Towards Accelerated Usage Control Based on Access Correlations

Richard Gay, Jinwei Hu, Heiko Mantel, and Johannes Schickel

Department of Computer Science, TU Darmstadt, Germany
lastname@mais.informatik.tu-darmstadt.de

Abstract. Low run-time overhead is crucial for the practicability of usage-control mechanisms. In this article, we propose an approach to accelerate usage control by exploiting access correlations. Our approach combines two main ingredients: firstly, a technique to compute decisions ahead of time and, secondly, a method to guide selection of usage events to pre-compute decisions for. For the first, we speculatively pre-compute decisions for usage events. For the second, we exploit access correlations to identify high acceleration potential. We implemented our approach and evaluated it in a case study of security policy enforcement in a distributed storage system. Our empirical results show that the speedup is substantial. More concretely, the speedup on average is up to 61.5%.

1 Introduction

Usage control [29] augments access control by protecting the access to resources as well as the subsequent use of the resources. For instance, usage control can ensure that a confidential document can only be accessed by authorized users and can also constrain the number of times the document is printed or propagated to other authorized users. Dynamic mechanisms are a popular approach to enforce usage control (e.g., [7,9,13,15,20]). Analysis of the system at run-time allows such approaches to precisely enforce usage control policies. However, by operating at the run-time of the system, dynamic mechanisms inevitably impose a performance overhead on the system.

Large performance overheads can easily deter the users of a system. How much overhead is acceptable in practice depends on the application domain. The question how much overhead is tolerable has been investigated, for instance, in the area of web services with the finding that delays of already a few hundred milliseconds can result in sales loss, reduced service use, and generally a competitive disadvantage [5, 12, 25, 30, 31].

A standard approach to reduce overhead of dynamic mechanisms is to reduce the number of program instructions that are instrumented to invoke the mechanism by static analysis (e.g., [1, 1, 3, 10, 11, 22, 27, 27]). In this article we follow an orthogonal approach: We exploit domain knowledge to accelerate enforcement during run-time. Thus, our approach can be employed in addition to existing approaches based on static analysis.

To appear in:
H. Lipmaa (Eds.): NordSec 2017, LNCS 10674, pp. 245-261, 2017.
© Springer International Publishing AG 2017
The final publication will be available at Springer via https://dx.doi.org/10.1007/978-3-319-70290-2_15

In this article, we propose SPEED^{AC}, an approach to **spe**culatively pr**e**-compute **d**ecisions based on **a**ccess **c**orrelations. Concretely, a usage-control mechanism following our approach computes and stores decisions for possible future usage events on the side and uses these pre-computed decisions when the events actually occur. Our approach exploits that a lookup of a decision can be more efficient than computing the decision. For example, in a distributed setting, computing a decision is expensive when it requires network communication [13, 20]. For selecting which decisions to pre-compute, SPEED^{AC} exploits access correlations on pieces of data such as files, database entries, and in-memory objects. By access correlations we refer to correlations between accesses to data in a program resulting from access patterns encoded in the program logic and from how the program is used. For instance, that an employee uses her company's storage service to access the agenda of a business meeting correlates with her accessing the meeting presentation.

We demonstrate our approach in a case study of a distributed storage system in which we employ usage control against conflicts of interest. For this scenario, we implemented our approach in a concrete usage-control mechanism and empirically evaluated the performance of our implementation based on the 6-hour MSN BEFS access trace by Microsoft [18]. Our evaluation showed speedup on average of up to 61.5% compared to not utilizing access correlations, which indicates that our approach is feasible and can significantly accelerate usage control.

In summary, the technical contributions of this article are:

– the SPEED^{AC} approach to accelerate usage control by speculatively pre-computing decisions, where selection of usage events is guided by access correlations
– an implementation of SPEED^{AC} against conflict of interest in distributed storage systems, and
– an empirical evaluation of the performance of our proposed mechanism based on a 6-hour access trace by Microsoft.

To our knowledge, our work is the first based on the idea of exploitation of *probabilistic* correlations *between* usage events to accelerate dynamic mechanisms. Our evaluation shows that the speedup can be substantial and hence that this is an interesting direction to counter the overhead caused by dynamic usage-control mechanisms. Our article constitutes a first step: Further research will enable a better understanding of the full design space for concrete SPEED^{AC}-based mechanisms and its potential to accelerate usage control.

2 The Problem

Performing usage control means ensuring that the usage events performed by a target program comply with given usage constraints. Dynamic approaches can utilize accurate usage histories for precisely enforcing usage constraints. This comes at the cost of run-time overhead that is perceivable by users of the system.

According to studies by Amazon, Bing, and Google Search [5, 12, 30], page load time increases of 100–200ms already have a negative impact on sales and

reference	experiments	overhead of	overhead
[15]	file copying; compilation	dummy policy	1.5–33ms
[15]	file copying; compilation	policy monitoring	4–1000ms
[19]	FTP and HTTP file transfers	data-flow tracking (best-case), dummy policy	106–2931ms
[19]	FTP and HTTP file transfers	data-flow tracking (worst-case), dummy policy	131–53353ms
[7]	data storage via custom test program	access control, trust and reputation management	6–400ms
[13]	FTP file transfers	local decision-making	1.9–3ms
[13]	FTP file transfers	cooperative decision-making	2.7–16.1ms

Table 1: Perceivable overhead caused by usage-control mechanisms

user experience. This suggests that the run-time overhead caused by a usage-control mechanism for a single page request should remain below 100ms for the mechanism to be acceptable in practice. A single page request, however, can trigger a cascade of requests. For example, we observed that loading a single page of Dropbox's web interface for browsing photos[1] triggered Firefox to send 75 requests to Dropbox. That is, in this example a usage-control mechanism may take at most 1.33ms per request to remain below 100ms in total.

