Thoth: Comprehensive Policy Compliance in Data Retrieval Systems

Eslam Elnikety, Aastha Mehta, Anjo Vahldiek-Oberwagner, Deepak Garg, and Peter Druschel, ***Max Planck Institute for Software Systems (MPI-SWS)***

https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/elnikety

This paper is included in the Proceedings of the 25th USENIX Security Symposium

August 10–12, 2016 • Austin, TX

ISBN 978-1-931971-32-4

Open access to the Proceedings of the 25th USENIX Security Symposium is sponsored by USENIX

Thoth: Comprehensive Policy Compliance in Data Retrieval Systems

Eslam Elnikety Aastha Mehta Anjo Vahldiek-Oberwagner Deepak Garg
Peter Druschel

Max Planck Institute for Software Systems (MPI-SWS)

Abstract

Data retrieval systems process data from many sources, each subject to its own data use policy. Ensuring compliance with these policies despite bugs, misconfiguration, or operator error in a large, complex, and fast evolving system is a major challenge. Thoth provides an efficient, kernel-level compliance layer for data use policies. Declarative policies are attached to the systems' input and output files, key-value tuples, and network connections, and specify the data's integrity and confidentiality requirements. Thoth tracks the flow of data through the system, and enforces policy regardless of bugs, misconfigurations, compromises in application code, or actions by unprivileged operators. Thoth requires minimal changes to an existing system and has modest overhead, as we show using a prototype Thoth-enabled data retrieval system based on the popular Apache Lucene.

1 Introduction

Online data retrieval systems typically serve a searchable corpus of documents, web pages, blogs, personal emails, online social network (OSN) profiles and posts, along with real-time microblogs, stock and news tickers. Examples include large providers like Amazon, Facebook, eBay, Google, and Microsoft, and also numerous smaller, domain-specific sharing, trading and networking sites run by organizations, enterprises, and governments.

Each data item served or used by a retrieval system may have its own usage policy. For instance, email is private to its sender/receiver(s), OSN data and blogs limited to friends, and corporate documents limited to employees. External data stream providers may restrict the use of (meta)data, and require expiration. The provider's privacy policy may require that a user's query and click stream be used only for personalization. Lastly, providers must comply with local laws, which may require them, for instance, to filter certain data items within a given jurisdiction.

Ensuring compliance with applicable policies is labor-intensive and error-prone [36]. The policy actually in effect for a data item may depend on checks and settings in many components and several layers of a system, making it difficult to audit and reason about. Moreover, any bug, misconfiguration, or compromise in a large and evolving application codebase could violate a policy. The problem affects both large providers with complex, fast evolving systems and smaller providers with limited IT budgets. Indeed, reports of data losses abound [14, 1, 44, 11, 13]. The stakes are high: providers stand to lose customer confidence, business and reputation, and may face fines. Hence, developing technical mechanisms to enforce policies in data retrieval systems is important. In fact, the Grok system combines lightweight static analysis with heuristics to annotate source code to check for policy violations in Bing's back-end [36].

Existing policy compliance systems for data retrieval, including Grok, usually target *column-specific policies*—policies that apply uniformly to all data of a specific type, e.g., the policy "no IP address can be used for advertizing." However, no existing work covers the equally important *individual policies* that are specific to individual data items or to a given client's data items. For example, Alice's blog posts, but not Bob's, may be subject to the policy "visible only to Alice's friends". Similarly, the expiration time of every item in a news ticker may be different. In fact, all policies mentioned a couple of paragraphs ago are individual policies. It is this (significant and important) missing part of policy enforcement that we wish to address in this paper. Specifically, we present Thoth, a policy compliance layer integrated into the Linux kernel to enforce both individual and column-specific policies efficiently.

We briefly describe the key insights in Thoth's design. First, by design, Thoth separates policies from application code. A policy specifying confidentiality and integrity requirements may be associated with any data conduit, i.e, a file, key-value tuple, named pipe or network connection, and is enforced on all application code that accesses the conduit's data or data derived from that data. Thoth provides a declarative language for specifying policies. The language itself is novel; in addition to standard access (read/write) policies, it also allows specifying data declassification policies by stipulating how

access policies may change along a data flow.

Second, unlike column-specific policies, individual policies may not be very amenable to static analysis because a given program variable may contain data with very different individual policies over time at the same program point and, hence, the abstraction of static analysis may lose precision quickly. So, Thoth uses dynamic analysis. It intercepts I/O in the kernel, tracks the flow of data at the granularity of conduits and processes (similar to Flume [28]), and enforces policies at process boundaries. This incurs a runtime overhead but we show that the overhead is not too high. With an optimized prototype implementation, we measure an overhead of 0.7% on indexing and 3.6% on query throughput in the widely used search engine Apache Lucene. While this overhead may be too high for large-scale data retrieval systems, we believe that it can be optimized further and that it is already suitable for domain-specific, medium-scale data retrieval systems run by organizations, enterprises and governments. Moreover, application code requires very few changes to run with Thoth (50 lines in a codebase of 300,000 LoC in our experiments).

Third, the complexity of a data retrieval system often necessitates some declassification to maintain functionality. For instance, a search process that consults an index computed over a corpus containing the private data of more than one individual cannot produce any readable results without declassification. To handle this and similar situations, we introduce a new form of declassification called *typed declassification*, which allows the declassification of data in specific forms (types). To accommodate the aforementioned search process, all source data policies allow declassification into a list of search results (document names). Hence, the search process can function as usual. At the same time, the possibility of data leaks is limited to a very narrow channel: To leak information from a private file, the search process' code must maliciously encode the information in a list of valid document names. Given that the provider has a genuine interest in preventing data breaches and that the search process is an internal component that is unlikely to be compromised in a casual external attack, the chance of having such malicious code in the search process is low. Thus, typed declassification is a pragmatic design point in the security-functionality trade-off for our threat model. Note that typed declassification needs content-dependent policies, which our policy language supports.

To summarize, the contributions of this work are: (1) A policy language that can express individual access and declassification policies declaratively (Section 2); (2) the design of a kernel-level monitor to enforce policies by I/O interception and lightweight taint propagation (Section 3); (3) application of the design to medium-scale data retrieval systems, specifically Apache's Lucene (Sections 2; 5); and (4) an optimized prototype implementation and experimental evaluation to measure overheads (Sections 4, 6).

2 Thoth policies

Thoth is a kernel-level policy compliance layer that helps data retrieval system providers enforce confidentiality and integrity policies on the data they collect and serve. In Thoth, the *provider* attaches policies to data sources (documents and live streams, posts and profiles, user click history, etc.) based on the privacy preferences of clients, external (e.g., legal) and internal usage requirements. Thoth *tracks data flows* by intercepting all IPC and I/O in the kernel, and it propagates source policies along these flows. It enforces policy conditions when data leaves the system, or when a declassification happens. The policy attached to a data source is a *complete, one point description* of all privacy and integrity rules in effect for that source.

