, and F-Score

The detection of policy changes is based on detecting the deviation of access results. We look into how well Time-Change Detection Tree (TCDDT) can serve as a classifier to decide the

access results by whether the decision matches the actual access results. Higher accuracy means more accurate policy learning. We compare TCDT with an implementation of the traditional decision tree implemented using Apache Spark MLlib [102] denoted as Spark-DT.

The experiment is conducted on the Center and Course datasets as they have longer duration of logs and enable a comparison between different testing sets. Every three months of logs are used as training set and the last month of logs are used as testing set.

Figure 3.11 shows that P-DIFF's TCDT achieves a precision of 0.997, a recall of 0.92, and a F-score of 0.94, while Spark-DT achieves a precision of 0.83, a recall of 0.86, a F-score of 0.80. P-DIFF-TCDT improves precision by 19.5%, recall by 6.5%, and F-score by 17.3%. P-DIFF's TCDT improves the prediction precision of Spark-DT for all testing sets, and the improvement is more prominent when the training set contains more rule changes, e.g. the training set for the February testing set in the Center dataset. P-DIFF-TCDT also improves the prediction recall for the other testing sets.

P-DIFF-TCDT does not increase the prediction accuracy on accesses happened in Nov in Figure 3.11 (a). We consulted the sysadmin and learned that this month the website had a major change: it is ported to another server and many new pages are added. P-DIFF has no knowledge of these new pages and so reports accesses to them as UNKNOWN (cf. §3.6). Although this hurts the accuracy in our evaluation, in real usage P-DIFF will retrain a new TCDT with the new accesses and gets good accuracy. As shown from the result of Dec, Jan, and Feb in Figure 3.11 (a), after training TCDT with the new accesses, P-DIFF gets precision and recall both above 0.9.

3.10.5 False Positive and False Negative

P-DIFF generates false positives and negatives when the training set does not have enough information. For false positives, P-DIFF may generate wrong rules. In one case, P-DIFF generates a rule that access to ‘‘/proj1’’ should be denied based on the observation that accesses to ‘‘/proj1/1.htm’’ and ‘‘/proj1/2.htm’’ are all denied in the training phase. In the

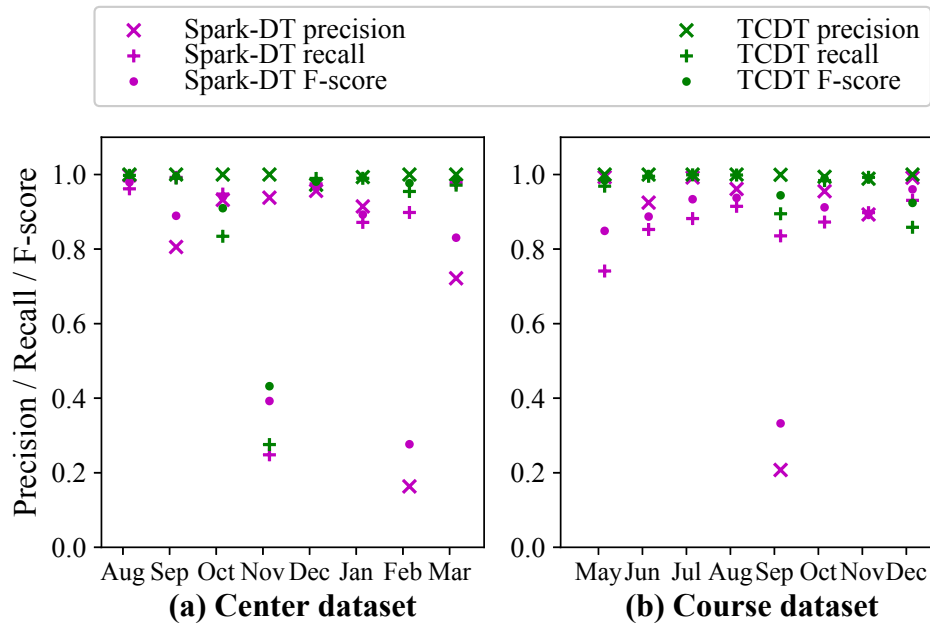


Figure 3.11. Precision, recall, and F-score of TCDT classifying access results for the Center and Course datasets. The x-axis shows the time of the testing data, which is a month of logs in the dataset. The training data is the three continuous months of logs before the testing month.

detecting phase, P-DIFF observes accesses to ‘ ‘/proj1/3.htm’ ’ are allowed and then decides a rule change on ‘ ‘/proj1’ ’. But in fact, the access rules are in the file level instead of the directory level and accesses to ‘ ‘/proj1/3.htm’ ’ are always allowed. P-DIFF generates a wrong rule because there is no access to ‘ ‘/proj1/3.htm’ ’ in the training set.

False negatives are mainly because of “rare” objects (§ 3.10.2). P-DIFF fails to infer the rules and thus cannot detect the change. For example, in the training phase, P-DIFF observes all accesses from an IP ‘ ‘192.168.1.1’ ’ are allowed and so infers an allow rule for this IP. In the detecting phase, P-DIFF observes accesses from ‘ ‘192.168.1.2’ ’ are denied. Since P-DIFF has no rule for ‘ ‘192.168.1.2’ ’, it cannot detect the change and the change is actually access to subnet ‘ ‘192.168.1.*’ ’ has been modified from ALLOW to DENY.