We looked at several usage-control mechanisms whose perceivable overhead has been measured experimentally: a non-distributed [15] and a distributed [19] mechanism for usage control on data and copies of data; an access control mechanism for grid computing [7]; and a generic mechanism to enforce security policies in distributed systems, which has been used to enforce Chinese Wall policies in distributed systems [13]. Table 1 summarizes our observations. The mechanism by Harvan et al. [15] exhibits overheads between about 1.5ms and 1s, depending on the test case. The mechanism by Kelbert et al. [19] introduces overheads of at least 106ms for file transfers of size 100kB. The mechanism by Colombo et al. [7] exhibits overheads between 6ms and 400ms. CliSeAu, by Gay et al. [13], yields overheads between 1.9ms and 16.1ms, depending on the concrete experiment.

The overhead of the different approaches is low already. However, in the Dropbox example, even the lowest overheads – 1.5ms per access for a dummy policy and 1.9ms for a Chinese Wall policy – accumulate to a total overhead of 112.5ms for 75 requests. This raises the question how to further reduce overhead of dynamic mechanisms. The problem we therefore address in this article is how to further accelerate dynamic usage-control mechanisms.

3 Our Approach: SPEED$^{\mathrm{AC}}$

The SPEED$^{\mathrm{AC}}$ approach for accelerating usage control is to speculatively pre-compute decisions ("SPEED") on the side and to use access correlations ("AC")

[1] https://www.dropbox.com/photos

to determine the speculative aspect of the pre-computation. Through the pre-computation on the side, SPEED^{AC} enables a reduction of the overhead that one can perceive when using a program that is subject to a usage-control mechanism.

3.1 Speculative Pre-computation of Decisions

With SPEED^{AC}, the decisions made by a usage-control mechanism are not all computed when they are needed but are, to some extent, pre-computed. That is, the mechanism computes decisions for some usage events already before the events actually occur. Since it is typically not known in advance which usage events occur, decisions are pre-computed speculatively for usage events that the mechanism can suspect to occur. Such a decision is then computed as if the usage event was actually about to occur. Rather than being used directly, the decision is stored in memory or in a database and might not be used at all when the respective usage event never occurs. For brevity, in the following we refer to speculatively pre-computed decisions simply as pre-computed decisions.

The pre-computation of decisions with SPEED^{AC} is performed "on the side" by a usage-control mechanism. That is, the mechanism need not suspend the target program for pre-computing decisions but uses, e.g., a concurrent thread for the pre-computation. The mechanism triggers the pre-computation of decisions after it has handled a concrete usage event. In particular, it allows the pre-computation to take the decision for this newest usage event into account.

Concretely, the decision-making with SPEED^{AC} integrates as follows into how the usage-control mechanism processes usage events. When the mechanism intercepts a usage event that the running target program is about to perform, the mechanism first performs a lookup for a pre-computed decision. In case of success, i.e., if a pre-computed decision for the intercepted event is available, this decision is enforced. For instance, if the decision is to permit the usage event, then the mechanism allows the program to perform the event. If the decision is to prevent the usage event, then the mechanism can, e.g., return an error code to the target such that it can afterwards resume its execution. When the lookup fails, i.e., when no pre-computed decision is available, then the mechanism computes a decision on the spot and enforces this decision. Unless a decision demanded to terminate the target program, the mechanism resumes the target after enforcing the decision and simultaneously triggers the pre-computation of decisions.

Figures 1 and 2 illustrate the cases of lookup success and, respectively, lookup failure. In the figures, time flows from left to right and shaded boxes represent functionality that is performed during the time. Notably, the target is blocked while the mechanism has intercepted an event and has not yet enforced a decision for the event. The target is not blocked while decisions are pre-computed.

We consider an attacker model of a malicious user of the target program. Concretely, the attacker can interact with the interface exposed by the program, such as a graphical user interface or a web interface, for trying to circumvent usage control. The attacker cannot directly observe or modify the mechanism. For the given attacker, neither soundness nor precision need to be sacrificed for increased performance when exploiting access correlations as well as parallelism

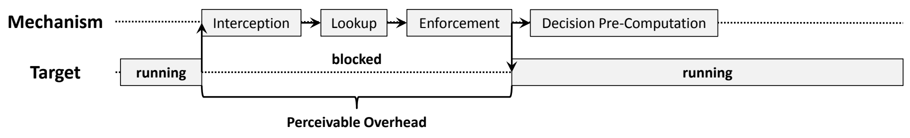


Fig. 1: Enforcement in case of lookup success

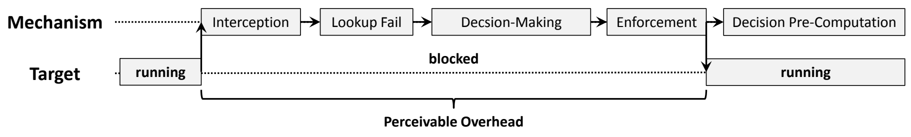


Fig. 2: Enforcement in case of lookup failure

for speculatively pre-computing decisions on the side. Preserving soundness and precision requires careful design and implementation of concrete mechanisms that use SPEED^{AC}. For instance, pre-computed decisions should not be utilized by a mechanism when they have been rendered obsolete by subsequent events.