Thoth policies are specified in a new, expressive declarative language, separate from application code. In this section, we describe this policy language briefly, discuss example policies that clients, data sources, and the provider might wish to enforce in a data retrieval system, and give a glimpse of how to express these policies in Thoth's policy language. More policy examples are included in Appendix A. Section 3 explains how Thoth enforces these policies. We note that our policy language and enforcement are general and apply beyond data retrieval systems.

Policy language overview A Thoth policy can be attached to any *conduit*—a file, key-value tuple, named pipe or network socket that stores data or carries data in transit. The policy on a conduit protects the confidentiality and integrity of the data in the conduit and is specified in two layers. The first layer, an *access control policy*, specifies which principals may **read** and **update** the conduit and under what conditions (e.g., only before or only after a certain date). A second layer protects data *derived* from the conduit by restricting the policies of conduits downstream in the data pipeline. This layer can **declassify** data by allowing the access policies downstream to be relaxed progressively, as more and more declassification conditions are met. The second layer that specifies declassification by controlling downstream policies is the language's key novelty.[1] Another noteworthy feature is that we allow policy evaluation to depend on a conduit's state—both its data and its metadata. This allows expressing content-dependent policies and, in particular, a kind of declassification that we call typed declassification.

[1] Our full language also supports provenance policies in the second layer by allowing control over upstream policies. Due to lack of space, we omit provenance policies here.

Arithmetic/string		Conduit		Content	
add(x,y,z)	x=y+z	cNameIs(x)	x is the conduit pathname	(c,off) says	$x_1,\ldots,x_n$ is the tuple found in
sub(x,y,z)	x=y-z	cIdIs(x)	x is the conduit id	$(x_1,\ldots,x_n)$	conduit c at off
mul(x,y,z)	x=y*z	cIdExists(x)	x is a valid conduit id	(c,off) willsay	ditto for the update of c in the
div(x,y,z)	x=y/z	cCurrLenIs(x)	x is the conduit length	$(x_1,\ldots,x_n)$	current transaction
rem(x,y,z)	x=y%z	cNewLenIs(x)	x is the new conduit length	each in (c,off) says	for each tuple in c at off, assign
concat(x,y)	x ‖ y	hasPol(c, p)	p is conduit c's policy	$(x_1,..,x_n)$ {condition}	to $x_1,..,x_n$ and evaluate condition
vType(x,y)	is x of type y?	cIsIntrinsic	does this conduit connect two confined processes?	each in (c,off) willsay $(x_1,..,x_n)$ {condition}	ditto for the update of c in the current transaction
Relational		**Session**		**Declassification rules**	
eq(x,y)	x=y	sKeyIs(x)	x is the session's	c1 until c2	condition c1 must hold on the
neq(x,y)	x!=y		authentication key		downstream flow until c2 holds
lt(x,y)	x<y	sIpIs(x)	x is the session's source IP	isAsRestrictive(p1,p2)	the permission p1 is at least as
gt(x,y)	x>y		address		restrictive as p2
le(x,y)	x<=y	IpPrefix(x,y)	x is IP prefix of y		
ge(x,y)	x>=y	timeIs(t)	t is the current time		

Table 1: Thoth policy language predicates and connectives

Layer 1: Access policies The first layer of a conduit's policy contains two rules that specify who can read and update the conduit's state under what conditions. We write both rules in the syntax of Datalog, which has been used widely in the past for the declarative specification of access policies [18, 20, 30]. Briefly, the read rule has the form (**read** :- cond) and means that the conduit can be read if the condition "cond" is satisfied. The condition "cond" consists of *predicates* connected with conjunction ("and", written $\wedge$) and disjunction ("or", written $\vee$). All supported predicates are listed in Table 1. Similarly, the update rule has the form (**update** :- cond).

Example (Client policies) Consider a search engine that indexes clients' private data. A relevant security goal might be that a client Alice's private emails and profile should be visible only to Alice, and only she should be able to modify this data. This *private data* policy can be expressed by attaching to each conduit holding Alice's private items **read** and **update** rules that allow these operations only in the context of a session authenticated with Alice's key. The latter condition can be expressed using a single predicate sKeyIs($k_{\texttt{Alice}}$), which means that the active session is authenticated with Alice's public key, denoted $k_{\texttt{Alice}}$. Hence, the read rule would be **read** :- sKeyIs($k_{\texttt{Alice}}$). The update rule would be **update** :- sKeyIs($k_{\texttt{Alice}}$). (Clients, or processes running on behalf of clients, authenticate directly to Thoth, so Thoth does not rely on untrusted applications for session authentication information.)

Alice's *friends only* blog and OSN profile should be readable by her friends as well, which can be expressed with an additional disjunctive clause in the read rule:

read :- sKeyIs($k_{\texttt{Alice}}$) $\vee$
(sKeyIs(K) $\wedge$ ("Alice.acl", `Offset`) says isFriend(K))

The part after the $\vee$ is read as "the key K that authenticated the current session exists in Alice.acl at some offset `Offset`." Here, Alice.acl is a trusted key-value tuple that contains Alice's friend list.

Following standard Datalog convention, terms like K and `Offset` that start with uppercase letters are existentially quantified variables. The predicate sKeyIs(K) binds K to the key that authenticates the session. During each policy evaluation, application code is expected to provide a binding for the variable `Offset` that refers to a location in the tuple's value saying that K belongs to a friend of Alice. Note that policy compliance does not depend on application correctness: if the application does not provide a correct offset, access will be denied.

Extending further, visibility to Alice's *friends of friends* can be allowed by modifying the read rule to check that Alice and the owner of the current session's key have a common friend. Then, the application code would be expected to provide an offset in Alice's acl where the common friend exists and an offset in the common friend's acl where the current session's key exists.

Layer 2: Declassification policies The second layer of a conduit's policy contains a single rule that controls the policies of downstream conduits. This rule is written (**declassify** :- cond), where "cond" is a condition or predicate on all *downstream sequences of conduits*. For instance, "cond" may say that in any downstream sequence of conduits, the access policies must allow read access only to Alice, until the calendar year is at least 2017, after which the policies may allow read access to anyone. This represents the declassification policy "private to Alice until the end of 2016".

We represent such declassification policies using the notation of *linear temporal logic* (LTL), a well-known syntax to represent predicates that change over time [32]. We allow one new connective in "cond" in the **declassify** rule: c1 until c2, which means that condition c1 must

hold of all downstream conduits until condition c2 holds. Also, we allow a new predicate isAsRestrictive(p1, p2), which checks that policy p1 is at least as restrictive as p2. The two together can represent expressive declassification policies, as we illustrate next.

