# 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[†]

University of California, San Diego      *University of Illinois Urbana-Champaign      [†]Whova, Inc.

## ABSTRACT

Access control is often reported to be "profoundly broken" in real-world practices due to prevalent policy misconfigurations introduced by system administrators (sysadmins). Given the dynamics of resource and data sharing, access control policies need to be continuously updated. Unfortunately, to err is human—sysadmins often make mistakes such as over-granting privileges when changing access control policies. With today's limited tooling support for continuous validation, such mistakes can stay unnoticed for a long time until eventually being exploited by attackers, causing catastrophic security incidents.

We present P-DIFF, a practical tool for monitoring access control behavior to help sysadmins early detect unintended access control policy changes and perform postmortem forensic analysis upon security attacks. P-DIFF continuously monitors access logs and infers access control policies from them. To handle the challenge of policy evolution, we devise a novel time-changing decision tree to effectively represent access control policy changes, coupled with a new learning algorithm to infer the tree from access logs. P-DIFF provides sysadmins with the inferred policies and detected changes to assist the following two tasks: (1) validating whether the access control changes are intended or not; (2) pinpointing the historical changes responsible for a given security attack.

We evaluate P-DIFF with a variety of datasets collected from five real-world systems, including two from industrial companies. P-DIFF can detect 86%–100% of access control policy changes with an average precision of 89%. For forensic analysis, P-DIFF can pinpoint the root-cause change that permits the target access in 85%–98% of the evaluated cases.

## CCS CONCEPTS

• **Security and privacy** → **Access control**; *Web application security*; • **Applied computing** → **System forensics**; • **Computing methodologies** → **Machine learning approaches**;

## KEYWORDS

Access control; misconfiguration; policy change; access logs; forensics; decision tree

## 1 INTRODUCTION

### 1.1 Motivation

As the *de facto* mechanism for protecting computer systems against unauthorized access, access control has been reported as "profoundly broken" due to prevalent misconfigurations introduced by system administrators (sysadmins) [15, 31, 55, 68]. In recent years, access control misconfigurations have become one major cause of security incidents such as data theft and system compromises, as quantified by security analysis reports [24, 61] and exemplified by the newsworthy security incidents listed in Table 1.

| Time | Incident | Organization |
|---|---|---|
| 2017.6 | 198 million US voter records leaked [39] | Deep Root Analytics |
| 2017.7 | 14 million customer records leaked [42] | Verizon |
| 2017.9 | Half million vehicle records leaked [28] | SVR Tracking |
| 2018.2 | 119,000+ personal IDs exposed [29] | FedEx |
| 2018.3 | 42,000 patients information leaked [17] | Huntington hospital |
| 2018.4 | 63,551 patients records breached [16] | Middletown medical |
| 2019.1 | 24 million financial records leaked [19] | Ascension |
| 2019.9 | 20 million citizen records exposed [76] | Novaestrat |

**Table 1: Recent publicly-reported security incidents caused by access control misconfigurations.**

One of the key missing pieces in today's access-control management is *continuous behavior validation*, which allows sysadmins to validate whether a system behaves as they expect after a configuration change. Access control configuration is never a one-time effort but requires continuous policy changes to accommodate the dynamics of data and resource sharing, as well as high churn and updates of new protection domains and organizational roles. [1] The following lists a few common cases when sysadmins need to modify the access control policy 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 may start to contain sensitive information.

---

[1]We define an access control *policy* to be a complete set of access control *rules*. Each rule is codified by access control *configurations*.

- *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, to accommodate changes listed above, sysadmins may introduce misconfigurations. Accommodating the changes may not be easy, because changing the access control policy may involve modification to multiple components, such as web server, application server, database and file system, etc. Meanwhile, sysadmins may have time pressure when they are requested to make the changes. One common case is about handling access-denied issue: 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 privilege or grant additional unexpected permissions. In this case, such changes are prone to be access control misconfigurations. A recent analysis on the resolutions of real-world access-denied issues shows that 38.1% of the changes introduced misconfigurations that over-grant permissions and create vulnerabilities [68].

Despite a number of efforts on testing and verifying access control configurations [8, 9, 15, 20, 34, 37, 54], it is still prohibitively difficult to eliminate all access control misconfigurations in real-world systems (§12). Specifically, state-of-the-art detection tools for access control misconfiguration [8, 9, 15] mostly detect inconsistencies of configurations and cannot understand configurations at a higher policy level. As noted in [15], 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 [71, 73], access control misconfigurations cannot be manifested through observable symptoms (e.g., crashing behavior, dysfunctions, or performance degradation). According to a recent report [33], it takes 206 days on average for US companies to detect a data breach, which is too late for any remedies.

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 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 paper 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.

## 1.2 Contributions

This paper presents P-DIFF, a practical tool for inferring access control behavior and behavior changes from *access logs*. As we will show in §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. §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