3.2 Utilization of Access Correlations

Pre-computing decisions for all possible usage events is infeasible with regard to storage and computation time due to the generally vast number of such events. For selecting a limited set of decisions to pre-compute and yet achieving a high rate of successful lookups, we propose to use access correlations.

Access correlations on data result from access patterns encoded in a target program and from usage patterns established by the program's users. They capture which accesses to pieces of data are stochastically dependent. In this article, we focus on positive correlations, i.e., on cases in which the likelihood of accesses to two pieces of data occurring together during the run-time of the program is *higher* than the likelihood would be in case of stochastic independence. When the correlation between accesses to two pieces of data is sufficiently strong, we call the pieces of data correlated. An example of access correlations are correlations between accesses to disk blocks and files. Outside the domain of usage control, exploiting such correlations has already been proposed for accelerating file accesses through improved caching and prefetching (e.g., [16, 23]). Access correlation between files have been successfully calculated, e.g., by treating the metadata of files as a multi-dimensional attribute space and marking files in close proximity as correlated [16].

SPEED^{AC} proposes that a mechanism utilizes access correlations as follows. Suppose a decision for a usage event that accesses a piece of data d has just been enforced. Then the mechanism selects, using some selection strategy, a subset D of all pieces of data correlated to d and pre-computes decisions for usage events on D. Later during the run-time, the mechanism employs some termination strategy to remove obsoleted pre-computed decisions again.

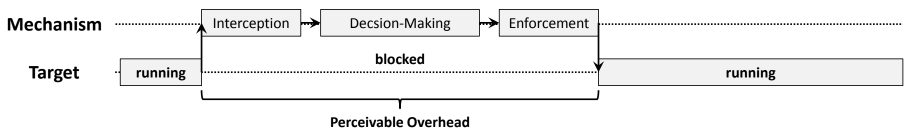


Fig. 3: Enforcement in current approaches

When access correlations are considered sufficiently strong as well as the selection strategy and termination strategy are scenario-specific and to be specified by concrete instances of SPEED^{AC}. When the chosen access correlations and strategies capture the actual program and user behavior well, the success rate of lookups is high and pre-computed decisions can be used often.

3.3 Perceivable Overhead

The perceivable overhead of a usage-control mechanism on a program is the additional delay, caused by the mechanism, that a user of the program experiences in her interaction with the program. The user might experience this delay, e.g., between a mouse click and the response by the GUI program or between a browser request to a web service and the resulting page being displayed.

Figures 1 and 2 depict the perceivable overhead caused with SPEED^{AC} for a single usage event. The overhead for a successful lookup includes the time for intercepting the event, for looking up a decision, and for enforcing a decision. The overhead for a failed lookup additionally includes the decision-making. The pre-computation of decisions is not part of the perceivable overhead, as it is performed while the target is running rather than while the target is blocked.

Traditional usage-control mechanisms (e.g., [1, 11, 13, 20, 27]) handle usage events as shown in Figure 3: The mechanisms block the target program for interception, decision-making, and enforcement. They do not pre-compute decisions and perform their functionality sequentially to the target.

The perceived overhead in traditional mechanisms clearly is smaller than the perceived overhead caused by failed lookups in SPEED^{AC}: The latter comprises all tasks of the former and additionally includes the lookup. How the case of a lookup success compares to the traditional approach boils down to how the successful lookup compares to the decision-making. Decision-making can be significantly more time-consuming than a lookup in scenarios with complex usage constraints or in distributed systems, in which decision-making involves network communication. SPEED^{AC} reduces the perceived overhead if the time saved through successful lookups outweighs the overhead of failed lookups on average. We elaborate an example of such a setting in the remainder of this article.

4 Case Study

We use a case study to demonstrate how SPEED^{AC} can be realized in a concrete application scenario and how much acceleration can be achieved. In the application

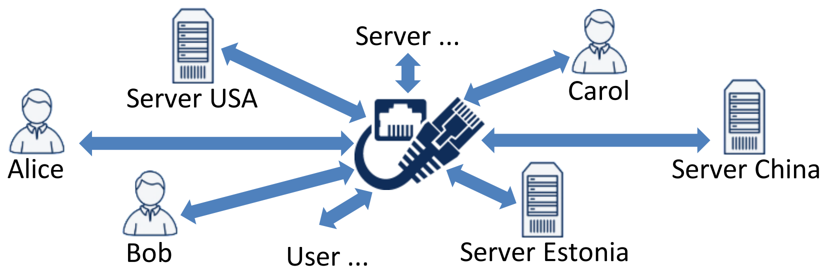


Fig. 4: Distributed storage service

scenario, a distributed storage service offers storage space to its users. The service consists of multiple, spatially distributed servers through which the users can access the storage. Figure 4 depicts the possible interactions between users and the service. Through a network such as the Internet, each user can connect to any of the servers for storing and retrieving files from the service.