Example (Index policy) In the last example, we discussed confidentiality policies that reflect data owners' privacy choices. For the retrieval system to do its job, however, the input data policies must allow some declassification. Without it, the search engine, which consults an index computed over the entire corpus, including the private data of several individuals, would not be allowed to produce any readable output. We rely on the policy language's novel ability to refer to a conduit's (meta-)data to allow the selective, *typed declassification* of search results. The policy can be implemented by adding the following **declassify** rule to all searchable input data:

$$\textbf{declassify} \text{ :- } \mathsf{isAsRestrictive}(\textbf{read}, \texttt{this}.\textbf{read}) \text{ until ONLY\_CND\_IDS}$$

This policy stipulates that data derived from Alice's data can be written into conduits whose read rule is at least as restrictive as Alice's (which is bound to `this`.**read**), until it is written into a conduit which satisfies the condition ONLY_CND_IDS. This macro stipulates that only a list of valid conduit ids has been written. The macro expands to

$$\begin{array}{l}\mathsf{cCurrLenIs}(\texttt{CurrLen}) \wedge \mathsf{cNewLenIs}(\texttt{NewLen}) \wedge \\ \mathsf{each\ in}(\mathsf{this}, \texttt{CurrLen}, \texttt{NewLen})\ \mathsf{says}(\texttt{CndId}) \\ \quad \{\mathsf{cIdExists}(\texttt{CndId})\}\end{array}$$

and permits the declassification of a list of proper conduit ids. A conduit id is a unique identifier for a conduit (conduit ids are defined in Section 3). The predicate "each in () says () {}" iterates over the sequence of tuples in the newly written data and checks that each is a valid conduit id. By including this declassification rule in her data item's policy, Alice allows the search engine to index her item and include it in search results. To view the contents, of course, a querier still has to satisfy each conduit's confidentiality policy.[2]

So far, we have assumed that the conduit ids (i.e., the names of indexed files) are not themselves confidential. If the conduit ids are themselves confidential, then the above **declassify** rule is insufficient since it stipulates no restriction on policies after ONLY_CND_IDS holds. Thus, a more restrictive **declassify** rule is needed. Ideally, we want that the read and declassify rules of the conduit that contains the list of conduit ids be at least as restrictive as the read and declassify rules of all conduits in the list. This can be accomplished by the following replacement for ONLY_CND_IDS.

$$\begin{array}{l}\mathsf{cCurrLenIs}(\texttt{CurrLen}) \wedge \mathsf{cNewLenIs}(\texttt{NewLen}) \wedge \\ \mathsf{each\ in}(\mathsf{this}, \texttt{CurrLen}, \texttt{NewLen})\ \mathsf{willsay}(\texttt{CndId}) \\ \{\mathsf{cIdExists}(\texttt{CndId}) \wedge \mathsf{hasPol}(\texttt{CndId}, \texttt{P}) \wedge \\ \ \mathsf{isAsRestrictive}(\textbf{read}, \texttt{P}.\textbf{read}) \wedge \\ \ \mathsf{isAsRestrictive}(\textbf{declassify}, \texttt{P}.\textbf{declassify})\}\end{array}$$

The predicate $\mathsf{hasPol}(\texttt{CndId}, \texttt{P})$ binds `P` to the policy of the conduit `CndId`, and the predicates $\mathsf{isAsRestrictive}(\textbf{read}, \texttt{P}.\textbf{read})$ and $\mathsf{isAsRestrictive}(\textbf{declassify}, \texttt{P}.\textbf{declassify})$ enforce that the read and declassify rules of the search results are at least as restrictive as those of `CndId`. We call this modified macro ONLY_CND_IDS+.

Other data retrieval policies

We briefly describe several other policies relevant to data retrieval systems that we have represented in our policy language and implemented in our prototype. For the formal encodings of these policies, see Appendix A.

Data analytics Many retrieval systems transform logs of user activity into a user preferences vector, which is used for targeting ads, computing user profiles, and providing recommendations. Raw logs of user clicks and queries are typically private, so a profile vector derived from them cannot be used for any of these purposes without a declassification. A policy that allows typed declassification into a vector of a fixed size can be attached to raw user logs to ensure that the raw logs cannot be leaked from the system, but that the profile vector can be used for the above-mentioned purposes.

Provider policies The provider may need to censor certain documents when a query arrives from a particular country. For this purpose, the system uses a map of IP address prefixes to countries. Separately, the provider maintains a per-country blacklist, containing a list of censored conduit ids. The *censorship policy* takes the form of a common declassification rule on source files. The rule requires that, at a conduit connecting to a client, the client's IP prefix is looked up in the prefix map, and the corresponding blacklist is checked to see if any of the search results are censored. Both the prefix map and the blacklist are maintained in sorted order for efficient lookup. The sort order is enforced by an integrity policy on the conduits.

A second common provider policy allows employees to access client's private data for troubleshooting pur-

[2] Our declassification policies can be intuitively viewed as state machines whose states are access policies and whose transitions are events in the data flow. For instance, the declassification policy just described is a two state machine, whose initial state has a read policy as restrictive as Alice's, and whose second state allows read access to everyone. The transition from the first to the second state is allowed when data passes through a conduit that satisfies ONLY_CND_IDS. This state-machine view of our policies is universal because it is well known that all LTL formulas can be represented as Büchi automata.

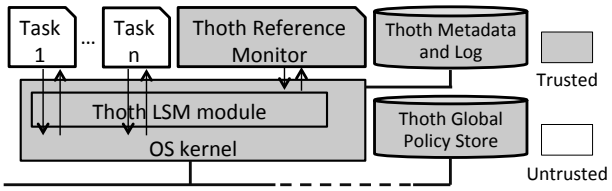

Figure 1: Thoth architecture

poses, as long as such accesses are logged for auditing. A *mandatory access logging (MAL)* policy can be added for this purpose. The policy allows accesses by authorized employees, if and only if an entry exists in a separate log file, which states a signature by the employee, the conduit being accessed, and a timestamp. The log file itself has an integrity policy that allows appends only, thus ensuring that an entry cannot be removed or overwritten. Finally, data sources must consent to provider access by allowing declassification into a conduit readable by authorized employees subject to MAL.

3 Thoth architecture and design

3.1 Overview

Figure 1 depicts the Thoth architecture. At each participating node, Thoth comprises a kernel module that intercepts I/O, a trusted reference monitor process that maintains per-task taint sets[3] and evaluates policies, a persistent store for metadata and transaction log, and a persistent policy store. Each node tracks taint and enforces policies independently of other nodes. The policy store is accessible exclusively by the reference monitors and provides a consistent view of all policies. This can be attained by using either central storage for policies or a consensus protocol like Paxos [29].

Figure 2 shows the data flow model of a Thoth-protected system. An application consists of a set of tasks (i.e., processes) that execute on one or more nodes. Data flows among the tasks via *conduits*. A file, named pipe or a tuple in a key-value store is a conduit. A network connection or a named pipe is a pair of conduits, one for each direction of data traffic. Thoth identifies each conduit with a unique numeric identifier, called the conduit id. The conduit id is the hash of the path name in case of a file or named pipe, the hash of the 5-tuple ⟨srcIP, srcPort, protocol, destIP, destPort⟩ in case of a network connection, or the key in case of a key-value tuple. Any conduit may have an associated policy.[4]

The core of the application system is a set of `CONFINED` tasks within Thoth's *confinement boundary*. The system interacts with the outside world via conduits (typically network connections) to external, `UNCONFINED` tasks. `UNCONFINED` tasks represent external users or components. Neither type of task is trusted by Thoth, although an `UNCONFINED` task may represent a user and may possess the user's authentication credentials.