**(a) Heterogenous configuration and code implementation of access control policies across different components**

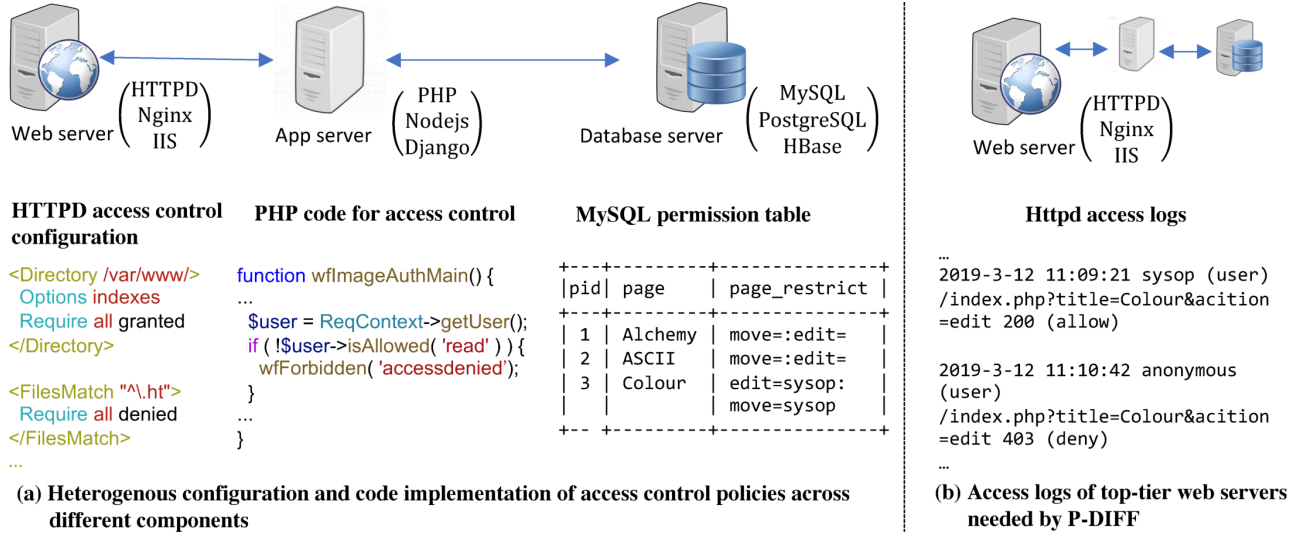**(b) Access logs of top-tier web servers needed by P-DIFF**

**Figure 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.**

inference algorithms cannot decide whether "ABC" is a decisive attribute in the rules because of the related result changes (cf. §8).

To address the second challenge, we extend the traditional decision tree to support time-series information, referred in this paper 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. §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 86%–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.

## 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.

### 2.1 Access-Control Configurations

Despite the uniform model of access control (e.g., *access-control matrix* [30]), 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) [46] to restrict access to files based on the identity of users and groups. The Apache web server adopts *attribute-based* access control (ABAC) [27], 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) [51] 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.

### 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 $< S, O, A, R >$ 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 2 shows access log formats of nine different software systems of various types [4, 12, 41, 45, 53, 60]. It shows that most

| 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 |

**Table 2: Information encoded in access logs of different software. S = subject, O = object, A = action, R = result.**

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 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 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).

## 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 §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 DESIGN DECISIONS

### 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 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 [55, 56]. 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 §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 [69, 70]. 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 [64], 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 [22], let alone analyzing the interactions between multiple components. On the other hand, access logs have simple and clear semantics, as discussed in §2.2.

### 3.2 Using Decision Tree Based Models

As discussed in §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 [47], Association rule learning [1], Logistic Regression [25] and Neural Networks [23]. 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 §2.3, an access control policy consists of multiple steps of decision-making and may contain decisions on
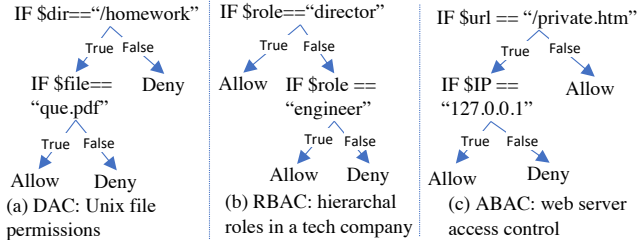
Figure 2: Decision tree representations of three access control models. The tree structure can effectively represent access control hierarchies for (a) files, (b) roles, and (c) URLs with multiple decision steps based on different attributes.

hierarchical attributes, which can naturally be represented by a decision tree. Figure 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.

## 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 [13] 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 $< S, O, A, R >$ (§2.2) can be observed from historical logs. In other words, one cannot train a classifier with the complete dataset of every $< S, O, A, R >$ 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, 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 §7.

## 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 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 [7–9, 15, 52, 55, 56, 68], 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
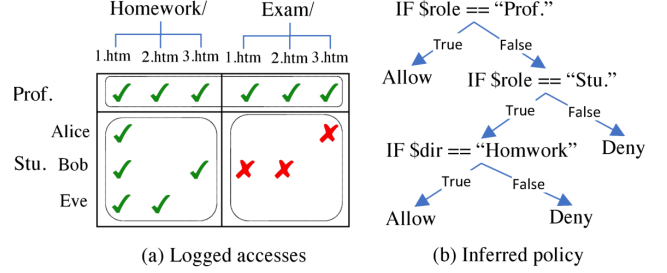


Figure 3: Example of sparse accesses to a course website and the decision tree inferred from them. In (a), a green|red mark means the access was allowed|denied. A vacancy means the access did not happen, and thus the policy is not reflected in the accesses. To address it, our learning algorithm groups vacancy with green|red marks and infers a group policy as shown in (b).

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

## 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 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 §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.
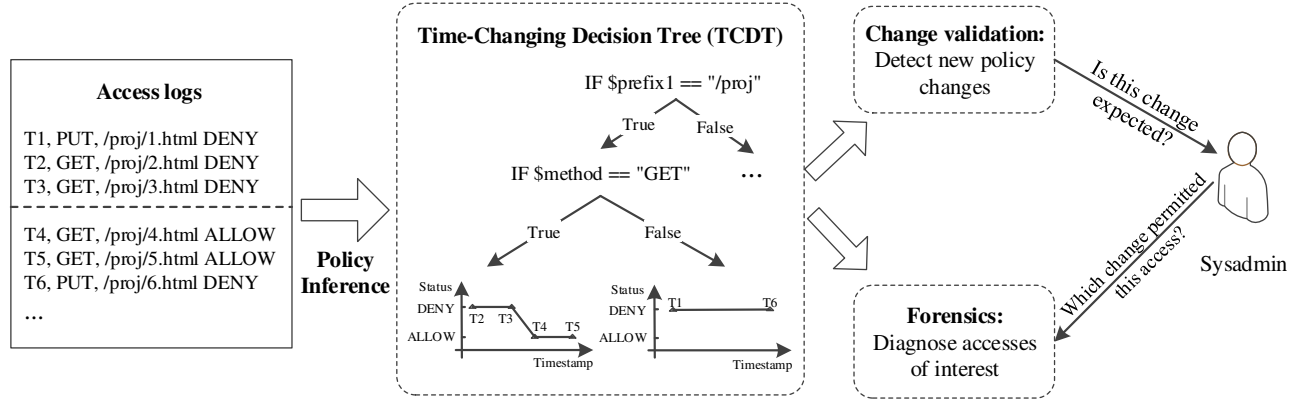
**Figure 4: The workflow of P-DIFF. P-DIFF infers access control policies from access logs. It maintains the inferred policies in a data structured named Time-Changing Decision Tree which records the entire change history. P-DIFF supports two uses cases, change validation and forensics analysis, as elaborated in §5.**