The usage of the service shall be constrained according to a Chinese Wall policy [4] in order to technically counter conflicts of interest. That is, from a class of competing companies, each user may only access the files owned by one company. A mechanism can enforce this usage constraint by controlling the read and write events performed on behalf of users. We chose the Chinese Wall policy as an example of a business security requirement for which the computation of decisions in a distributed setting is non-trivial in general [26].

In the remainder of this article, we call two files *conflicting* if they are owned by competing companies. We lift this notion to usage events (i.e., read and write events) by calling two usage events conflicting when the files accessed by the usage events are conflicting. The notion establishes an irreflexive and symmetric binary relation on usage events. By *equivalence classes* on usage events, we refer to the equivalence classes of the reflexive transitive closure of this relation.

5 Enforcement Mechanism

We design and implement a mechanism following SPEED$^{\mathrm{AC}}$ for the setting in Section 4. The core challenge is to design a mechanism that is effective *and* performant. Effectiveness demands *soundness*, i.e., that the mechanism assures the absence of policy violations, and *transparency*, i.e., that the mechanism permits all accesses that do not violate the policy [24].

Our mechanism is built on Gay, Hu, and Mantel's mechanism in [13]. To monitor and to intervene with a target program's execution, the mechanism encapsulates the target program into *enforcement capsules*, which we refer to as *nodes* in our setting. We re-use interception and enforcement, but design our own decision-making algorithm. The original decision-making algorithm works as follows: The mechanism maps usage events to nodes that are responsible for deciding on them. Effectiveness is achieved through the following requirement:

Whenever two usage events conflict, the same node is responsible to decide on the events. We call this property on the responsibility distribution *properness.*

We first explore the design space for the decision-making algorithm. Second, we describe the proposed decision-making algorithm and the underlying design decisions. Finally, an overview over key implementation details is presented.

5.1 Design Space

In the design space for applying SPEED^{AC} to a mechanism for our application scenario, we identify three particular dimensions: the *selection* of usage events for the pre-computation, the *location* at which pre-computed decisions are stored, and the *lifetime* of pre-computed decisions. For each dimension we discuss its impact on soundness and performance and provide points in the design space.

In our case study, pre-computation does not affect transparency: As time advances and users access files, the set of accesses that comply with the Chinese Wall policy shrinks monotonically. Pre-computed decisions from earlier points in time, thus, do not violate transparency when they are enforced.

Selection. According to SPEED^{AC}, a mechanism can select the usage events for the pre-computation from the set of usage events that are correlated to the previously intercepted usage event. A greedy strategy is to select all correlated usage events. More cautious strategies are, e.g., to select at most a single correlated usage event and ensure that for each user only one pre-computed decision exists or to select a maximal set of correlated usage events such that for each user and equivalence class only one pre-computed decision exists.

The selection strategy can affect the soundness and performance of a mechanism. For instance, the greedy strategy might select two permissible but conflicting usage events for the pre-computation. If the mechanism would enforce the decisions for both events, it would violate the Chinese Wall policy. The more cautious strategies do not exhibit this property, as they prevent conflicting pre-computed decisions. Concerning performance, the greedy strategy yields a higher chance of successful lookups than the cautious strategies. However, it also increases the lookup and maintenance costs for pre-computed decisions.

Location. The location, i.e., the node at which a pre-computed decision is stored, is a dimension opened up by the distributed setting. A strategy to select the location can be static or dynamic. A static strategy does not adapt to system behavior but always uses the same node for each pre-computed decision. For example, the strategy could fix a node for each usage event based on the file location. A dynamic strategy selects the node based on observed system behavior. For example, the strategy could track where a usage event occurs most often and store the associated pre-computed decision there.

The locations of pre-computed decisions affect the mechanism's performance. A pre-computed decision can only be looked up efficiently, when it is stored at the node intercepting the respective usage event. Otherwise, comparatively expensive network communication is required. A dynamic strategy can increase efficient lookup chances but at the cost of additional bookkeeping. When some nodes are only temporarily reachable via the network, using these nodes to store pre-computed decisions also affects the soundness of the mechanism.

Lifetime. The lifetime of pre-computed decisions can be controlled by termination strategies. A termination strategy can be to terminate certain pre-computed decisions during the lookup, after the enforcement of a pre-computed decisions, during on-the-spot decision-making, and/or during the selection of events for pre-computation. The strategy can be to terminate pre-computed decisions that would conflict with newly selected usage events for the pre-computation. Conversely, the termination strategy can also be to keep once pre-computed decisions and rather select fewer events in the next pre-computation step.

The termination strategy can affect the soundness and performance of the mechanism. For instance, if conflicting pre-computed decisions are not terminated when a permitting decision is computed on the spot, then subsequently utilizing a pre-computed decision might violate the Chinese Wall policy. The performance impact of a termination strategy is in two directions. Firstly, terminating a pre-computed decision that is not located at the node that triggers the termination requires network communication and is therefore expensive. Secondly, terminating more or fewer decisions has the same impact on the performance as choosing fewer or, respectively, more events for the pre-computation.