Policies on inbound and outbound conduits that cross the confinement boundary represent the ingress and egress policies, respectively. The read and declassification rules of an ingress policy control how data can be used and disseminated by the system whereas the update rule of an ingress policy determines who may feed data into the system. The read rule of an egress policy defines who outside the system may read the output data.

3.2 Threat model

The Thoth kernel module and reference monitor, as well as the Linux system and policy store they depend on, are trusted. Active attacks on these components are out of scope. We assume that correct policies are installed on ingress and egress conduits. In our current prototype, storage systems that hold application data are assumed to be trusted. This assumption can be relaxed by encrypting and checksumming application data in the Thoth kernel module.

Thoth makes no assumptions about the nature of bugs and misconfigurations in application components, the type of errors committed by unprivileged operators, or errors in policies on internal conduits. Subject to this threat model, Thoth provably enforces all ingress policies. In information flow control terms, Thoth can control both explicit and implicit flows, but leaks due to covert and side-channels are out of scope.

Justification Trusting the Thoth kernel module, reference monitor, and the Linux system they depend on is reasonable in practice because (i) reputable providers will install security patches on the OS and Thoth components, and correct policies; (ii) OS and Thoth are maintained by a small team of experts and are more stable than applications; thus, an attacker will likely find it more difficult to find a vulnerability in the OS or Thoth than in a rapidly evolving application with a large attack surface.

Typed declassification policies admit limited information flows, which can be exploited by malicious applications covertly. For instance, malware injected into a search engine can encode private information in the set of conduit ids it produces, if the conduits in the set themselves are public. This channel is out of scope. In practice, such attacks require significant sophistication. A successful attack must inject code strategically into the data flow before a declassification point and encode private data on a policy-compliant flow.

On the other hand, Thoth prevents the large class of practical attacks that involve direct flows to unauthorized parties, and accidental policy violations due to applica-

[3] A task's taint set is the set of policies of conduits it has read.

[4] If a file has multiple hard links, each of its path names can be associated with a different policy. When a path name is used to access the file, that path name's policies are checked.

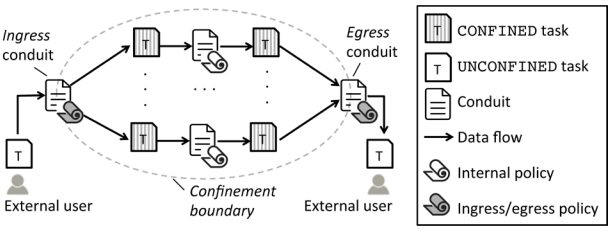

Figure 2: Thoth data flow

tion bugs, misconfigurations, and errors by unprivileged operators. We demonstrate this in Section 6.3 where a Thoth compliant search engine is able to enforce data policies, preventing (real and synthetic) bugs and misconfigurations from leaking information.

3.3 Data flow tracking and enforcement

Tracking data flow Thoth tracks data flows coarsely at the task-level. CONFINED and UNCONFINED tasks are subject to different policy checks. A CONFINED task may read any conduit, irrespective of the conduit's **read** rule, but Thoth enforces each such conduit's **declassify** rule when the task writes to other conduits. To do this, Thoth maintains the **declassify** rules of conduits read by each CONFINED task in the task's metadata (these rules constitute the *taint set* of the task).

UNCONFINED tasks form the ingress and egress points for Thoth's flow tracking; they are subject to access control checks, not tainting. An UNCONFINED task may read from (write to) a conduit only if the conduit's **read** (**update**) rule is satisfied. For example, to read Alice's private data, an UNCONFINED task must authenticate with Alice's credentials. Conduits without policies can be read and written by all tasks freely.

In summary, Thoth tracks data flows across CONFINED tasks coarsely, and enforces declassification policies on these flows. At the ingress and egress tasks (UNCONFINED tasks), Thoth imposes access control through the **read** and **update** rules. Every new task starts UNCONFINED. The task may transition to the CONFINED state through a designated Thoth API call. The reverse transition is disallowed to prevent a task from reading private data in the CONFINED state and leaking the data to a conduit without any policy protection after transitioning to the UNCONFINED state.

Conduit interceptors The Thoth kernel component includes a conduit interceptor (CI) for each type of conduit. A CI for a given conduit type intercepts system calls that access or manipulate conduits of that type, and associates a conduit with its policy. Thoth has built-in CIs for kernel-defined conduit types, namely files, named pipes, and network connections. CIs for additional conduit types can be plugged in. For instance, our prototype uses a CI for the memcached key-value store (KV).

Question: Should a conduit read or write be allowed?
Inputs: t, the task reading or writing the conduit
f, the conduit being read or written
op, the operation being performed (read or write)
Output: Allow or deny the access
Side-effects: May update the taint set of t

```
1  if t is UNCONFINED:
2    if op is read:
3      Check f's read rule.
4    if op is write:
5      Check f's update rule.

6  if t is CONFINED:
7    if op is read:
8      Add f's policy to t's taint set.
9    if op is write:
10     // Enforce declassification policies of t's taint set
11     for each declassification rule (c until c') in t's taint set:
12       Check that EITHER c' holds OR (c holds AND
         f's declassification policy implies (c until c')).
```

Figure 3: Thoth policy enforcement algorithm

The CIs for files and named pipes associate a policy with the unique pathname of a file or pipe. The socket CI associates a policy with the network connection's 5-tuple ⟨srcIP, srcPort, protocol, destIP, destPort⟩. The 5-tuple may be underspecified. For instance, the policy associated with ⟨?, ?, ?, destIP, destPort⟩ applies to any network connection with the specified destination IP address and port. Both ends of a network connection have the same policy. The KV CI associates a policy with a tuple's key. The KV CI can automatically derive policies from policy templates that cover a subspace of keys (e.g., all keys with prefix *#user_profile*). It can also replace template variables with metadata, e.g., the time at which the key was created.

Policy enforcement algorithm Figure 3 summarizes the abstract checks that Thoth makes when it intercepts a conduit access. If the calling task is UNCONFINED, then Thoth evaluates the read or update policy of the conduit (lines 1–5). If the calling task is CONFINED and the operation is a read, then Thoth adds the policy of the conduit being read to the taint set of the calling task. No policy check is performed in this case (lines 6–8). To reduce the size of a CONFINED task's taint set, our prototype performs *taint compression* when possible: A policy is not added if the taint set already includes an equally or more restrictive policy.

When a CONFINED task t writes a conduit f, there is a potential data flow from every conduit that t has read in the past to f. Hence, all declassification rules in t's taint set are enforced (lines 11–12). Suppose (c until c') is a **declassification** rule in t's taint set. Since this rule means

that condition c must continue to hold downstream *until* the declassification condition c' holds, this rule can be satisfied in one of two ways: Either the declassification condition c' holds now, or c holds now and the next downstream conduit (f here) continues to enforce (c until c'). Line 12 makes exactly this check.