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 §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. §12 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 [40, 43]. It can be deployed on different platforms that support Python.

## 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. §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 [47]. 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 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.
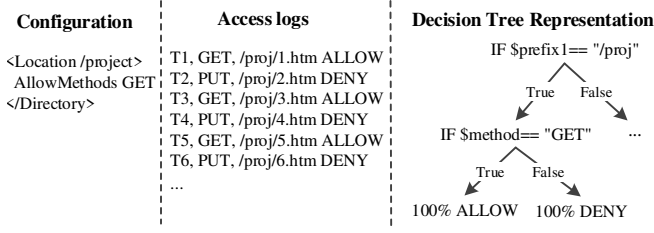
Figure 5: An example of access-control policies in configuration files, access logs, and a decision tree. T1-T6 are the timestamps.
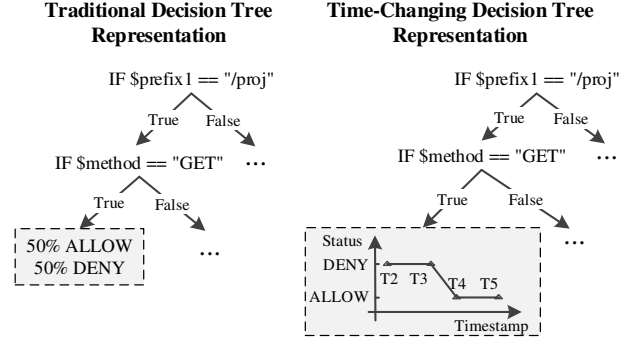


Figure 6: A traditional decision tree and a Time-Changing Decision Tree (TCDT) generated from the logs in Figure 4. Both decision trees have the same internal nodes; however, in a traditional decision tree, the leaf nodes are associated with proportional results; in a TCDT, the leaf nodes are associated with the time-series results which can be used to represent policy changes.

In order to maintain the evolution history of access control policies, we design TCDT. A TCDT has a similar exterior structure as a traditional DT. Differently, TCDT makes each leaf node encodes a time series $T$:

$$T = ((\tau_1, r_1), (\tau_2, r_2), \cdots, (\tau_n, r_n)) \quad (1)$$

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

Figure 6 shows a TCDT generated from access logs in Figure 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 §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 [72]. 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 categories. We position TCDT in the machine learning literature and discuss the related work in detail in §12.

***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 §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 12 in §10.7).

## 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 §8.

### 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 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. §7.3).

### 7.2 Policy Learning Algorithm

P-DIFF uses a classic decision tree learning algorithm [47] 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

$$L = \{(x_{i_1}, \ldots, x_{i_n}, y) | i \in [1, m]\} \quad (2)$$

where $(x_{i_1}, \ldots, x_{i_n})$ 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 $(x_{i_1}, \ldots, x_{i_n})$. 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 §7.3

| 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 |

**Table 3: 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.**

and transforms the expanded features into a feature vector with one-hot encoding [66].

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

$$L_l = \{(x_{k_1}, \ldots, x_{k_n}, y) | k \in [1, m] \wedge x_{k_j} = x_{i_j}\} \quad (3)$$

$$L_r = L - L_l \quad (4)$$

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 [10, 47] as the metric. We will show that those metrics cannot handle policy changes in §8, and the new metric we design for P-DIFF to learn TCDTs.

### 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 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) 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 [77] for the best-split step (Algorithm 1, Line 3) in the decision tree learning. Specifically, P-DIFF follows a hierarchical order to choose a feature that best-splits the input data. Let us assume that there exist three

---

**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** $mg\,!= 0$ **then**
7:         $root.left \leftarrow DTL(L_l)$
8:         $root.right \leftarrow DTL(L_r)$
9:     **return** $root$

[a] $L = \{(x_{i_1}, \ldots, x_{i_n}, y) | i \in [1, m]\}$, the training data.
[b] Find 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.
[c] Split $L$ into $L_l = \{(x_{k_1}, \ldots, x_{k_n}, y) | k \in [i, m] \wedge x_{k_j} = x_{i_j}\}$ and $L_r = L - L_l$.
[d] Calculate $metric(L_l) + metric(L_r) - metric(L)$, where metric is a function measures the label purity of a set, e.g. entropy or Gini Impurity.

---

attributes `["user","method", "file path"]`. P-DIFF first expands the three attributes to five features in the feature vector: `["user","method","prefix0", "prefix1","file name"]`. P-DIFF then 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.