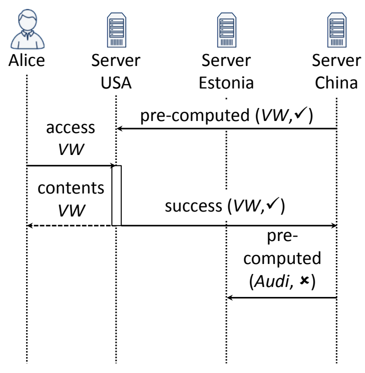
5.2 Design for Effectiveness

In our mechanism responsible nodes compute and pre-compute decisions. Each node memorizes its permitted usage events to compute decisions in compliance with the Chinese Wall policy. A node decides to prevent a usage event only when a conflicting usage event was permitted previously. On enforcement of a permit decision the responsible node memorizes the permission of the usage event.

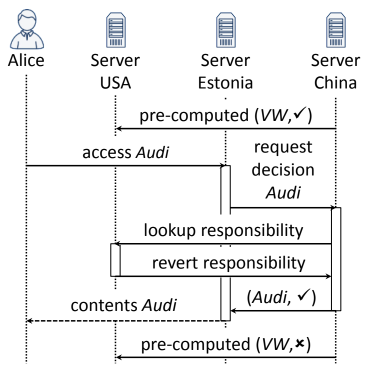
We let pre-computed decisions induce *temporary responsibility shifts*: a node storing the pre-computed decision becomes temporarily responsible for the equivalence class of the underlying usage event. Conversely, when the pre-computed decision is terminated, the responsibility shifts back to the originally responsible node. As a result, the responsibility distribution is proper also in presence of temporary responsibility shifts. A temporarily responsible node can only lookup the pre-computed decision, it can not compute decisions by itself.

Our mechanism employs a cautious selection strategy: it selects a maximal set of correlated usage events such that for each user and equivalence class only one pre-computed decision exists. The employed termination strategy is twofold: it terminates pre-computed decisions after their enforcement and during on-the-spot decision-making. We fix a static strategy to select the location of pre-computed decisions: for each usage event a node is fixed based on the file location.

Augmenting the mechanism, i.e., [13] with SPEED^{AC} based on the design choices we made preserves the effectiveness of the mechanism. Our mechanism's on-the-spot decision-making preserves effectiveness due the responsibility distribution being proper even in presence of temporary responsibility shifts. As previously



(a) Success Case



(b) Fail Case

Fig. 5: Decision-making examples

discussed, pre-computed decisions do not affect transparency of our mechanism. Enforcing a pre-computed decision u would only break soundness when u permits a usage event and a decision v permits a conflicting usage event. We distinguish the two cases that could lead to a policy violation. (1) between computation of u and its enforcement an on-the-spot decision v is enforced. However, the termination strategy prevents this by terminating u during computation of v. (2) between computation of u and its enforcement a pre-computed decision v is enforced. The selection strategy prevents this: if v is computed before u, u is never pre-computed. If u is computed before v, v is never pre-computed. Thus, only either u or v is enforced. Absence of both (1) and (2) preserves soundness.

Example. Consider an audit process where *Alice* is hired to audit car manufacturing companies. The companies' data is distributed over servers *USA*, *Estonia*, and *China*. To avoid conflict-of-interest *Alice* is only given access to a single company's data. Server *China* is responsible to decide on usage events. Files of the companies *VW* and *Audi* are correlated in both directions.

A successful lookup is given in Figure 5a. Initially, a pre-computed decision for access to *VW* arrives at node *USA*. An access to *VW* on server *USA* is performed by *Alice*. *USA* does a lookup of the pre-computed decision, terminates the decision, and reverts the responsibility shift through a notification to *China*. The notification causes *China* to update the set of permitted events. Since *VW* and *Audi* are correlated, a decision for access to *Audi* is pre-computed.

A failed lookup is given in Figure 5b. After the pre-computed decision for *VW* arrives at *USA*, Alice performs an access to *Audi* on *Estonia*. A decision for access to *Audi* is computed on-the-spot. The mechanism terminates the pre-computed decision for *VW* and reverts the responsibility shift. Correlation between *Audi* and *VW* causes a pre-computation of a decision for access to *VW*.

A key property of our design is that a lookup of a pre-computed decision requires no coordination among the nodes, as indicated also in our example.

The coordination was done on the side during the pre-computation. That is, no network communication takes place for a usage event in case of a lookup success. We therefore expect that our mechanism exhibits a lower perceivable overhead compared to traditional approaches, which perform all coordination on the spot.

5.3 Implementation

We prototypically implemented our mechanism using the CliSeAu tool [13]. The implementation of the decision-making algorithm consists of 757 source lines of Java code (SLOC; empty/comment lines excluded) in 13 classes. This implementation is generic with respect to the target program. For a concrete target program to establish the distributed storage service, we selected CrossFTPServer.[2] The target-specific implementation consists of 81 SLOC in 3 classes.

We complement SPEED$^{\text{AC}}$ for efficiency at the design level by efficient data structures at the implementation level that assure efficient lookup and maintenance of pre-computed decisions. This includes: pre-computed decision lookup, temporary responsibility shifts, and our usage event selection strategy. We realized all utilizing hash maps, which feature average $O(1)$ running time for all operations [8, p. 253]. By using representatives of equivalence classes as keys, we could efficiently implement lookup, deletion, and responsibility shifting.

To assure effectiveness of our mechanism we implemented JUnit tests and on top applied systematic manual testing. The JUnit tests cover functionality of decision-making, responsibility shifting, and bookkeeping for event selection. We complemented the tests with systematically testing an instantiation of our mechanism for CrossFTPServer. In a system setup with a concrete a policy, we manually accessed files to test the soundness and transparency of our implementation. In all cases we found our mechanism to be sound and to be transparent.