Table 3.7. Performance of P-DIFF: training time, and time for validation and forensics per access.

| Dataset | # Log entries for training | Training time | Validation (per access) | Forensics (per access) |
|-----------|-------------------------------|------------------|----------------------------|---------------------------|
| Wikipedia | 11.4M | 188s | 240 μ s | 143ms |
| Wikipedia | 258 M | 746 s | 963 μ s | 12.8 ms |
| Center | 4.13 M | 43.7 s | 10.8 μ s | 2.7 ms |
| Course | 1.73 M | 20.3 s | 13.0 μ s | 4.8 ms |
| Company | 70.0 K | 2.46 s | 26.1 μ s | 2.8 ms |
| Group | 17.6 K | 9.93 s | 38.2 μ s | 3.7 ms |

3.10.6 Execution Time

Table 3.7 shows the execution time of P-DIFF for training, validation, and forensics, respectively in previous validation and forensic experiments (cf. §3.10.3). P-DIFF’s training is very efficient. For the largest dataset (Wikipedia) with 258 million log entries, the training that learns the TCDT only takes 12 minutes. For smaller datasets, P-DIFF takes less than 1 minute for training. Note that the training is an offline process without the need of being real-time.

The validation and forensics can be done in microseconds and milliseconds per access. The efficiency is decided by the depth of the TCDT—the deeper the TCDT is, the more time it takes. Therefore, the time taken for validation and forensics in Wikipedia dataset is larger than the others. Forensics takes a longer time for backtracking the time series. The performance for forensics is satisfying, given that forensics is typically postmortem to the security incidents and is done offline. For validation, P-DIFF needs to validate every access recorded in the access log. Note that this does not need to be done sequentially but can be done in parallel as each access is independent. Therefore, we conclude that the execution time of P-DIFF is acceptable in real-world settings.

3.10.7 Scalability

To understand how P-DIFF scales with real-world access log data, we evaluate P-DIFF with different numbers of log entries using the Wikipedia dataset. We use the continuous log

entries of 10 million, 20 million, and all the way up to 320 million logs as different training sets. Note that 320 million is the number of logs from Wikipedia for 12 days out of the total 14 days of logs, which is the largest real-world dataset we can collect by far (the remaining 2 days of logs are used as the testing set). The results in §3.10.3 show that less than 320 million of logs have already been effective for P-DIFF to do an accurate change detection (100% precision and recall) and forensic analysis (97% success). Therefore, in practice, we can only maintain the most recent 12 days of logs for P-DIFF to be effective for Wikipedia.

Figure 3.12 shows the training, validation and forensics time respectively. When the number of log entries increases from 10 million to 320 million, the training time increases linearly from 2 seconds to 19 minutes, as shown in Figure 3.12 (a). The linear increase of the training time is due to the fact that P-DIFF needs to go through every log entry to infer policies and policy changes. Considering training is an offline process, it is acceptable to take 19 minutes to train a TCDT once a while. Training is only necessary for the first time usage of P-DIFF or when P-DIFF encounters too many UNKNOWN accesses (cf. §3.11).

The validation and forensics time for an access only takes a few milliseconds, as shown in Figure 3.12 (b). Both validation and forensics need to search the decision tree to find the leaf node applies for a given access. Therefore, the depth of the leaf node decides the execution time of the validation and forensics. The average validation and forensics time are decided by the depth of the leaf node that encodes a dominant policy, which applies to most accesses. In Wikipedia, there is a dominant policy that “all page read should be allowed”. This dominant policy node can be located in different depth based on the training set. This explains the variation of the execution time for validation and forensic analysis in Figure 3.12(b). Overall, the variance is small enough for efficient validation and forensics.

3.10.8 Validation Overhead

There are two types of validation that sysadmins need to perform. The first one is when P-DIFF detects a change, it would inform sysadmins to validate whether the change is intended

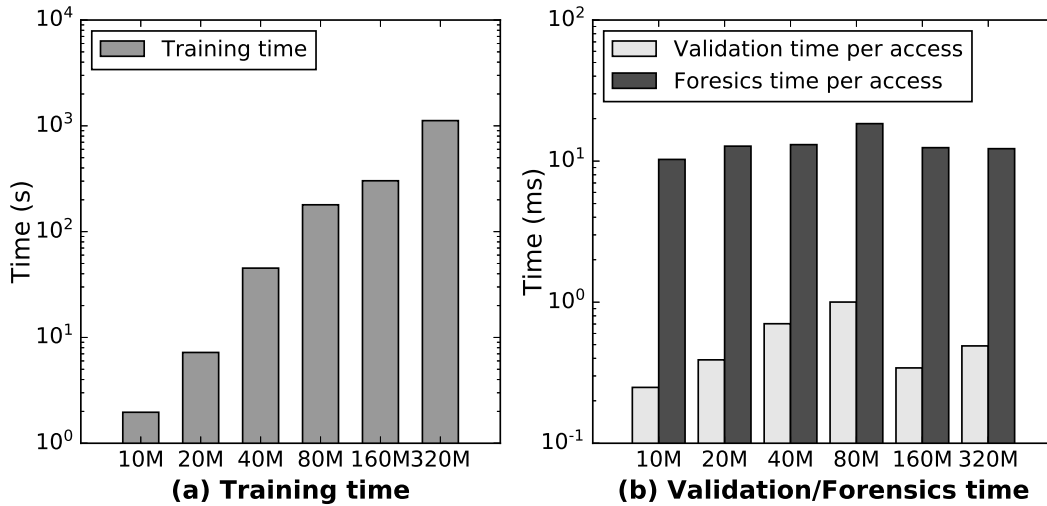


Figure 3.12. Scalability analysis in terms of execution time for training, validation, and forensics with the increasing numbers of log entries from the Wikipedia dataset. Training time is linear to the number of log entries; it takes 19 minutes to train a model from 320 million log entries. Validation and forensics time are always low (1–10 milliseconds).

or not. Our evaluation results show the overhead for this type of validation is acceptable. Take Wikipedia dataset as an example, only 25 validation needs to be done for the tested 111 million accesses during the test period.

The second type of validation is when P-DIFF reports an UNKNOWN access after a new object (e.g., a file) is added to the system. In the five datasets used in the evaluation, only 12 new objects were unobserved during the training period. Therefore, only 12 validations need to be done for those UNKNOWN cases (shown as false positives in Table 3.5). Intuitively, popular objects should be observed during the training period, while rare objects, even not observed during the training period, would not incur too much overhead for validation because of its rareness.