## 8 POLICY CHANGE MANAGEMENT

### 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 (§6).

Moreover, traditional algorithms (e.g., CART, ID3 and C4.5 [10, 47, 48]) 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 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 sequence:

$$\mathcal{L} = ((\tau_1, x_{1_1} \ldots x_{1_n}, y_1), \ldots, (\tau_m, x_{m_1} \ldots x_{m_n}, y_m)) \quad (5)$$

where $\tau_i < \tau_j$ for $i < j$, $x_{i_1}, \ldots, x_{i_n}$ 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 7.
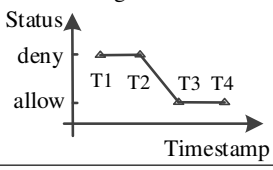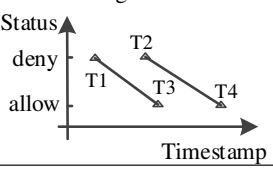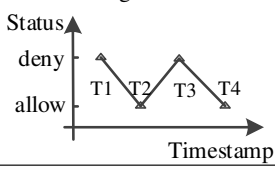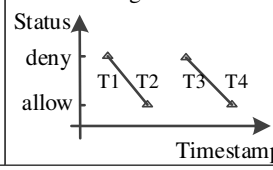
| | Case 1: A single rule change (should not split) | | Case 2: Multiple rule changes (should split) | |
|---|---|---|---|---|
| **Policy Change** | GET, /proj/* DENY$\rightarrow$ALLOW | | GET, /proj/1.htm DENY$\longrightarrow$ ALLOW<br>GET, /proj/2.htm DENY$\longrightarrow$ ALLOW | |
| **Access log subset** | T1, GET, /proj/1.htm DENY<br>T2, GET, /proj/2.htm DENY<br>T3, GET, /proj/1.htm ALLOW<br>T4, GET, /proj/2.htm ALLOW | | T1, GET, /proj/1.htm DENY<br>T2, GET, /proj/1.htm ALLOW<br>T3, GET, /proj/2.htm DENY<br>T4, GET, /proj/2.htm ALLOW | |
| | If not split | If split | If not split | If split |
| **Purity metrics** | $p_{allow}$: 0.5, $p_{deny}$: 0.5<br>Gini Impurity: 0.5<br>Entropy: 1 | $p_{allow\_left}$: 0.5, $p_{allow\_right}$: 0.5<br>Gini Impurity: 0.5<br>Entropy: 1 | $p_{allow}$: 0.5, $p_{deny}$: 0.5<br>Gini Impurity: 0.5<br>Entropy: 1 | $p_{allow\_left}$: 0.5, $p_{allow\_right}$: 0.5<br>Gini Impurity: 0.5<br>Entropy: 1 |
| **Time Series** | ChangeCount: 1 | ChangeCount: 2 | ChangeCount: 3 | ChangeCount: 2 |

Figure 7: Examples that demonstrate splitting events in TCDT-based policy learning (cf. §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.

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:

$$CC(\mathcal{L}) = \sum_{i=1}^{n-1} |y_{i+1} - y_i| \qquad (6)$$

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:

$$i, j = \operatorname*{argmax}_{i \in [1,m], j \in [1,n]} (CC(\mathcal{L}) - \sum_{\mathcal{L}_k \in split(\mathcal{L}, j, x_{i_j})} CC(\mathcal{L}_k)) \qquad (7)$$

where $split(\mathcal{L}, j, x_{i_j})$ 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 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 8.

## 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 $x_{i_i}$ 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 §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 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 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 [50] 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 [40].

With these two optimizations, our implementation of P-DIFF is able to handle the entire Wikipedia dataset. Our evaluation in §10.7 shows P-DIFF only takes 19 minutes to train a model from 320 million log entries.

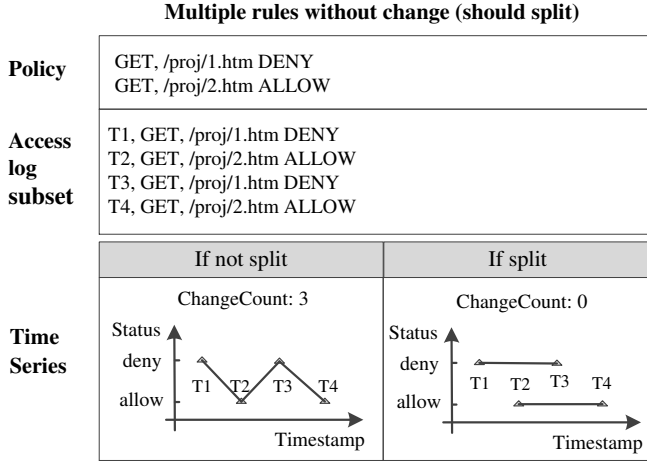**Multiple rules without change (should split)**



Figure 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.

## 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 §6–§8. In this section, we show how to use the learned TCDT as a policy evolution representation for supporting the two use cases.

### 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 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 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.

### 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
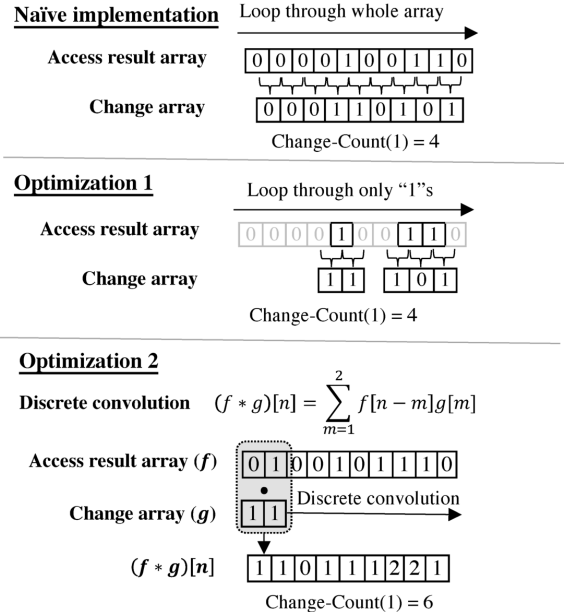


Figure 9: Two optimizations to efficiently calculate change-count. The naïve implementation is to loop through the whole access result array and count the changes. Optimization 1 improves it by only looping through the seldom occurred DENYs (value 1). Optimization 2 uses discrete convolution [50] $(f * g)[n]$ to calculate the sum of every two adjacent numbers (if the sum is 1, there is a change).

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 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.