6 Performance Evaluation

We experimentally evaluate the performance of our mechanism by investigating its perceivable overhead. Through the evaluation, we assess whether and to which extent SPEED$^{\text{AC}}$ reduces perceivable overhead compared to a traditional usage-control mechanism in our case study. As the reference point for the comparison, we employ the mechanism by Gay, Hu, and Mantel [13], which we call SOA (abbreviating State-Of-the-Art) in the following.

6.1 Experimental Setup

For our experimental evaluation we employ a distributed file-storage system, a concrete instance of the system setting in Section 4. The system consists of 8 servers hosting a file structure modeled after the MSN BEFS trace [18], which captures the operation of a file server of Microsoft's Live services. We replay a post-processed MSN BEFS trace to simulate a system execution.

[2] http://www.crossftp.com/crossftpserver.htm

type	**0.8**	**0.8_d**	**0.8_g**	**0.5**	**0.2**	**∅**
SOA	2.62ms	2.62ms	2.58ms	2.02ms	1.49ms	2.27ms
SPEED^{AC}	1.01ms	1.48ms	1.83ms	0.99ms	0.96ms	1.25ms
abs. speedup	1.61ms	1.16ms	0.75ms	1.03ms	0.53ms	1.02ms
rel. speedup	61.5%	44.3%	29.1%	51.0%	35.6%	44.9%

Table 2: Perceivable overhead

The MSN BEFS trace is a block I/O trace of a Microsoft's Live services server containing 8 physical disks. The trace captures operation during 6 hours. For our experiments, a file access event represents an aggregation of block accesses in close succession. We place the files for each disk on a separate server.

Our experiments use multiple synthesized Chinese Wall policies, i.e., 0.2, 0.5, 0.8, 0.8_d, 0.8_g. For each policy we target a rate of cooperative decision-making in SOA for a trace replay. Conflicting files distributed over nodes require SOA to cooperatively compute decisions. Our synthesis randomly selects files from different nodes and marks them as conflicting until we reach the targeted rate. The rate is represented by the name, e.g., 0.2 targets a rate of 20% in SOA. For policy 0.8_g an equivalence class contains files from at most seven nodes, for 0.8_d from at most four, and for the remaining ones from at most two.

Our experiments use a synthesized file-correlation that predicts, for each file access, the following file access in 80% of cases. Our synthesis follows the process: For each file f, the most frequent access following f is marked correlated until the hit-rate reaches 80%.[3]

In our experiments we employ 8 Lenovo ThinkCentre M93p as servers. Each is equipped with an Intel(R) Core(TM) i5-4590 CPU, 32 GB of RAM, and an 1000Mbit/s Intel I217-LM Ethernet adapter. As operating system Ubuntu Linux 14.04.2 LTS is run. The FTP server we employ is CrossFTPServer version 1.11. The JavaVM is OpenJDK version 7u79-2.5.5-0ubuntu0.14.04.2.

We conduct experiments for each Chinese Wall policies for both SOA and SPEED^{AC}. An experiment consists of 5 independent trace replays, i.e., we start fresh instances of all software. A trace replay measures the response time for 5552150 file accesses by 256 distinct users. We average the obtained results.

6.2 Perceivable Overhead

Our system exhibits an average response time of 2.03ms without usage control enforcement. The perceivable overhead of SPEED^{AC} and SOA is obtained by subtracting 2.03ms from their response times. Table 2 presents our results.

For SOA perceivable overhead is between 1.49ms (for 0.2) and 2.62ms (for 0.8 and 0.8_d) with an average of 2.27ms. The perceivable overhead for SPEED^{AC}

[3] The seemingly high hit-rate of 80% in fact constitutes a conservative choice: Hua et al. [16] obtained a 95.2% hit-rate on the same trace data.

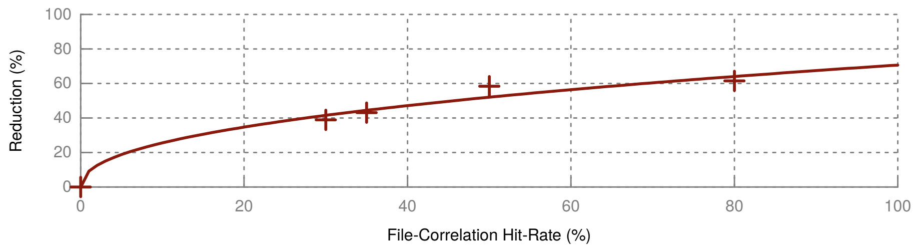


Fig. 6: Effects of file-correlation on perceivable overhead.

is between 0.96ms (for 0.2) and 1.83 (for 0.8_g) with an average of 1.25ms. The reduction is between 29.1% (for 0.8_g) and 61.5% (for 0.8), averaging at 44.9%.

We particularly find the variability in the perceivable overhead among experiments 0.8, 0.8_d, and 0.8_g with SPEED$^{\mathrm{AC}}$ very interesting, given that these experiments feature nearly the same perceivable overhead with SOA. We identified the *size* of an equivalence class, i.e., its number of usage events, as cause. Our selection and termination strategies allow for fewer pre-computed decisions with increased size of equivalence classes. Further investigation showed that, indeed, fewer pre-compute decisions are enforced during 0.8_d and even fewer during 0.8_g.