Moreover, the overhead for importing new objects (e.g., files) should not be excessive, because sysadmins typically do not need to validate every single new object but can validate the higher level hierarchy. For example, typically all the files in the same directory have the same permission settings and so the directory can be validated in total, avoiding validating individual new files. Certainly, in a case that every file under the same directory has distinct permission

settings, the sysadmin needs to validate them one by one (but this also reflects a pathological practice in the security management).

3.11 Discussions and Limitations

P-DIFF learns access control policies from access logs, and thus is limited to the information recorded in access logs. As discussed in §3.6, if a new access with an unseen attribute or value in logs, P-DIFF explicitly classifies it as UNKNOWN and notify sysadmins to validate it. There are two cases that could cause UNKNOWN. First, some new attributes or values are added to the access control policies, such as the creation of new users, roles, and files, etc. In this case, the UNKNOWN is a real change and so is desirable to be validated. Second, if the resource is extremely cold (there are very few accesses), P-DIFF may not be able to learn the related rules due to the lack of information. We observe in our datasets that public resources are more frequently accessed than private resources. In a few extreme cases, the private resources are only accessed once a month. In this case, the validation request on UNKNOWN is also acceptable because a rarely accessed resource is desired to be manually examined and it will not cause too much burden for sysadmins for its rareness. In addition, in both cases, P-DIFF will retrain a new TCDT so that the unknown attributes and values can be learned for future classification. As shown in §3.10.6, the training time of a TCDT in real-world datasets is in the range of 2 seconds to 19 minutes. Therefore, it is acceptable to retrain a new TCDT in a normal frequency like once an hour or once a day.

P-DIFF cannot work with access logs that miss important information (i.e., subject, object, action, and result). As shown in Table 3.1, although most of the studied software systems record the required information in their access logs, there do exist some systems that missing certain key information, such as subject and access results. P-DIFF cannot infer rules from access logs of those systems before the logs are enhanced. As discussed in [17, 127], incomplete access logging is a significant flaw that impairs auditing and forensics and thus should be fixed. Enhancing

logging is beyond the scope of P-DIFF. Future work in automatically enhancing access logs would be a valuable direction to be explored.

3.12 Acknowledgement

This chapter, in part, is a reprint of the material as it appears in Proceedings of the 26th ACM Conference on Computer and Communications Security, 2019. Xiang, Chengcheng; Wu, Yudong; Shen, Bingyu; Shen, Mingyao; Huang, Haochen; Xu, Tianyin; Zhou, Yuanyuan; Moore, Cindy; Jin, Xinxin; Sheng, Tianwei. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Related Work

4.1 Access-control Misconfiguration

4.1.1 Access-control Configuration Testing and Verification

Testing and verification techniques for access-control policies were previously explored mainly by the software engineering community, as a promising approach to detect access-control misconfigurations [63, 64, 80, 46, 35, 48, 54, 71, 53]. The basic idea is to write or generate test cases that assert on the access results of test requests (i.e., test inputs). With comprehensive test inputs and oracles, misconfigurations will fail the tests.

Prior techniques on access-control testing and verification are based on formal modeling (e.g., using ACML). However, it is non-trivial to encode the actual system configurations using the modeling language, due to the diversity and complexity of access-control implementations [125]. Furthermore, it takes significant effort to maintain the consistency between the model and system configuration changes, which has been reported as one of the main reason that impair access-control in real world [99]. Another major obstacle is that the prior approaches require sysadmins to provide oracles or specifications (to check the test results). With the velocity of access-control changes [99, 23], it is untenable to maintain consistency or oracles/specifications.

ACTESTs address the fundamental limitations of prior access-control testing and verification approaches—it requires neither formal policy models nor oracles/specifications: 1) ACTESTs take advantage of the original program to test its own access-control configurations to achieve

high fidelity; 2) ACTESTs focus on the impact of access-control changes and flags high-impact changes, without the need for sysadmins to write test oracles or specifications.

4.1.2 Access-control Misconfiguration Detection

Prior work proposes to detect access-control misconfigurations based on the heuristic that users tend to have similar permissions to similar resources [23, 11, 97]. So, those techniques classify users and resources into groups and identify misconfigurations when the same group of users has different permissions to the same group of resources. Such techniques can only detect limited types of misconfigurations where the heuristic applies. As noted by Das et al. [23], *“We do not claim that techniques will find all misconfigurations, as the notion of policy itself is not defined in most of our deployment settings. Also, given that access permissions change very organically over time and several of these changes are linked to adhoc and one-off access requirements, it is very difficult for an automated system to deduce the exact and complete list of all misconfigurations.”* Our work presents a complementary approach to focus on detecting changes of dynamic system behaviors instead of the similarity of static permissions.

4.1.3 Characteristic Study of Access-deny Issues

Xu et al. [127] studied the access-deny issues from a Human-Computer Interaction (HCI) perspective by looking at the practices of how sysadmins resolving access-deny issues from online forums. Even though their study shed light on the important problem that many current server programs do not provide adequate feedback information, but they did not study further into the details. In particular, *no solution was proposed and evaluated in their study*. Our work not only provides additional insights to their findings (e.g., Finding 3 on commit history) but also provides a solution to enhance access-deny log messages and insert new log statements.

4.2 Other Types of Misconfiguration

4.2.1 Configuration Testing

A few recent works also propose to use testing techniques to detect misconfigurations, e.g., Ctest [105] and PCheck [126]. We share the same view that testing is a practical and effective approach for misconfiguration detection; however, ACTESTs are fundamentally different. First, no prior work can be applied to access-control configurations, because they rely on failure symptoms to determine the correctness of configuration values (e.g., crashing behavior, exceptions, and performance degradation). However, access-control misconfigurations do not lead to clear failure symptoms. For the same reason, most of the existing misconfiguration detection techniques for functional correctness and performance cannot address access-control misconfigurations [136, 107, 47, 79, 90, 8, 68, 16, 124]. Moreover, existing configuration testing techniques are limited to unit-level tests [126, 105]. Differently, ACTESTs focus on system-level tests in order to reason about end-to-end access-control. This requires addressing the system challenges to run expensive tests efficiently.