## 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 the Wikipedia website, the firewall of a software company, and three websites hosted by academic organizations. Table 4 describes these datasets.

### 10.1 Systems and Datasets

- `Wikipedia`. A free online encyclopedia website that has 33 million registered users. We collect access logs from a public dataset of request traces to Wikipedia in September 2007 [62], the largest trace dataset we can find online. Its access control is implemented by the MediaWiki wiki engine [35]. Its protected
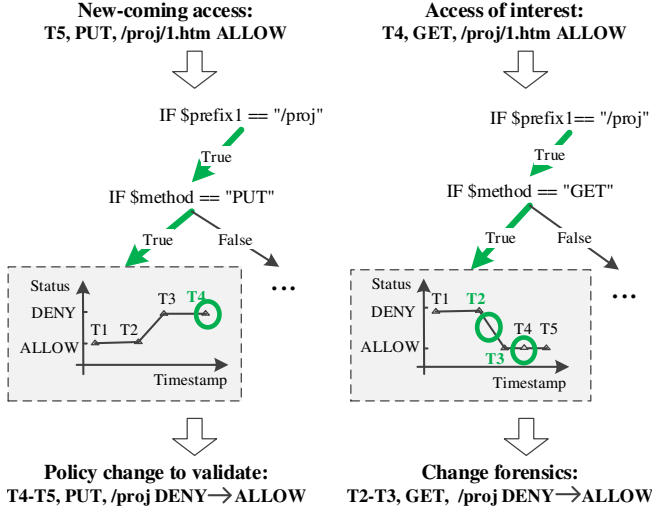
**Figure 10: Two use cases with P-DIFF: (a) change validation and (b) forensic analysis. For (a), P-DIFF detects a policy change based on the deviation of the access results at T5 and the policy maintained in TCDT. P-DIFF then notifies the sysadmins with the policy change for validation. For (b), P-DIFF backtracks the time series at the leaf node to pinpoint the root-cause change between T2 and T3 that permitted the access.**

resources include 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 web server hosting home pages, online tools and personal pages for a research center with more than 10 faculty members and 50 graduate students in a research university. Its resource protection is through the configuration of the Apache HTTPD server (Figure 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 website hosting 300+ courses each year. Its resources are mainly protected by the Linux file permissions. The protected resources include public and private web pages of course materials. The protection policies of different courses 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` An Iptable firewall deployed by a software company that serves millions of users. The policies include 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. The resource protection (mostly for private web pages) is done through the

| 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 |

**Table 4: Datasets used in our evaluation. The datasets cover a variety of systems, protection mechanisms, access control configurations at different scales (§10.1).**

Linux file permissions. The access policies are maintained by one sysadmin.

## 10.2 Experimental Design

In order to evaluate the effectiveness of P-DIFF, it requires two pieces of information: (1) access logs that records access requests and access results, and (2) access control policy changes that controlled the access results (the ground truth).

The `Wikipedia` dataset [62] includes both of the two pieces of information. The policy changes can be obtained based on `Wikipedia`'s page protection change history [36], 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 randomly generated policy changes and synthesized access results for requests recorded in the access logs (the original access results are ignored). If a generated change is "DENY to ALLOW", then the results for the related requests before the change are set to DENY and after the change are set to ALLOW. Vice versa, if a generated change is "ALLOW to DENY", then the results are set to ALLOW and DENY respectively. If a request has no related change generated, it is set to ALLOW by default. In this way, we also generated a random initial policy. Note another possible way to synthesize access results is using a learning algorithm to infer the initial policy from logs and applying the generated changes to the initial policy. However, in this way the derived initial policy is already easy to infer for a learning algorithm and so the synthesized policy is also biased to be easy to infer. To avoid the bias, we use a random initial policy instead of a learned initial policy.

As shown in Table 5, 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". This 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 5.

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.

| Category | Change Types (# Changes) | Dataset Applied |
|----------|--------------------------|-----------------|
| File permission | Type 1: allow file access (15) | Course |
| | Type 2: block file access (15) | Group |
| | 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) | |

Table 5: 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 §10.2.

## 10.3 Overall Results

*10.3.1* **Change Validation.** We evaluate P-DIFF's effectiveness in detecting policy changes. For each dataset, we divide the time span (cf. Table 4) 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 6 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 websites (Wikipedia ranked the 7th popular website in the world [2]).

*10.3.2* **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 7, P-DIFF pinpoints the root-cause policy change for 283 (93%) out of 303 accesses of interest. For `Wikipedia` and `Center` datasets, our inferred TCDT correctly encodes 114 out of 123 changed rules on normal objects (i.e. user and method). For the `Course`, `Company` and `Group` datasets, TCDT correctly encodes 169 out of 180 changed rules on hierarchical objects (i.e. directory

| 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) |

Table 6: Policy changes detected by P-DIFF. FP stands for false positive and FN stands for false negative.

| 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%) |

Table 7: Effectiveness of forensic analysis. P-DIFF can pinpoint the root-cause policy changes that permit the access of interest in the evaluation.

and subnet). Once a rule is correctly encoded, P-DIFF can always correctly backtrack the corresponding 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 §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 to pinpoint the precise root cause change.

*10.3.3* **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 6 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 7, 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 §2.
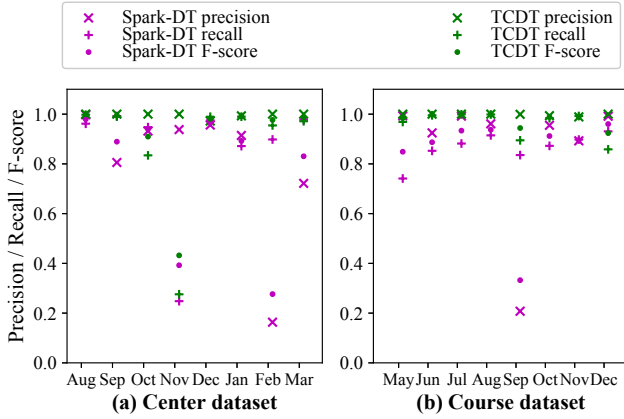
**Figure 11: Precision, recall, and F-score of TCDT classifying access results for the `Center` and `Course` datasets. The x-axis shows the time of the testing data, which is a month of logs in the dataset. The training data is the three continuous months of logs before the testing month.**

## 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 (TCDT) 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 [57] 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 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 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. §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 11 (a), after training TCDT with the new accesses, P-DIFF gets precision and recall both above 0.9.