Our results show a reduced perceivable overhead for SPEED$^{\mathrm{AC}}$ in all cases. On average SPEED$^{\mathrm{AC}}$ reduces perceivable overhead by 44.9% compared to SOA, i.e., the average perceivable overhead is reduced from 2.27ms to 1.25ms. We find this encouraging to depoly SPEED$^{\mathrm{AC}}$ for efficient usage control enforcement.

6.3 File-Correlation Effects

Our results made us curious about the effects of different file-correlation hit-rates on perceivable overhead. We conducted additional experiments to identify the relation between hit-rate and reduction of perceivable overhead.

A single run was conducted for 3 additionally synthesized file-correlations with hit-rates 50%, 35%, and 30%. As lowest hit-rate 30% captures only correlating the most frequent successive file access for each file. Our experiments use policy 0.8 due to high reductions for SPEED$^{\mathrm{AC}}$ in our previous experiments.

Figure 6 shows the results of our experiments. For 80% the reduction is taken from our previous experiment. Hit-rates 50%, 35%, and 30% show a reduction of 58%, 43%, and respectively 38%. Between hit-rates 50% and 80% the perceivable overhead reduction only differs by 3%. A curve, fitted on the obtained results, allows to anticipate reduction rates for hit-rates beyond the ones we employed.

To our surprise hit-rates above 50% only cause marginal additional reduction of perceivable overhead. On the other side of the spectrum, even hit-rates low as 30% result in a significant reduction of 38%. Thus, our results show SPEED$^{\mathrm{AC}}$ is useful even in settings with limited knowledge of access correlations.

7 Related Work

Optimizations for enforcement mechanisms, including usage-control mechanisms, are common and have been pursued in several directions. Mechanisms that utilize the inlining technique [10], e.g., based on aspect-oriented programming [21], use static program analysis to reduce the number of program instructions that are instrumented to invoke the mechanism (e.g., [1, 13, 27]). SASI [11] and Clara [3] expand the analysis to sequences of instructions in order to further reduce the number of invocations of the mechanism. Automata-theoretic techniques have been proposed for minimizing the composition of a program and an enforcement mechanism [6, 22]. The optimizations performed by these approaches can be viewed as a form of statically pre-computed decisions ("permit") for security-irrelevant events.

Techniques for optimizing the performance of enforcement mechanisms themselves have also been proposed. JavaMOP [27] employs an optimization for enforcing properties on individual Java objects based on a decentralized indexing scheme that accelerates the lookup of mechanism state. JavaMOP furthermore optimizes the number of monitor state updates by exploiting the structure of the enforced property to achieve a low number of monitors. A dynamic optimization of an enforcement mechanism is proposed by the RV system [17], which at run-time collects dead monitors to reduce bookkeeping overhead.

Particularly for distributed enforcement mechanisms, architecture-based optimizations have been proposed. Gay et al. [13] and, subsequently, Decat et al. [9] propose to use a decentralized coordination among the distributed components of the mechanism for efficiently and effectively enforcing given properties. Kelbert et al. [20] employ a general-purpose distributed database for an efficient coordination. Our approach, i.e., optimizing via access correlations, is orthogonal to the related works presented in this section. That is, existing optimization techniques based on static analysis, individual decision-making, and distributed architectures can be utilized in addition to our optimization.

Application scenarios similar to the one in our case study have been subject of enforcement mechanisms before [9, 13, 14, 26]. We use the scenario because, firstly, the Chinese Wall policy [4] is a classic business requirement and, secondly, we could use a publicly available mechanism as a reference for our evaluation [13]. Note, however, that SPEED$^{\mathrm{AC}}$ is a generally applicable approach for accelerating usage control. How special-purpose mechanisms, e.g., for DRM [28] or distributed access control [2], could be optimized is beyond the scope of the article.

8 Conclusion

We proposed SPEED$^{\mathrm{AC}}$, an approach for accelerating distributed usage control enforcement by speculatively pre-computing decisions for usage events based on access correlations. In our case study, we developed a usage-control mechanism with SPEED$^{\mathrm{AC}}$ against conflicts of interest in distributed storage systems. The performance evaluation based on a real world data trace from Microsoft's Live services provides first evidence that our approach has the potential to significantly accelerate usage control. Concretely, our mechanism exhibited perceivable overheads that are up to 61.5% lower on average compared to not utilizing SPEED^{AC}. In absolute terms, the acceleration allowed us to reduce the perceivable overhead from 2.27ms to 1.25ms on average (see Table 2).

Our work constitutes a first step in this promising direction. Further investigation in this direction will provide a better understanding of the full potential of our approach. Questions for further investigations are: How can the approach be exploited to accelerate usage control even further and in other application scenarios, e.g., involving also dynamic policies? Does SPEED^{AC} influence how much information an attacker capable of measuring perceivable overhead can learn about processed secrets?

Acknowledgments We thank the anonymous reviewers for their constructive comments. This work was partially funded by CASED (www.cased.de) and by the DFG as part of the project Secure Refinement of Cryptographic Algorithms (E3) within the CRC 1119 CROSSING.