4.2.2 Misconfiguration Detection

Previous work proposed many novel techniques [136, 107, 47, 79, 90, 8, 68, 16, 124] for detecting misconfigurations that cause reliability issues. These techniques mainly infer correctness rules from source code, other systems' configurations and documents, and then use the rules to check target configurations. Though effective on detecting reliability misconfigurations, these techniques may not suit for detecting access-control misconfigurations. Unlike other types of configurations, access-control configurations are typically flexibly customizable and so it is hard to infer general rules on what configurations are correct from either source code, other systems' or documents. For example, source code may encode the valid range of a thread number parameter as an if condition, but source code itself will not decide which user permission is valid or not as it needs to provide as much flexibility as possible for sysadmins to make the decision.

Therefore, instead of checking configurations against static rules, ACTEST run access-control configurations with a modified program and observe how the running program behave differently.

4.3 Other Related Techniques

4.3.1 Access-control Bug Detection

Besides misconfigurations, vulnerabilities can also be introduced by software bugs, e.g. the software could miss permission checks. Sun *et al.* [104] use static analysis to infer the protected domains from the source code of a web applications, and then detect any unchecked accesses to these pages. RESIN [133] is a runtime system that enforces data-flow assertions to prevent exploits of web application security vulnerabilities. Nemesis [22] is a runtime system for preventing authentication and access control bypass attacks. SPACE [72] is a tool to find access control bugs in web application based on a catalog of patterns. Our work has a complementary focus on access control misconfigurations—even if we have a correct coded software, misconfigurations can still introduce security holes.

4.3.2 Intrusion Detection

One related research area on security-related log analysis is intrusion detection [26]. An intrusion detection system (IDS) monitors system or network events, detects malicious activities, and reports them to sysadmins. Previous works on IDS can be classified into *signature*-based and *anomaly*-based methods. *Signature*-based IDS detects known attacks by recognizing their patterns, such as a specific sequence of network traffic. *Anomaly*-based IDS detects unknown attacks by heuristics or rules [59]. P-DIFF aims at detecting access control policy changes (that may open up to attacks) instead of detecting the attacks. It can help backtrack the configuration change that permitted the detected intrusions.

4.3.3 Decision Tree Algorithms

Handling time-changing data, known as *concept drift adaption* in the literature, has been one of the main challenges in machine learning [38]. Concept drift means the underlying model of the data is changed along with time, i.e., an access control policy is changed from ALLOW to DENY in our case. Previously most works propose to build concept drift decision tree by learning multiple trees, each with the data in a time window [49, 103, 78]. However, it is hard to apply those approaches in the access-control policy change problem, due to the challenges of choosing the appropriate time window length, as different policies change can be performed by sysadmins at some random time. To address this problem, we propose a new TCDT learning algorithm which treats the whole dataset as a consecutive time series instead of discrete time windows and encodes all the policies along with their changes in a single decision tree. Although the TCDT is designed for the access-control policy change problem, it can be applied to other binary classification problem with concept drift.

4.3.4 Execution Acceleration

Existing techniques [111, 56, 106, 31] for accelerating execution focus on reducing the number of requests to execute. They aggregate similar requests, execute them in one round for the most parts and only split the executions when there are divergences on control or data flow. ACTESTs take an orthogonal approach. It minimizes the execution time of each individual request by trimming server programs. We observe that server programs typically perform access-control checks at the beginning of a request handling; therefore, we can skip the costly operations after the access-control checks. To make ACTESTs easy to deploy, our approach does not introduce any new dependency or record-and-replay systems as prior approaches. Combining our trimming and previous deduplication approaches may further reduce the testing execution time.

4.4 Acknowledgement

This chapter, in part, is a reprint of the material under submission. Xiang, Chengcheng; Mugnier, Eric; Nguyen, Nathaniel; Huang, Haochen; Zhong, Li; Zhou, Yuanyuan; Xu, Tianyin. The dissertation author was the primary investigator and author of this paper.

This chapter, in part, is a reprint of the material as it appears in Proceedings of the 26th ACM Conference on Computer and Communications Security, 2019. Xiang, Chengcheng; Wu, Yudong; Shen, Bingyu; Shen, Mingyao; Huang, Haochen; Xu, Tianyin; Zhou, Yuanyuan; Moore, Cindy; Jin, Xinxin; Sheng, Tianwei. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Conclusion

This dissertation presents two validation methods for access-control configurations. This dissertation presents ACTESTs to help sysadmins test their access-control configuration changes before the changes are deployed and presents P-DIFF to help sysadmins detect access-control behavior changes after configuration changes are deployed in production.

ACTESTs are a new type of test programs for testing access control configuration changes. ACTESTs can detect what requests are impacted by access-control configuration changes and warn sysadmins to validate if the impacts are intended. This can help sysadmins detect mis-configuration vulnerabilities before they are exploited by attackers. To help developers build ACTESTs, we present a general technique to transform a target program into an ACTEST. Our evaluation on real-world Docker images shows that ACTESTs detect 168 new vulnerabilities from 72 images. So far 25 of these vulnerabilities have been confirmed and 19 of them have been fixed by image maintainers. Our evaluation on five real-world deployed systems shows that ACTESTs can effectively and efficiently detect all the change impacts.

P-DIFF is a practical tool for continuously monitoring access logs to help sysadmins detect unintended access-control policy changes as well as help identify historical policy changes for a known security incident. We propose a novel TCDT structure and learning algorithm to automatically infer access policies and changes from access logs. We evaluate P-DIFF with access logs from five real-world systems. The results show that P-DIFF is effective in both detection

of access control policy changes and forensic investigation of security incidents. In addition, although our TCDT learning algorithm is only used for inferring access control policies in this dissertation, it can be generally adopted to address the challenges in inferring other policies with result changes.