End-to-end correctness of policy enforcement Within Thoth's threat model, the checks described above enforce all policies on conduits and, specifically, all ingress policies. Incorrect policy configuration on internal conduits cannot cause violation of ingress policies but may cause compliant data flows to be denied by the Thoth reference monitor. Informally, this holds because our checks ensure that the conditions in every declassification policy are propagated downstream until they are satisfied. [5]

Policy comparison Thoth compares policies for restrictiveness in three cases: for taint compression, when evaluating the predicate isAsRestrictive(), and in line 12 of the enforcement algorithm (Figure 3). The general comparison problem is undecidable for first-order logic, so Thoth uses the following heuristics: 1) *Equality*: Compare the hashes of the two policies. 2) *Inclusion*: Check that all predicates in the less restrictive policy also appear in the more restrictive one, taking into account variable renaming and conjunctions and disjunctions between the predicates. Inclusion has exponential time complexity in the worst case, but is fast in practice. 3) *Partial evaluation*: Evaluate and delete an application-specified subpart of each policy, then try equality and inclusion. These heuristics suffice in all cases we have encountered.

Note that a policy comparison failure can never affect Thoth's safety. However, a failure can (a) defeat taint compression and therefore increase taint size and policy evaluation overhead; or (b) cause a compliant data flow to be denied. In the latter case, a policy designer may re-state a policy so that the policy comparison succeeds.

3.4 Thoth API

Table 2 lists Thoth API functions provided to user-level tasks by means of a new system call. To check structural properties of written data (e.g., that the data is a list of conduit ids), it is often necessary to evaluate the **update** rule atomically on a *batch* of writes. Hence, Thoth supports write transactions on conduits. By default, a transaction starts with a POSIX open() call and ends with the close() call on a conduit. This behavior can be overridden by passing additional flags to open() and close(). Transactions can also be explicitly started and ended using the Thoth API calls *open_tx* and *close_tx*.

[5] A formal proof of this fact is the subject of a forthcoming paper. Our formal model and implementation support nested uses of the until operator, which we omitted here.

During a transaction, Thoth buffers writes in a persistent re-do log. When the transaction is closed by the application, Thoth makes the policy checks described in Figure 3. If the policy checks succeed, then the writes are sent to the conduit, else the writes are discarded. The re-do log allows recovery from crashes and avoids expensive filesystem syncs when a transaction commits.

Summary Thoth enforces ingress and egress policies despite application-level bugs, misconfigurations, and compromises, or actions by unprivileged operators. A data source's policy specifies both access and declassification conditions and describes the source's allowed uses completely. Thoth uses policies as taint, which differs significantly from the standard information flow control practice of using abstract labels as taint. That practice requires trusted application processes to declassify data and to control access at system edges. In contrast, Thoth relies entirely on its reference monitor for all access and declassification checks, and no application processes have to be trusted.

4 Thoth prototype

Our prototype consists of a Linux kernel module that plugs into the Linux Security Module (LSM) interface, and a reference monitor. We also changed a few (22) lines of the kernel proper to provide additional system call interception hooks not included in the LSM interface, and a new system call that allows applications to interact with Thoth. A small application library consisting of 840 LoC exports the API calls shown in Table 2 based on this system call.

LSM module The Thoth LSM module comprises approximately 3500 LoC and intercepts I/O related system calls including open, close, read, write, socket, mknod, mmap, etc. Intercepted system calls are redirected to the reference monitor for taint tracking and validation. The module includes conduit interceptors for files, named pipes and sockets, as well as interceptors for client connections to a memcached key-value store [12].

Thoth reference monitor Thoth's reference monitor is implemented as a trusted, privileged userspace process. It implements the policy enforcement logic and maintains the process taint, session state and transaction state in DRAM. The monitor accesses the persistent Thoth metadata store, which includes per-conduit metadata (conduit pathname, conduit id, a pointer to the policy in effect in the policy store, and for each persistent file conduit, its current size), the transaction log, and the global policy store. The metadata and transaction log are stored in NVRAM. A write-through DRAM cache holds recently accessed metadata and policies.

The monitor is multi-threaded so it can exploit multicore parallelism. Each worker thread invokes the Thoth

Function	Description
confine ()	Transition calling process from `UNCONFINED` to `CONFINED` state.
authenticate (key)	Authenticate process with the private key *key* to satisfy identity-based policies.
add_policy (p)	Store a policy *p* in Thoth metadata and return an id *p_id* for it.
set_tx_flags (c_id, flags)	Set flags *flags* (type and partial evaluation hints) for a transaction on conduit *c_id*.
open_tx (c_id)	Open a transaction on conduit *c_id* and return a file handle.
close_tx (fd)	Close a transaction *fd*. Return 0 if successful, or error code of a policy check fails.
set_policy (fd, p_id)	Attach policy id *p_id* to the conduit running transaction *fd*. Passing (-1) for *p_id* sets the null policy. The new policy is applied only after *fd* is successfully closed. The declassification condition of the conduit's existing policy determines whether the policy change or removal is allowed.
get_policy (c_id, buf)	Retrieve the policy attached to conduit *c_id* into buffer *buf*.
cache (fd, off, len)	Cache content (for policy evaluation) from file handle *fd* from offset *off* with length *len*.

Table 2: Thoth API calls

system call and normally blocks in the LSM module waiting for work. When an application issues a system call that requires an action by the reference monitor, a worker thread is unblocked and returns to the reference monitor with appropriate parameters; when the work is done, the thread invokes the system call again with the appropriate results causing the original application call to either be resumed or terminated. As an optimization, the LSM seeks to amortize the cost of IPC by buffering and dispatching multiple asynchronous requests to a worker thread whenever possible. The reference monitor was implemented in 19,000 LoC of C, not counting the OpenSSL library used for secure sessions and crypto.

Limitations Memory-mapped files are currently supported read-only. Interception is not yet implemented for all I/O-related system calls. None of these missing features are used by our prototype data retrieval system.

5 Policy-compliant data retrieval

We use Thoth for policy compliance in a data retrieval system built around a distributed Apache Lucene search engine. While Apache Lucene's architecture is not appropriate for large, public search engines like Google or Bing, it is frequently used in smaller, domain-specific data retrieval systems.

5.1 Baseline configuration

Lucene Apache Lucene is an open-source search engine written in Java [2]. It consists of an indexer and a search component. The sequential indexer is a single process that scans a corpus of documents and produces a set of index files. The search component consists of a multi-threaded process that executes search queries in parallel and produces a set of corpus file names relevant to a given search query. The size of the Apache Lucene codebase is about 300,000 LoC.

Lucene can be configured with replicated search processes to scale its throughput. Here, multiple nodes run a copy of the search component, each with the full index. A search query can be processed by any machine. Lucene can also be sharded to scale with respect to the corpus size. In this case, the corpus is partitioned, each partition is indexed individually, and multiple nodes run a copy of the search component, each with one partition index. A search query is sent to all search components, and the results combined. Replication and sharding can be combined in the obvious way.