References

1. Bauer, L., Ligatti, J., Walker, D.: Composing Expressive Runtime Security Policies. TOSEM 18(3), 9:1–9:43 (2009)
2. Becker, M.Y., Sewell, P.: Cassandra: Distributed Access Control Policies with Tunable Expressiveness. In: POLICY. pp. 159–168. IEEE Computer Society (2004)
3. Bodden, E., Hendren, L.: The Clara Framework for Hybrid Typestate Analysis. STTT 14(3), 307–326 (2012)
4. Brewer, D.F., Nash, M.J.: The Chinese Wall Security Policy. In: IEEE S&P. pp. 206–214 (1989)
5. Brutlag, J.: Speed Matters for Google Web Search. https://services.google.com/fh/files/blogs/google_delayexp.pdf (2009), accessed 2017-07-16
6. Colcombet, T., Fradet, P.: Enforcing Trace Properties by Program Transformation. pp. 54–66. ACM (2000)
7. Colombo, M., Martinelli, F., Mori, P., Petrocchi, M., Vaccarelli, A.: Fine Grained Access Control with Trust and Reputation Management for Globus. In: OTM. pp. 1505–1515. LNCS 4804, Springer (2007)
8. Cormen, T.H., Leierson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009)
9. Decat, M., Lagaisse, B., Joosen, W.: Scalable and Secure Concurrent Evaluation of History-based Access Control Policies. In: ACSAC. pp. 281–290. ACM (2015)
10. Erlingsson, U.: The Inlined Reference Monitor Approach to Security Policy Enforcement. Ph.D. thesis, Cornell University (2004)
11. Erlingsson, Ú., Schneider, F.B.: SASI Enforcement of Security Policies: A Retrospective. In: NSPW. pp. 87–95. ACM (1999)
12. Forrest, B.: Bing and Google Agree: Slow Pages Lose Users. http://radar.oreilly.com/2009/06/bing-and-google-agree-slow-pag.html (2009), accessed 2016-07-16
13. Gay, R., Hu, J., Mantel, H.: CliSeAu: Securing Distributed Java Programs by Cooperative Dynamic Enforcement. In: ICISS. pp. 378–398. LNCS 8880, Springer (2014)

14. Gay, R., Mantel, H., Sprick, B.: Service Automata. In: FAST. pp. 148–163. LNCS 7140, Springer (2012)
15. Harvan, M., Pretschner, A.: State-Based Usage Control Enforcement with Data Flow Tracking Using System Call Interposition. In: NSS. pp. 373–380. IEEE Computer Society (2009)
16. Hua, Y., Jiang, H., Zhu, Y., Feng, D., Xu, L.: SANE: Semantic-Aware Namespace in Ultra-Large-Scale File Systems. TPDS 25(5), 1328–1338 (2014)
17. Jin, D., Meredith, P.O., Griffith, D., Rosu, G.: Garbage Collection for Monitoring Parametric Properties. In: PLDI. pp. 415–424. ACM (2011)
18. Kavalanekar, S., Worthington, B.L., Zhang, Q., Sharda, V.: Characterization of Storage Workload Traces from Production Windows Servers. In: IISWC. pp. 119–128 (2008)
19. Kelbert, F., Pretschner, A.: Data Usage Control Enforcement in Distributed Systems. In: CODASPY. pp. 71–82. ACM (2013)
20. Kelbert, F., Pretschner, A.: A Fully Decentralized Data Usage Control Enforcement Infrastructure. In: ACNS. pp. 409–430. LNCS 9092, Springer (2015)
21. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: ECOOP. pp. 220–242. Springer-Verlag (1997)
22. Lemay, F., Khoury, R., Tawbi, N.: Optimized Inlining of Runtime Monitors. In: NordSec. pp. 149–161. LNCS 7161, Springer (2012)
23. Li, Z., Chen, Z., Srinivasan, S.M., Zhou, Y.: C-Miner: Mining Block Correlations in Storage Systems. In: FAST. pp. 173–186. USENIX (2004)
24. Ligatti, J., Bauer, L., Walker, D.: Edit Automata: Enforcement Mechanisms for Run-time Security Policies. International Journal of Information Security 4(1–2), 2–16 (2005)
25. Lohr, S.: Bing and Google Agree: Slow Pages Lose Users. http://www.nytimes.com/2012/03/01/technology/impatient-web-users-flee-slow-loading-sites.html (2012), accessed 2017-07-16
26. Martinelli, F., Matteucci, I.: Synthesis of Local Controller Programs for Enforcing Global Security Properties. In: ARES. pp. 1120–1127. IEEE Computer Society (2008)
27. Meredith, P.O., Jin, D., Griffith, D., Chen, F., Roşu, G.: An Overview of the MOP Runtime Verification Framework. STTT 14(3), 249–289 (2012)
28. Ongtang, M., Butler, K.R.B., McDaniel, P.D.: Porscha: Policy Oriented Secure Content Handling in Android. In: Gates, C., Franz, M., McDermott, J.P. (eds.) ACSAC. pp. 221–230. ACM (2010)
29. Park, J., Sandhu, R.S.: The UCON_{ABC} Usage Control Model. TISSEC 7(1), 128–174 (2004)
30. Shalom, N.: Amazon found every 100ms of latency cost them 1% in sales. https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/ (2008), accessed 2017-07-16
31. Singla, A., Chandrasekaran, B., Godfrey, B., Maggs, B.M.: The Internet at the Speed of Light. In: HotNets. pp. 1:1–1:7. ACM (2014)