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

**Table 8: Performance of P-DIFF: training time, and time for validation and forensics per access.**

## 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 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 (§ 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.

## 10.6 Execution Time

Table 8 shows the execution time of P-DIFF for training, validation, and forensics, respectively in previous validation and forensic experiments (cf. §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.

## 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 §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 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 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. §11).

The validation and forensics time for an access only takes a few milliseconds, as shown in Figure 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 12(b). Overall, the variance is small enough for efficient validation and forensics.

## 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 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 6). 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.
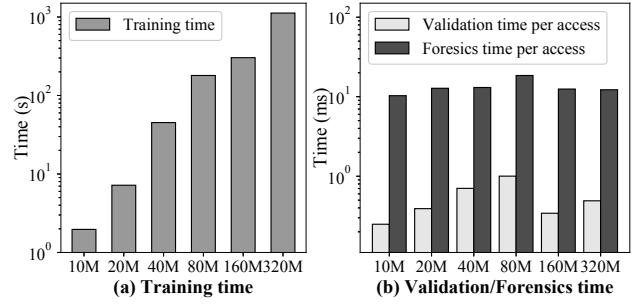


**Figure 12: Scalability analysis in terms of execution time for training, validation, and forensics with the increasing numbers of log entries from the `Wikipedia` dataset. Training time is linear to the number of log entries; it takes 19 minutes to train a model from 320 million log entries. Validation and forensics time are always low (1–10 milliseconds).**

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).

## 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 §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 §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 2,

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 [11, 68], 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.

## 12 RELATED WORK

### 12.1 Access Control Misconfigurations

Despite the extensive works in access control modeling and design, only a few efforts have been conducted in the past related to access-control misconfiguration detection. Most of the prior works attempt to detect access-control misconfiguration by finding inconsistencies between access-control policies [8, 9, 15, 54]. However, inconsistencies only reflect a very small, specific set of access-control misconfigurations. As acknowledged in these works, misconfigurations could totally be consistent, which often leads to even more severe consequences. In addition, these works require domain knowledge to interpreting specific access-control policies of different software. P-DIFF is complementary to the aforementioned works as we focus on access-control policy changes along with time instead of policy inconsistencies at a particular moment. Moreover, P-DIFF automatically infers access control policies from access logs without any domain knowledge or specification from sysadmins and thus is more general.

Testing and verification approaches [20, 34, 37] have been proposed to detect access control misconfigurations. While testing and verification have demonstrated promising results, they have not been widely deployed in practice due to the extensive efforts in writing testing cases or verification specifications. Especially, existing testing and verification approaches require a unified and centralized model (e.g., XACML); however, today's systems access control policies are kept in various forms including various configuration file formats, file permissions or database privilege table. It is hard to cover all the combinations of the access control configurations. Exactly due to this problem, P-DIFF chooses to infer access control policies from access logs regardless of configuration formats.

### 12.2 Other Types of Misconfigurations

To tackle misconfigurations, previous work has been done on detecting or troubleshooting system misconfigurations [5, 6, 49, 63, 65, 67, 75, 78, 79]. While those techniques are effective for detecting or troubleshooting misconfigurations that lead to system failures, they cannot deal with access control misconfigurations. Access control misconfigurations are fundamentally different from general software misconfigurations that lead to functional failures or performance degradation (which is assumed by the aforementioned techniques). Instead, they can go unnoticed for months until being exploited by malicious users. In addition, access control misconfigurations are typically "valid" configuration settings but do not conform to users' or organizational security policy.

### 12.3 Access Control Code Vulnerability

Besides misconfigurations, vulnerabilities can also be introduced by software bugs, e.g. the software could miss permission checks. Sun *et al.* [59] 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 [74] is a runtime system that enforces data-flow assertions to prevent exploits of web application security vulnerabilities. Nemesis [14] is a runtime system for preventing authentication and access control bypass attacks. SPACE [38] 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.

### 12.4 Intrusion Detection

One related research area on security-related log analysis is intrusion detection [18]. 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 [32]. 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.

### 12.5 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 [21]. 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 [26, 44, 58]. 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.

## 13 CONCLUSION

This paper presents P-DIFF, a practical tool for continuously monitoring access logs to help sysadmins detects 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 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 paper, it can be generally adopted to address the challenges in inferring other policies with result changes.

## ACKNOWLEDGMENTS

## REFERENCES

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. 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)*.

[2] Alexa Internet, Inc. 2019. Alexa traffic ranks. https://www.alexa.com/siteinfo/wikipedia.org.

[3] Hussein Almuallim, Yasuhiro Akiba, and Shigeo Kaneda. 1995. On Handling Tree-Structured Attributes in Decision Tree Learning. In *Proceedings of the 12th International Conference on Machine Learning (ICML'95)*.

[4] Amos Jeffries. 2015. Squid proxy access log format. https://wiki.squid-cache.org/Features/LogFormat.

[5] Mona Attariyan, Michael Chow, and Jason Flinn. 2012. X-ray: Automating Root-Cause Diagnosis of Performance Anomalies in Production Software. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*.

[6] Mona Attariyan and Jason Flinn. 2010. Automating Configuration Troubleshooting with Dynamic Information Flow Analysis. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*.

[7] Lujo Bauer, Lorrie Faith Cranor, Robert W. Reeder, Michael K. Reiter, and Kami Vaniea. 2009. Real Life Challenges in Access-control Management. In *Proceedings of the 2009 CHI Conference on Human Factors in Computing Systems*.