Front-end processes A simple front-end process accepts user requests from a remote client and forwards search queries to one or more search process(es) via a pipe. The search process(es) may forward the query to other search processes with disjoint shards. When the front-end receives the search results (a list of document file names), it produces a HTML page with a URL and a content snippet from each of the result documents, and returns the page to the Web client. When the client clicks on one of the URLs, the front-end serves the content.

A second, simple account manager front-end process accepts connections from clients for the purpose of creating accounts, managing personal profiles and policies. Clients choose from a set of policy templates for documents they have contributed to the corpus, and for their personal profile information and activity history.

Search personalization and advertising To include typical features of a data retrieval system, we added personalized search and targeted advertising components. A memcached daemon runs on each search node to provide a distributed key-value store for per-user information, including a suffix of the search and click histories, profile information, and the public key. The front-end process appends a user's search queries and clicks to the histories. It uses the profile information to rewrite search queries, re-order search results, and select ads for inclusion in the results page.

An aggregator process periodically analyses a user's search and click history, and updates the personal profile information accordingly. We are not concerned with the details of user profiling, personalized search, or ad

targeting. It suffices for our purposes to capture the appropriate data flows.

5.2 Controlling data flow with Thoth

With Thoth, the front-end, search, indexing, and aggregation tasks execute as `CONFINED` processes, and the account manager executes as an `UNCONFINED` process. Relative to the baseline system, we made minimal modifications, mostly to set an appropriate policy on output conduits. The modifications to Apache Lucene amounted to less than 20 lines of Java code and 30 lines of C code in a JNI library. These modifications set policies on internal conduits and, like the rest of Lucene, are not trusted. Finding the appropriate points to modify was relatively easy because Lucene's codebase has separate functions through which all I/O is channelled. For applications without this modularity, a dynamically-linked library can be used that overrides libc's I/O functions and adds appropriate policies.

Unlike in the baseline, the front-end process must be restarted after each user session, to drop its taint. We implement this by exec-ing the process when a new user session starts.

Ingress/egress policies Recall that the ingress and egress policies determine which data flows are allowed and reflect the policies of users, data sources, and provider. In our system, the network connection between the client and the front-end is both an ingress and an egress conduit. The document files in the corpus and the key-value tuples that contain a user's personal information are ingress conduits. Policies are associated with all ingress and egress conduits as described below. The primary difficulty here is to determine appropriate policies, a task that is required in any compliant system. Specifying the policies in Thoth's policy language is straightforward.

Account manager flow When Alice creates an account, credentials are exchanged for subsequent mutual authentication, and stored in the key-value store, along with any personal profile information Alice provides.

Alice can choose policies for her profile and history information, as well as any contributed content, typically from a set of policy templates written by the provider's compliance team. The declassification rule of each policy implicitly controls who can subsequently change the policy; normally, Alice would choose a policy that allows only her to make such a change. Alice may also edit her friend lists or other access control lists stored in the key-value store, which may be referenced by her policies.

Next, we explain the main data flows through the system. For lack of space, we cannot detail all policies on internal conduits, but we highlight the key steps.

Indexing flow Periodically, the indexer is invoked to regenerate the index partitions. A correct indexer only processes documents with the ONLY_CND_IDS (or ONLY_CND_IDS+) declassification clause, which is transferred to the index files. Note that the index may contain arbitrary data and can be read by any `CONFINED` process; however, an eventual declassification to an `UNCONFINED` process is only possible for a list of conduit ids.

Profile aggregation flow A profile aggregation task periodically executes in the background, to scan the suffix of a user's query and click history and update the user's profile vector. A correct aggregator only analyzes user history data that has the ONLY_CND_IDS (or ONLY_CND_IDS+) declassification clause, which is transferred to the profile vectors.

Search flow Finally, we describe the sequence of steps when Alice performs a search query. The search front-end authenticates itself to Alice using the credentials stored in the key-value store. A successful authentication assures Alice that (i) she is talking to the front-end, and (ii) the front-end process is tainted with the policy of Alice's credentials (only Alice can read, else declassify into a list of conduit ids) before Alice sends her search query. Next, Alice authenticates herself to the Thoth reference monitor via the search front-end, which proves to Thoth that the front-end process speaks for Alice.

The front-end now sends Alice's query to one or more search process(es) and adds it to her search history. The search results are declassified as a list of conduit ids, and therefore do not add new taint to the front-end. While producing the HTML results page, the front-end reads a snippet from each result document using Alice's credentials. Each document has a censorship policy, which checks that the document's conduit ID is not blacklisted in the client's region. These policies differ in the conduit IDs and so, in principle, the taint set on the front-end could become very large. To prevent this, we use partial evaluation (Section 3): *Before* a document's policy is added to the front-end's taint, we check that the document is not blacklisted. This way, the front-end's taint increases by a *single predicate* (which verifies Alice's IP address) when it reads the first document and does not increase when it reads subsequent documents.

Finally, the front-end sends the results page to the client. For this, it must satisfy the egress conduit policy, which verifies Alice's identity and her IP address.

Result caching High-performance retrieval systems cache search results and content snippets for reuse in similar queries. Although we have not implemented such caching, it can be supported by Thoth. Intermediate results can be cached at various points in the data flow, usually before their policies have been specialized (through

partial evaluation) for a particular client or jurisdiction.

Summary Assuming that the account manager correctly installs ingress and egress policies, Thoth ensures that Alice's documents, history and profile are used according to her wishes and that the provider's censorship and MAL policies are enforced, despite any bugs in the indexer, the front-end or the profile aggregator. Thoth's use in a data retrieval system highlights two different ways of preventing process overtainting. The front-end process is *user-specific*—it acts on behalf of one client. Consequently, the front-end must be re-execed at the end of a user session session to discard its taint. In contrast, the indexer is an *aggregator* process that is designed to combine documents with conflicting policies into a single index. To make its output (the index) usable downstream, the provider installs a typed declassification clause (ONLY_CND_IDS or ONLY_CND_IDS+) on all documents. Due to the declassification clause, there is no need to re-exec the search process.

6 Evaluation

In this section, we present results of an experimental evaluation of our Thoth prototype.

All experiments were performed on Dell R410 servers, each with 2x Intel Xeon X5650 2.66 GHz 6 hyper-threaded core CPUs, 48GB main memory, running OpenSuse Linux 12.1 (kernel version 3.13.1, x86-64). The servers are connected to Cisco Nexus 7018 switches with 1Gbit Ethernet links. Each server has a 1TB Seagate ST31000424SS disk formatted under ext4, which contains the OS installation and a 258GB static snapshot of English language Wikipedia articles from 2008 [43].

We allocate a 2GB memory segment on /dev/shm to simulate NVRAM used by Thoth to store its metadata and transaction log. NVRAM is readily available and commonly used to store frequently updated, fixed-sized persistent data structures like transaction logs.

In the following experiments, we compare a system where each OS kernel is configured with the Thoth LSM kernel module and reference monitor against an otherwise identical baseline system with unmodified Linux 3.13.1 kernels.