Lessons Learned

Throughout the execution of this dissertation, a few lessons are learned that may be helpful to guide future works:

First, there is no single silver bullet to solve all access-control misconfiguration issues. This dissertation explores two approaches that aim to detect misconfigurations both before and after the production deployment of configurations. Though we try to make each approach as complete as possible, each of them definitely cannot catch all the misconfigurations. Therefore, future work may consider to further solve the problem from different perspectives instead of pushing one solution to the end. For enterprise, it may be a more effective strategy to adopt a variety of solutions instead of relying on one major solution.

Second, more work needs to be done on the “human side” for access-control validation. This dissertation focuses on detecting the behavior changes of *systems* after a configuration change. Along with our study of the topic, a new challenge emerges—how to present the behavior changes to *humans* so that the validation takes as little human effort as possible. This dissertation presents heuristics to rank the changes for validation. However, more human computer interaction studies need to be done on how to make the validation process effective and efficient.

Third, access logs need to be improved. Our study found that many programs still do not have complete access logs—they either miss important information (e.g., subject, object or action) or only record access denial events. This makes it hard for humans or tools to do auditing. To improve access logs, we present three guidelines: 1) To have a complete auditing, all access-related information needs to be logged. As the subject, object and action in different system layers are different, every layer may need to keep its own access logs. 2) As the amount of

information to be logged can be large, the performance penalty of logging needs to be understood. Future work needs to be done on measuring and reducing the performance overhead of recording access logs. 3) Access log formats should be designed for machines to read instead of for humans to read. As access logs are usually of a large amount, it is impossible for humans to manually analyze logs without a tool.

Bibliography

- [1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. Mining Association Rules Between Sets of Items in Large Databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD'93)*, May 1993.
- [2] Alexa Internet, Inc. Alexa traffic ranks. <https://www.alexa.com/siteinfo/wikipedia.org>, 2019.
- [3] Hussein Almuallim, Yasuhiro Akiba, and Shigeo Kaneda. On Handling Tree-Structured Attributes in Decision Tree Learning. In *Proceedings of the 12th International Conference on Machine Learning (ICML'95)*, July 1995.
- [4] Amos Jeffries. Squid proxy access log format. <https://wiki.squid-cache.org/Features/LogFormat>, 2015.
- [5] Ann Marie. Don't test in production? test in production! <https://opensource.com/article/19/5/dont-test-production>, 2019.
- [6] Apache. Apache. <https://github.com/apache/httpd/blame/cfd93e6c706e6119274cb10c3fcd0279e4b03759/server/core.c#L5153>, 2021.
- [7] Timothy G Armstrong, Vamsi Ponnkanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, 2013.
- [8] Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci. Usable declarative configuration specification and validation for applications, systems, and cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*, pages 29–35, 2017.
- [9] Lujo Bauer, Lorrie Faith Cranor, Robert W. Reeder, Michael K. Reiter, and Kami Vaniea. Real life challenges in access-control management. In *Proceedings of the 2009 CHI Conference on Human Factors in Computing Systems*, 2009.
- [10] Lujo Bauer, Scott Garriss, and Michael K Reiter. Detecting and resolving policy misconfigurations in access-control systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT'08)*, 2008.

- [11] Lujo Bauer, Scott Garriss, and Michael K Reiter. Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):2, 2011.
- [12] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. *Classification and Regression Trees*. Wadsworth Publishing, 1983.
- [13] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 209–223, 2020.
- [14] Capital One Tech. Container adoption statistics: The future of the container market. <https://www.capitalone.com/tech/cloud/container-adoption-statistics/>, 2020.
- [15] Huseyin Cavusoglu, Birendra Mishra, and Srinivasan Raghunathan. A model for evaluating it security investments. *Communications of the ACM*, 47(7):87–92, 2004.
- [16] Qingrong Chen, Teng Wang, Owolabi Legunsen, Shanshan Li, and Tianyin Xu. Understanding and discovering software configuration dependencies in cloud and datacenter systems. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 362–374, 2020.
- [17] Anton Chuvakin and Gunnar Peterson. How to Do Application Logging Right. *IEEE Security & Privacy*, 8(4):82–85, July 2010.
- [18] Cindy Sridharan. Testing in production, the safe way. <https://copyconstruct.medium.com/testing-in-production-the-safe-way-18ca102d0ef1>, 2018.
- [19] Cloudera, Inc. Hadoop audit event. https://docs.cloudera.com/documentation/enterprise/5-4-x/topics/cn_iu_audits.html, 2019.
- [20] CNN Business. A hacker gained access to 100 million capital one credit card applications and accounts. <https://www.cnn.com/2019/07/29/business/capital-one-data-breach/index.html>, 2019.
- [21] Carlos Cotrini, Thilo Weghorn, and David Basin. Mining ABAC Rules from Sparse Logs. In *Proceedings of the 3rd European Symposium on Security and Privacy (EuroS&P’18)*, April 2018.
- [22] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security’09)*, August 2009.
- [23] Tathagata Das, Ranjita Bhagwan, and Prasad Naldurg. Baaz: A System for Detecting Access Control Misconfigurations. In *Proceedings of the 19th USENIX Security Symposium (USENIX Security’10)*, August 2010.