[8] Lujo Bauer, Scott Garriss, and Michael K Reiter. 2008. Detecting and resolving policy misconfigurations in access-control systems. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies (SACMAT'08)*.

[9] Lujo Bauer, Scott Garriss, and Michael K Reiter. 2011. Detecting and resolving policy misconfigurations in access-control systems. *ACM Transactions on Information and System Security (TISSEC)* 14, 1 (2011), 2.

[10] Leo Breiman, Jerome H. Friedman, Richard A. Olshen, and Charles J. Stone. 1983. *Classification and Regression Trees*. Wadsworth Publishing.

[11] Anton Chuvakin and Gunnar Peterson. 2010. How to Do Application Logging Right. *IEEE Security & Privacy* 8, 4 (July 2010), 82–85.

[12] Cloudera, Inc. 2019. Hadoop Audit Event. https://docs.cloudera.com/documentation/enterprise/5-4-x/topics/cn_iu_audits.html.

[13] Carlos Cotrini, Thilo Weghorn, and David Basin. 2018. Mining ABAC Rules from Sparse Logs. In *Proceedings of the 3rd European Symposium on Security and Privacy (EuroS&P'18)*.

[14] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. 2009. Nemesis: Preventing Authentication & Access Control Vulnerabilities in Web Applications. In *Proceedings of the 18th USENIX Security Symposium (USENIX Security'09)*.

[15] Tathagata Das, Ranjita Bhagwan, and Prasad Naldurg. 2010. Baaz: A System for Detecting Access Control Misconfigurations. In *Proceedings of the 19th USENIX Security Symposium (USENIX Security'10)*.

[16] Jessica Davis. Apr. 2018. 63,500 patient records breached by New York provider's misconfigured database. https://tinyurl.com/y88vh8u5.

[17] Jessica Davis. Mar. 2018. Long Island provider exposes data of 42,000 patients in misconfigured database. https://tinyurl.com/y7t5p99n.

[18] Dorothy E Denning. 1987. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering* 2 (1987), 222–232.

[19] BOB DIACHENKO. 2019. Document Management Company Left Credit Reports Online. https://securitydiscovery.com/document-management-company-leaks-data-online/.

[20] Kathi Fisler, Shriram Krishnamurthi, Leo A Meyerovich, and Michael Carl Tschantz. 2005. Verification and Change-Impact Analysis of Access-Control Policies. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*.

[21] João Gama, Indrė Žliobaitė, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. 2014. A Survey on Concept Drift Adaptation. *ACM Computing Surveys (CSUR)* 46, 4 (2014), 44.

[22] François Gauthier, Dominic Letarte, Thierry Lavoie, and Ettore Merlo. 2011. 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*.

[23] Simon Haykin. 1994. *Neural Networks: A Comprehensive Foundation*. Prentice Hall PTR.

[24] Hewlett Packard Enterprise. 2015. 2015. HP Cyber Risk Report 2015. http://www8.hp.com/h20195/v2/GetPDF.aspx/4AA5-0858ENN.pdf.

[25] David W Hosmer Jr, Stanley Lemeshow, and Rodney X Sturdivant. 2013. *Applied logistic regression*. Vol. 398. John Wiley & Sons.

[26] Geoff Hulten, Laurie Spencer, and Pedro Domingos. 2001. Mining Time-Changing Data Streams. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'01)*.

[27] Junbeom Hur and Dong Kun Noh. 2011. Attribute-based access control with efficient revocation in data outsourcing systems. *IEEE Transactions on Parallel and Distributed Systems* 22, 7 (2011), 1214–1221.

[28] Kromtech Security Center. 2017. Auto Tracking Company Leaks Hundreds of Thousands of Records Online. https://tinyurl.com/y8uvdy9j.

[29] Kromtech Security Center. 2018. FedEx Customer Records Exposed. https://mackeepersecurity.com/post/fedex-customer-records-exposed.

[30] Butler W Lampson. 1974. Protection. *ACM SIGOPS Operating Systems Review* 8, 1 (1974), 18–24.

[31] Butler W. Lampson. 2004. Computer Security in the Real World. *IEEE Computer* 37, 6 (June 2004), 37–46.

[32] Hung-Jen Liao, Chun-Hung Richard Lin, Ying-Chih Lin, and Kuang-Yuan Tung. 2013. Intrusion Detection System: A Comprehensive Review. *Journal of Network and Computer Applications* 36, 1 (2013), 16–24.

[33] Luke Irwin. 2019. 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/.

[34] Evan Martin and Tao Xie. 2007. Automated Test Generation for Access Control Policies via Change-Impact Analysis. In *Proceedings of the 3rd International Workshop on Software Engineering for Secure Systems*.

[35] MediaWiki. 2019. MediaWiki is a collaboration and documentation platform brought to you by a vibrant community. https://www.mediawiki.org/wiki/MediaWiki.

[36] MediaWiki. Apr. 2018. enwiki dump progress on 20180420. "https://dumps.wikimedia.org/enwiki/20180420/".

[37] Tejeddine Mouelhi, Franck Fleurey, Benoit Baudry, and Yves Le Traon. 2008. 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)*.

[38] Joseph P Near and Daniel Jackson. 2016. 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)*.

[39] Lily Hay Newman. 2017. The Scarily Common Screw-Up That Exposed 198 Millions Voter Record. https://www.wired.com/story/voter-records-exposed-database/.

[40] NumPy developers. 2018. NumPy. https://www.numpy.org.

[41] Oracle. 2019. MySQL audit log file formats. https://dev.mysql.com/doc/refman/8.0/en/audit-log-file-formats.html.

[42] Dan O'Sullivan. 2017. Cloud Leak: How A Verizon Partner Exposed Millions of Customer Accounts. https://www.upguard.com/breaches/verizon-cloud-leak.