6.1 Thoth-based data retrieval system

We study the total Thoth overheads in the prototype retrieval system described in Section 4.

Indexing First, we measure the overhead of the search engine's index computation. We run the Lucene indexer over a) the entire 258GB snapshot of the English Wikipedia, and b) a 5GB part of the snapshot. The sizes of the resulting indices are 54GB and 959MB, respectively. Table 3 shows the average indexing time and standard deviation across 3 runs. In both cases, Thoth's runtime overhead is below 1%.

	Dataset 258GB		Dataset 5GB	
	Avg. (mins)	σ	Avg. (mins)	σ
`Linux`	1956.1	30	27.8	0.06
`Thoth`	1968.6	24	28.0	0.11
Overhead	0.65%		0.7%	

Table 3: Indexing runtime overhead

Even in a sharded configuration, Lucene relies on a sequential indexer, which can become a bottleneck when a corpus is large and dynamic. Larger search engines may rely on parallel map/reduce jobs to produce their index. As a proof of concept, we built a Hadoop-based indexer using Thoth, although we don't use it in the following evaluation because it does not support all the features of the Lucene indexer. All mappers and reducers run as confined tasks, and receive the same taint as the original, sequential indexer.

Search throughput Next, we measure the overhead of Thoth on the query latency and throughput. To ensure load balance, we partitioned the index into two shards of 22GB and 33GB, chosen to achieve approximately equal query throughput. We use two configurations: **2SERVERS**: 2 server machines execute a Lucene instance with different index shards. **4SERVERS**: Here, we use two replicated Lucene instances in each shard to scale the throughput. The front-end forwards each search request to one of the two Lucene instances in each shard and merges the results.

We drive the experiment with the following workload. We simulate a population of 40,000 users, where each user is assigned a friend list consisting of 12 randomly chosen other users, subject to the constraint that the friendship relationship is symmetric. Each item in the corpus is assigned either a private, public, or friends-only policy in the proportion 30/50/20%, respectively. A total of 1.0% of the dataset is censored in some region. All simulated clients are in a region that blacklists 2250 random items.

We use query strings based on the popularity of Wikipedia page accesses during one hour on April 1, 2012 [42]. Specifically, we search for the titles of the top 20K visited articles and assign each of the queries randomly to one of the users. 24 simulated active users connect to each server machine, maintain their sessions throughout the experiment, and issue 48 (**2SERVERS**) and 96 (**4SERVERS**) queries concurrently to saturate the system. In addition, a simulated "employee" sporadically issues a read access to protected user files for a total of 200 MAL accesses.

During each query, the front-end looks up the user profile and updates the user's search history in the key-value store. To maximize the performance of the baseline and fully expose Thoth's overheads, the index shard and

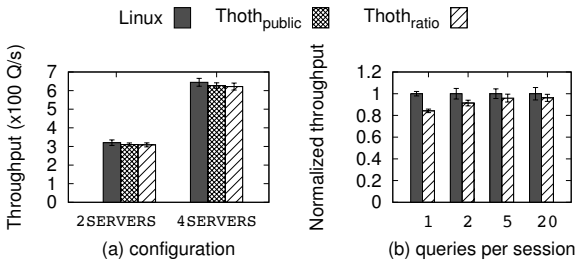

Figure 4: Search throughput

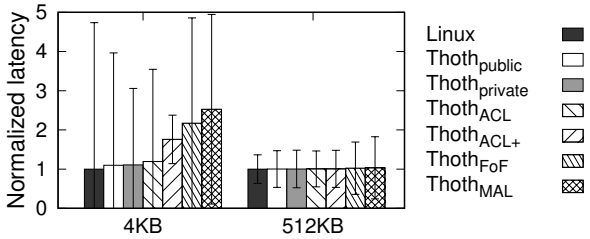

Figure 5: Read latency, normalized to Linux's

parts of the corpus relevant to our query stream are preloaded into the servers' main memory caches, resulting in a CPU-bound workload.

Figure 4 (a) shows the average throughput over 10 runs of 20K queries each, for the baseline (Linux) and Thoth under **2SERVERS** and **4SERVERS**. The error bars indicate the standard deviation over the 10 runs. We used two Thoth configurations, **Thoth$_{public}$** and **Thoth$_{ratio}$**. In **Thoth$_{public}$**, the policies permit all accesses. This configuration helps to isolate the overhead of Thoth's I/O interposition and reference monitor invocation. In **Thoth$_{ratio}$**, input files are private to a user, public, or accessible to friends-only in the ratio 30:50:20. All files allow employee access under MAL, enforce region-based censorship, and have the declassification condition with ONLY_CONDUIT_IDS+.

The query throughput scales approximately linearly from **2SERVERS** (320 Q/s) to **4SERVERS** (644 Q/s), as expected. Thoth with all policies enforced (**Thoth$_{ratio}$**) has an overhead of 3.63% (308 Q/s) in **2SERVERS** and 3.55% in **4SERVERS** (621 Q/s). We note that the throughput achieved with **Thoth$_{public}$** (310 Q/s and 627 Q/s, respectively) is only slightly higher than **Thoth$_{ratio}$**'s. This suggests that Thoth's overhead is dominated by costs like I/O interception, Thoth API calls, and metadata operations, which are unrelated to policy complexity.

To test whether overheads can be reduced further, we also implemented a rudimentary reference monitor in the kernel, which does not support session management and policy interpretation (which require libraries that are unavailable in the Linux kernel). This reduced in-kernel monitor suffices to execute **Thoth$_{public}$**. Moving the reference monitor to the kernel reduced the overhead of **Thoth$_{public}$** from 3% to under 1%, which suggests that overheads can be further reduced my moving the reference monitor to the kernel and, hence, eliminating the cost of IPC between the LSM and the reference monitor.

With Thoth, the front-end is re-exec'ed at the end of every user session to shed the front-end's taint. The relative overhead of doing so reduces with session length. Figure 4 (b) shows the average throughput normalized to the Linux baseline for session lengths of 1, 2, 5 and 20 queries in **2SERVERS**. Due to the per-session front-end exec, Thoth's overhead is higher for small sessions (15.8% for a single query); however, the overhead diminishes quickly to 8.6% for 2 queries per session, and the throughput is within a standard deviation of the maximum for 5 or more queries per session in all configurations, including **4SERVERS**.

Search latency Next, we measure the overhead on query latency. Table 4 shows the average query latency across 5 runs of 10K queries in **2SERVERS**. The results in **4SERVERS** are similar. In all cases, Thoth adds less than 6.7ms to the baseline latency.

	Avg. (ms)	σ	Overhead
`Linux`	47.09	0.43	
Thoth$_{public}$	51.60	0.29	9.6%
Thoth$_{ratio}$	53.78	0.20	14.2%

Table 4: Query search latency (ms)

6.2 Microbenchmarks

Next, we perform a set of microbenchmarks to isolate Thoth's overheads on different policies. We measure the latency of opening, reading sequentially, and closing 10K files in the baseline and with Thoth under different policies associated with the files. The files were previously written to disk sequentially to ensure fast sequential read performance for the baseline and therefore fully expose the overheads.