- [24] Datadog. 8 surprising facts about real docker adoption. <https://www.datadoghq.com/docker-adoption/>, 2018.
- [25] Datadog. 11 facts about real-world container use. <https://www.datadoghq.com/container-report/>, 2020.
- [26] Dorothy E Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, (2):222–232, 1987.
- [27] DivvyCloud. 2020 cloud misconfigurations report. <https://divvycloud.com/wp-content/uploads/2020/02/Cloud-Misconfiguration-Report-FINAL.pdf>, 2020.
- [28] Docker. Dockerhub. <https://hub.docker.com/search?q=&type=image>, 2021.
- [29] Docker. Overlay networks. <https://docs.docker.com/network/overlay/>, 2021.
- [30] dokuwiki. Dokuwiki. <https://www.dokuwiki.org/dokuwiki>, 2021.
- [31] Xianzheng Dou, Peter M Chen, and Jason Flinn. Shortcut: accelerating mostly-deterministic code regions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 570–585, 2019.
- [32] Drupal. Drupal - open source cms. <https://www.drupal.org/>, 2020.
- [33] Drupal. Drupal modules. https://www.drupal.org/project/project_module, 2021.
- [34] Dennis Fetterly, Maya Haridasan, Michael Isard, and Swaminathan Sundararaman. Tidyfs: A simple and small distributed file system. In *USENIX annual technical conference*, pages 34–34, 2011.
- [35] Kathi Fisler, Shriram Krishnamurthi, Leo A Meyerovich, and Michael Carl Tschantz. Verification and Change-Impact Analysis of Access-Control Policies. In *Proceedings of the 27th International Conference on Software Engineering (ICSE’05)*, May 2005.
- [36] Forbes. Microsoft security shocker as 250 million customer records exposed online. <https://www.forbes.com/sites/daveywinder/2020/01/22/microsoft-security-shocker-as-250-million-customer-records-exposed-online/?sh=58e328974d1b>, 2019.
- [37] Gadget 360. Bizongo data leak exposed details of customers making online purchases: Researchers. <https://gadgets.ndtv.com/internet/news/bizongo-data-breach-server-misconfiguration-amazon-flipkart-swiggy-zomato-customer-details-leak-2412565>, 2021.
- [38] João Gama, Indrè Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A Survey on Concept Drift Adaptation. *ACM Computing Surveys (CSUR)*, 46(4):44, 2014.
- [39] François Gauthier, Dominic Letarte, Thierry Lavoie, and Ettore Merlo. Extraction and comprehension of moodle’s access control model: A case study. In *Proceedings of the 9th Annual International Conference on Privacy, Security and Trust*, July 2011.

- [40] GCC. gcov—a test coverage program. <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>, 2012.
- [41] hackerone. Potential server misconfiguration leads to disclosure of vendor/ directory. <https://hackerone.com/reports/271391>, 2017.
- [42] hackerone. Sensitive data disclosure via exposed phpunit file. <https://hackerone.com/reports/543775>, 2020.
- [43] Tyler Harter, Dhruva Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Analysis of hdfs under hbase: A facebook messages case study. In *12th USENIX Conference on File and Storage Technologies (FAST 14)*, pages 199–212, 2014.
- [44] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR, 1994.
- [45] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. *Applied logistic regression*, volume 398. John Wiley & Sons, 2013.
- [46] Vincent C Hu, Rick Kuhn, and Dylan Yaga. Verification and test methods for access control policies/models. *NIST Special Publication*, 800:192, 2017.
- [47] Peng Huang, William J Bolosky, Abhishek Singh, and Yuanyuan Zhou. Confvalley: A systematic configuration validation framework for cloud services. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–16, 2015.
- [48] Graham Hughes and Tevfik Bultan. Automated verification of access control policies using a sat solver. *International journal on software tools for technology transfer*, 10(6):503–520, 2008.
- [49] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'01)*, August 2001.
- [50] Junbeom Hur and Dong Kun Noh. Attribute-based access control with efficient revocation in data outsourcing systems. *IEEE Transactions on Parallel and Distributed Systems*, 22(7):1214–1221, 2011.
- [51] IBM Security. Cost of a data breach report 2020. <https://www.capita.com/sites/g/files/nginej146/files/2020-08/Ponemon-Global-Cost-of-Data-Breach-Study-2020.pdf>, 2020.
- [52] IETF. Virtual extensible local area network (vxlan). <https://tools.ietf.org/html/rfc7348>, 2014.
- [53] Karthick Jayaraman, Vijay Ganesh, Mahesh Tripunitara, Martin Rinard, and Steve Chapin. Automatic error finding in access-control policies. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 163–174, 2011.

- [54] Somesh Jha, Ninghui Li, Mahesh Tripunitara, Qihua Wang, and William Winsborough. Towards formal verification of role-based access control policies. *IEEE Transactions on Dependable and Secure Computing*, 5(4):242–255, 2008.
- [55] joomla. Joomla content management system (cms). <https://www.joomla.org/>, 2021.
- [56] Taesoo Kim, Ramesh Chandra, and Nickolai Zeldovich. Efficient patch-based auditing for web application vulnerabilities. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 193–206, 2012.
- [57] Butler Lampson. Usable Security: How to Get It. *Communications of the ACM*, 52(11):25–27, November 2009.
- [58] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.
- [59] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. Intrusion Detection System: A Comprehensive Review. *Journal of Network and Computer Applications*, 36(1):16–24, 2013.
- [60] Linux. ipv6: Implement any-ip support for ipv6. <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=ab79ad14a2d51e95f0ac3cef7cd116a57089ba828>, 2010.
- [61] Linux. Overlay filesystem. <https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html>, 2021.
- [62] Luke Irwin. How long does it take to detect a cyber attack? <https://www.itgovernanceusa.com/blog/how-long-does-it-take-to-detect-a-cyber-attack/>, 2019.
- [63] Evan Martin and Tao Xie. Automated Test Generation for Access Control Policies via Change-Impact Analysis. In *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*, 2007.
- [64] Evan Martin and Tao Xie. A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web*, pages 667–676, 2007.
- [65] Roy A. Maxion and Robert W. Reeder. Improving User-Interface Dependability through Mitigation of Human Error. *International Journal of Human-Computer Studies*, 63(1-2):25–50, July 2005.
- [66] McAfee. Cloud adoption and risk report 2019. https://www.dlt.com/sites/default/files/resource-attachments/2019-09/Cloud-Cloud-Adoption-%2526-Risk-Report-2019_0_13.pdf, 2019.
- [67] MediaWiki. enwiki dump progress on 20180420. ”<https://dumps.wikimedia.org/enwiki/20180420/>”, Apr. 2018.