[43] Pandas. 2018. pandas: Python Data Analysis Library. https://pandas.pydata.org/.

[44] Lena Pietruczuk, Piotr Duda, and Maciej Jaworski. 2013. Adaptation of decision trees for handling concept drift. In *International Conference on Artificial Intelligence and Soft Computing*. Springer, 459–473.

[45] pure-ftpd. 2017. pure-ftpd - Linux man page. https://linux.die.net/man/8/pure-ftpd.

[46] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung, and Han Ratul Mahajan. 1985. Trusted computer system evaluation criteria. In *National Computer Security Center*. Citeseer.

[47] J. Ross Quinlan. 1986. Induction of Decision Trees. *Machine Learning* 1, 1 (1986), 81–106.

[48] J. Ross Quinlan. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers, Inc.

[49] Ariel Rabkin and Randy Katz. 2011. Precomputing Possible Configuration Error Diagnosis. In *Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering (ASE'11)*.

[50] Charles M Rader. 1972. Discrete Convolutions via Mersenne Transrorms. *IEEE Trans. Comput.* 100, 12 (1972), 1269–1273.

[51] Ravi S Sandhu, Edward J Coyne, Hal L Feinstein, and Charles E Youman. 1996. Role-Based Access Control Models. *IEEE Computer* 29, 2 (1996), 38–47.

[52] Bruce Schneier. 2009. Real-World Access Control. https://www.schneier.com/blog/archives/2009/09/real-world_acce.html.

[53] SELinux. 2014. SELinux auditing events. https://selinuxproject.org/page/NB_AL.

[54] Riaz Ahmed Shaikh, Kamel Adi, and Luigi Logrippo. 2017. A Data Classification Method for Inconsistency and Incompleteness Detection in Access Control Policy

Sets. *International Journal of Information Security* 16, 1 (2017), 91–113.

[55] Sara Sinclair and Sean W. Smith. 2010. What's Wrong with Access Control in the Real World? *IEEE Security & Privacy* 8, 4 (July 2010), 74–77.

[56] Sara Sinclair, Sean W. Smith, Stephanie Trudeau, M. Eric Johnson, and Anthony Portera. 2007. 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.

[57] Spark. 2018. Spark MLlib. https://spark.apache.org/docs/latest/ml-guide.html.

[58] Kenneth O Stanley. 2003. *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.

[59] Fangqi Sun, Liang Xu, and Zhendong Su. 2011. Static Detection of Access Control Vulnerabilities in Web Applications. In *Proceedings of the 20th USENIX Security Symposium (USENIX Security'11)*.

[60] The Apache Software Foundation. 2019. Apache2 access log format. https://httpd.apache.org/docs/current/logs.html.

[61] The Open Web Application Security Project. 2017. Jan. 2018. OWASP Top 10 - 2017: The Ten Most Critical Web Application Security Risks. https://www.owasp.org/images/7/72/OWASP_Top_10-2017_%28en%29.pdf.pdf.

[62] Guido Urdaneta, Guillaume Pierre, and Maarten van Steen. 2009. Wikipedia Workload Analysis for Decentralized Hosting. *Elsevier Computer Networks* 53, 11 (July 2009), 1830–1845. http://www.globule.org/publi/WWADH_comnet2009.html.

[63] Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, and Yi-Min Wang. 2004. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*.

[64] Rui Wang, XiaoFeng Wang, Kehuan Zhang, and Zhuowei Li. 2008. Towards Automatic Reverse Engineering of Software Security Configurations. In *Proceedings of the 15th ACM Conference on Computer and Communications Security (CCS'08)*.

[65] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. 2003. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of the 17th Large Installation Systems Administration Conference (LISA'03)*.

[66] Wikipedia. 2018. One-hot. https://en.wikipedia.org/wiki/One-hot.

[67] Tianyin Xu, Xinxin Jin, Peng Huang, Yuanyuan Zhou, Shan Lu, Long Jin, and Shankar Pasupathy. 2016. Early Detection of Configuration Errors to Reduce Failure Damage. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*.

[68] Tianyin Xu, Han Min Naing, Le Lu, and Yuanyuan Zhou. 2017. 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)*.

[69] Tianyin Xu, Vineet Pandey, and Scott Klemmer. 2016. An HCI View of Configuration Problems. *arXiv:1601.01747* (Jan. 2016).

[70] Tianyin Xu, Jiaqi Zhang, Peng Huang, Jing Zheng, Tianwei Sheng, Ding Yuan, Yuanyuan Zhou, and Shankar Pasupathy. 2013. Do Not Blame Users for Misconfigurations. In *Proceedings of the 24th Symposium on Operating System Principles (SOSP'13)*.

[71] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys (CSUR)* 47, 4 (July 2015).

[72] Yuu Yamada, Einoshin Suzuki, Hideto Yokoi, and Katsuhiko Takabayashi. 2003. 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)*.

[73] Zuoning Yin, Xiao Ma, Jing Zheng, Yuanyuan Zhou, Lakshmi N. Bairavasundaram, and Shankar Pasupathy. 2011. 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)*.

[74] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2009. Improving Application Security with Data Flow Assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP'09)*.

[75] Chun Yuan, Ni Lao, Ji-Rong Wen, Jiwei Li, Zheng Zhang, Yi-Min Wang, and Wei-Ying Ma. 2006. Automated Known Problem Diagnosis with Event Traces. In *Proceedings of the 1st EuroSys Conference (EuroSys'06)*.

[76] ZDNet. 2019. 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/.

[77] Jun Zhang and Vasant Honavar. 2003. Learning Decision Tree Classifiers from Attribute Value Taxonomies and Partially Specified Data. In *Proceedings of the 20th International Conference on Machine Learning (ICML'03)*.

[78] Jiaqi Zhang, Lakshmi Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. 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)*.

[79] Sai Zhang and Michael D Ernst. 2013. Automated Diagnosis of Software Configuration Errors. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*.