- [68] Sonu Mehta, Ranjita Bhagwan, Rahul Kumar, Chetan Bansal, Chandra Maddila, B Ashok, Sumit Asthana, Christian Bird, and Aditya Kumar. Rex: Preventing bugs and misconfiguration in large services using correlated change analysis. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 435–448, 2020.
- [69] Mark S Merkow and Jim Breithaupt. *Information security: Principles and practices*. Pearson Education, 2014.
- [70] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Le Traon. A model-based framework for security policy specification, deployment and testing. In *Proceedings of the 7th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML08)*, October 2008.
- [71] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *2011 IEEE Symposium on Security and Privacy*, pages 165–179. IEEE, 2011.
- [72] Joseph P Near and Daniel Jackson. Finding Security Bugs in Web Applications using a Catalog of Access Control Patterns. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, May 2016.
- [73] Night lion security. Astoria company data breach research and analysis. <https://www.nightlion.com/blog/2021/astoria-company-breach/>, 2021.
- [74] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, 2013.
- [75] NumPy developers. Numpy. <https://www.numpy.org>, 2018.
- [76] Oracle. Mysql audit log file formats. <https://dev.mysql.com/doc/refman/8.0/en/audit-log-file-formats.html>, 2019.
- [77] Pandas. pandas: Python data analysis library. <https://pandas.pydata.org/>, 2018.
- [78] Lena Pietruczuk, Piotr Duda, and Maciej Jaworski. Adaptation of decision trees for handling concept drift. In *International Conference on Artificial Intelligence and Soft Computing*, pages 459–473. Springer, 2013.
- [79] Rahul Potharaju, Joseph Chan, Luhui Hu, Cristina Nita-Rotaru, Mingshi Wang, Liyuan Zhang, and Navendu Jain. Confseer: leveraging customer support knowledge bases for automated misconfiguration detection. *Proceedings of the VLDB Endowment*, 8(12):1828–1839, 2015.
- [80] Alexander Pretschner, Tejeddine Mouelhi, and Yves Le Traon. Model-based tests for access control policies. In *2008 1st International Conference on Software Testing, Verification, and Validation*, pages 338–347. IEEE, 2008.

- [81] pure-ftpd. pure-ftpd - linux man page. <https://linux.die.net/man/8/pure-ftpd>, 2017.
- [82] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. Trusted computer system evaluation criteria. In *National Computer Security Center*. Citeseer, 1985.
- [83] J. Ross Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986.
- [84] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, Inc., 1993.
- [85] Charles M Rader. Discrete Convolutions via Mersenne Transforms. *IEEE Transactions on Computers*, 100(12):1269–1273, 1972.
- [86] Robert W. Reeder, Lujo Bauer, Lorrie Faith Cranor, Michael K. Reiter, Kelli Bacon, Keisha How, and Heather Strong. Expandable Grids for Visualizing and Authoring Computer Security Policies. In *Proceedings of the 26th ACM Conference on Human Factors in Computing Systems (CHI'08)*, Florence, Italy, April 2008.
- [87] Robert Christiansen. More enterprises are using containers; here's why. — cio. <https://www.cio.com/article/3434010/more-enterprises-are-using-containers-here-s-why.html>, 2019.
- [88] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.
- [89] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. Role-Based Access Control Models. *IEEE Computer*, 29(2):38–47, 1996.
- [90] Mark Santolucito, Ennan Zhai, Rahul Dhodapkar, Aaron Shim, and Ruzica Piskac. Synthesizing configuration file specifications with association rule learning. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–20, 2017.
- [91] Bruce Schneier. Real-World Access Control. https://www.schneier.com/blog/archives/2009/09/real-world_acce.html, 2009.
- [92] Security Boulevard. Misconfiguration leaks 138gb of information to the public. <https://securityboulevard.com/2021/03/misconfiguration-leaks-138gb-of-information-to-the-public/>, 2021.
- [93] Security World. 9 years to discover a data breach. <https://www.secureworldexpo.com/industry-news/9-years-incident-to-breach-discovery-time>, 2019.
- [94] SELinux. Selinux auditing events. https://selinuxproject.org/page/NB_AL, 2014.
- [95] ServerFault. How to test an alternate configuration with exim4? <https://serverfault.com/questions/230027/how-to-test-an-alternate-configuration-with-exim4>, 2011.

- [96] ServerFault. How to test configuration changes before they go live? <https://serverfault.com/questions/696962/how-to-test-configuration-changes-before-they-go-live>, 2015.
- [97] Riaz Ahmed Shaikh, Kamel Adi, and Luigi Logrippo. A Data Classification Method for Inconsistency and Incompleteness Detection in Access Control Policy Sets. *International Journal of Information Security*, 16(1):91–113, 2017.
- [98] Silicon Angle. Pharma giant pfizer exposes patient data on unsecured cloud storage. <https://siliconangle.com/2020/10/20/pharma-giant-pfizer-exposes-patient-data-unsecured-cloud-storage/>, 2020.
- [99] Sara Sinclair and Sean W. Smith. What’s Wrong with Access Control in the Real World? *IEEE Security & Privacy*, 8(4):74–77, July 2010.
- [100] Sara Sinclair, Sean W. Smith, Stephanie Trudeau, M. Eric Johnson, and Anthony Portera. Information Risk in Financial Institutions: Field Study and Research Roadmap. In *Proceedings for the 3rd International Workshop on Enterprise Applications and Services in the Finance Industry (FinanceCom’07)*, Montreal, Canada, December 2007.
- [101] Sophos. The state of cloud security 202. <https://secure2.sophos.com/en-us/medialibrary/Gated-Assets/white-papers/sophos-the-state-of-cloud-security-2020-wp.pdf>, 2020.
- [102] Spark. Spark mllib. <https://spark.apache.org/docs/latest/ml-guide.html>, 2018.
- [103] Kenneth O Stanley. Learning Concept Drift with a Committee of Decision Trees. Technical Report UT-AI-TR-03-302, Department of Computer Sciences, The University of Texas at Austin, 2003.
- [104] Fangqi Sun, Liang Xu, and Zhendong Su. Static Detection of Access Control Vulnerabilities in Web Applications. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security’11)*, August 2011.
- [105] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. Testing configuration changes in context to prevent production failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 735–751, 2020.
- [106] Cheng Tan, Lingfan Yu, Joshua B Leners, and Michael Walfish. The efficient server audit problem, deduplicated re-execution, and the web. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 546–564, 2017.
- [107] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. Holistic configuration management at facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 328–343, 2015.

- [108] Tech Times . Wordpress data breach affects 100,000 exposed websites after using responsive menu plugin. <https://www.techtimes.com/articles/257016/20210212/wordpress-data-breach-affects-100-000-exposed-websites-using-responsive.htm>, 2021.
- [109] Jeff Terrace and Michael J Freedman. Object storage on craq: High-throughput chain replication for read-mostly workloads. In *USENIX Annual Technical Conference*, number June, pages 1–16. San Diego, CA, 2009.
- [110] The Apache Software Foundation. Apache2 access log format. <https://httpd.apache.org/docs/current/logs.html>, 2019.
- [111] Joseph Tucek, Weiwei Xiong, and Yuanyuan Zhou. Efficient online validation with delta execution. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 193–204, 2009.
- [112] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks*, 53(11):1830–1845, July 2009. http://www.globule.org/publi/WWADH_comnet2009.html.
- [113] Verizon. 2020 verizon data breach investigations report,. <https://enterprise.verizon.com/content/verizonenterprise/us/en/index/resources/reports/2020-data-breach-investigations-report.pdf>, 2020.
- [114] Rui Wang, XiaoFeng Wang, Kehuan Zhang, and Zhuowei Li. Towards Automatic Reverse Engineering of Software Security Configurations. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS’08)*, October 2008.
- [115] Ryan West. The Psychology of Security. *Communications of the ACM*, 51(4):34–40, April 2008.
- [116] Wikimedia Foundation . Extensions let you customize how mediawiki looks and works. <https://www.mediawiki.org/wiki/Manual:Extensions>, 2021.
- [117] Wikimedia Foundation. Mediawiki is a collaboration and documentation platform brought to you by a vibrant community. <https://www.mediawiki.org/wiki/MediaWiki>, 2020.
- [118] Wikipedia. One-hot. <https://en.wikipedia.org/wiki/One-hot>, 2018.
- [119] Wikipedia. Help:log. <https://en.wikipedia.org/wiki/Help:Log>, 2021.
- [120] Wired. Hack brief: An adult cam site exposed 10.88 billion records. <https://www.wired.com/story/cam4-adult-cam-data-leak-7tb/>, 2020.
- [121] Wordpress . Extend your wordpress experience with 58,390 plugins. <https://wordpress.org/plugins/>, 2021.
- [122] wordpress. Wordpress.com: Create a free website or blog. <https://wordpress.com/read>, 2021.

- [123] Xdebug. Code coverage analysis. https://xdebug.org/docs/code_coverage, 2021.
- [124] Chengcheng Xiang, Haochen Huang, Andrew Yoo, Yuanyuan Zhou, and Shankar Pasupathy. Pracextractor: Extracting configuration good practices from manuals to detect server misconfigurations. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 265–280, 2020.
- [125] Chengcheng Xiang, Yudong Wu, Bingyu Shen, Mingyao Shen, Haochen Huang, Tianyin Xu, Yuanyuan Zhou, Cindy Moore, Xinxin Jin, and Tianwei Sheng. Towards continuous access control validation and forensics. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 113–129, 2019.
- [126] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. Early detection of configuration errors to reduce failure damage. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*, November 2016.
- [127] Tianyin Xu, Han Min Naing, Le Lu, and Yuanyuan Zhou. How Do System Administrators Resolve Access-Denied Issues in the Real World? In *Proceedings of the 35th Annual CHI Conference on Human Factors in Computing Systems (CHI'17)*, May 2017.
- [128] Tianyin Xu, Vineet Pandey, and Scott Klemmer. An HCI View of Configuration Problems. *arXiv:1601.01747*, January 2016.
- [129] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)*, November 2013.
- [130] Tianyin Xu and Yuanyuan Zhou. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys (CSUR)*, 47(4), July 2015.
- [131] Yuu Yamada, Einoshin Suzuki, Hideto Yokoi, and Katsuhiko Takabayashi. Decision-tree Induction from Time-series Data Based on a Standard-example Split Test. In *Proceedings of the 20th International Conference on Machine Learning (ICML'03)*, August 2003.
- [132] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. An Empirical Study on Configuration Errors in Commercial and Open Source Systems. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*, October 2011.
- [133] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*, October 2009.
- [134] ZDNet. Database leaks data on most of Ecuador's citizens, including 6.7 million children. <https://www.zdnet.com/article/database-leaks-data-on-most-of-ecuadors-citizens-including-6-7-million-children/>, 2019.

- [135] Zero Day. Hacker ransoms 23k mongodb databases and threatens to contact gdpr authorities. <https://www.zdnet.com/article/hacker-ransoms-23k-mongodb-databases-and-threatens-to-contact-gdpr-authorities/>, 2020.
- [136] Jiaqi Zhang, Lakshmi Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. EnCore: Exploiting System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*, March 2014.
- [137] Jun Zhang and Vasant Honavar. Learning Decision Tree Classifiers from Attribute Value Taxonomies and Partially Specified Data. In *Proceedings of the 20th International Conference on Machine Learning (ICML'03)*, August 2003.
- [138] Sai Zhang, David Saff, Yingyi Bu, and Michael D Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 353–363, 2011.