In the Thoth experiments, accesses are performed by an `UNCONFINED` task to force an immediate policy evaluation. The following policies are used. **Thoth$_{public}$**: files can be read by anyone. **Thoth$_{private}$**: access is restricted to a specific user. **Thoth$_{ACL}$**: access to friends only (all users have the same friend list). **Thoth$_{ACL+}$**: access to friends only (each user has a different friend list). **Thoth$_{FoF}$**: access to friends of friends (each user has a different friend list). All friend lists used in the microbenchmark have 100 entries. **Thoth$_{MAL}$**: each file has a MAL policy, where each read requires an entry in a log with an append-only integrity policy.

Figure 5 shows the average time for reading a file of sizes 4K and 512K, normalized to the baseline Linux latency (0.145ms and 3.6ms, respectively); the error bars indicate the standard deviation among the 10K file reads. We see that Thoth's overheads increase with the complexity of the policy, in the order listed above. For the 4KB files, the overheads range from 10.6% for $\textbf{Thoth}_{\textbf{public}}$ and $\textbf{Thoth}_{\textbf{private}}$ to 152.7% for $\textbf{Thoth}_{\textbf{MAL}}$. The same trend holds for larger files, but the overhead range diminishes to 0.6%–23% for 96KB files (not shown in the figure) and 0.34%–3.3% for 512KB files.

We also experimented with friend list sizes of 12 and 50 entries for $\textbf{Thoth}_{\textbf{ACL}}$, $\textbf{Thoth}_{\textbf{ACL+}}$ and $\textbf{Thoth}_{\textbf{FoF}}$; the resulting latency was within 2.4% of the corresponding 100-entry friend list latency. This is consistent with the known complexity of the friend lookup, which is logarithmic in the list size.

We also looked at the breakdown of Thoth latency overheads. With $\textbf{Thoth}_{\textbf{ACL}}$ and 4KB files, Thoth's overhead for file read is on average 28µs, which are spent intercepting the system call and maintaining the session state. Interpreting the policy and checking the friend lists takes 6µs, but this time is completely overlapped with the disk read.

Write transaction latency We performed similar microbenchmarks for write transactions. In general, Thoth's write transactions have low overhead since its transaction log is stored in (simulated) NVRAM. As in the case of read latency, the overhead depends on the granularity of writes and the complexity of the policy being enforced. Under the index policy, the overhead ranges from 0.25% for creating large files to 2.53x in the case of small files. The baseline Linux is very fast at creating small files that are written to disk asynchronously, while Thoth has to synchronously update its policy store when a new file is created. The overhead is 5.8x and 8.6x in the case of a write of 10 conduit ids to a file under the ONLY_CND_IDS and ONLY_CND_IDS+ policies, respectively. This high overhead is due to checking that each conduit id being written exists (and is written into a file with a stricter policy in the case of ONLY_CND_IDS+). However, this overhead amounts to only a small percentage of the overall search query processing, as is evident from Table 4.

6.3 Fault-injection tests

To double-check Thoth's ability to stop unwanted data leaks, we injected several types of faults in different stages of the search pipeline.

Faulty Lucene indexer We reproduced a known Lucene bug [5] that associates documents with wrong attributes during index creation. This bug is security-relevant because, in the absence of another mechanism, attributes can be used for labeling data with their owners. In our experiment Thoth successfully stopped the flow in all cases where the search results contained a conduit whose policy disallowed access to the client.

We also intentionally misconfigured the indexer to index the users' query and click histories, which should not show up in search results. Thoth prevented the indexer from writing the index after it had read either the query or the click history.

Faulty Lucene search We reproduced a number of known Lucene bugs that produce incorrect search results. Such bugs may produce Alice's private documents in Bob's search. The bugs include incorrect parsing of special characters [7], incorrect tokenization [9], confusing uppercase and lowercase letters [10], using an incorrect logic for query expansion [4, 3], applying incorrect keyword filters [8], and premature search termination [6]. We confirmed that all policy violations resulting from these bugs faults were blocked by Thoth.

To check the declassification condition ONLY_CND_IDS+, we modified the search process to (incorrectly) output text from the index in place of conduit ids. Thoth prevented the search process from producing such output.

Faulty front-end We issued accesses to a private file protected by the MAL policy without adding appropriate log entries. Thoth prevented the front-end process from extricating data to the caller. We performed similar tests for the region-based censorship policy with similar results.

7 Related work

Search engine policy compliance Grok [36] is a privacy compliance tool for the Bing search engine. Grok and Thoth differ in techniques, expressiveness and target policies. Grok uses heuristics and selective manual verification by developers to assign *attributes* — abstract labels that represent intended confidentiality — to processes and data stores. Grok policies, written in a language called Legalese, specify allowed data flows on attributes. Attributes and policies apply at the granularity of fields (types), not individual users or data items, so Legalese cannot express the private, friends only and friends of friends policies from Section 2. (This restriction applies broadly to most static analysis-based policy enforcement techniques.) Legalese also does not support content-dependent policies and cannot express the mandatory access logging, censorship and typed declassification policies from Section 2. Grok enforces policies with a fast static analysis on computations written in languages like Hive, Dremel, and Scope. Grok imposes no runtime overhead. Thoth uses kernel-level interception and is language-independent, but has a small runtime overhead. Grok-assigned attributes may be incorrect, so Grok may have false negatives. In contrast,

Thoth enforces all conduit policies without false negatives.

Cloud policy compliance Maniatis et al. [31] outline a vision, architecture and challenges for data protection in the Cloud using secure data capsules. Thoth can be viewed as a realization of that vision in the context of a data retrieval system, and contributes the design of a policy language, enforcement mechanism, and experimental evaluation. Secure Data Preservers (SDaPs) [27] are software components that mediate access to data according to a user-provided policy. Unlike Thoth, SDaPs are suitable only for web services that interact with user data through simple, narrow interfaces, and do not require direct access to users' raw data. LoNet [26] enforces data-use policies at the VM-level. Unlike Thoth, declassification requires trusted application code and interception is limited to file I/O using FUSE, which results in very high overhead.

Information flow control (IFC) Numerous systems restrict a program's data flow to enforce security policies, either in the programming language (Jif [34]), in the language runtime (Resin [46], Nemesis [19]), in language libraries (Hails [25]), using software fault isolation (duPro [35]), in the OS kernel (e.g., Asbestos [22], HiStar [47], Flume [28], Silverline [33]), or in a hypervisor (Neon [48]). Thoth differs from these systems in a number of ways. Unlike language-based IFC, Thoth applications can be written in any language.

Architecturally, Thoth is close to Flume. Both isolate processes using a Linux security extension and a user-space reference monitor, both enforce policies on conduits and both distinguish between `CONFINED` and `UNCONFINED` processes in similar ways. However, like all other kernel-level solutions for IFC (Asbestos, HiStar, Silverline), Flume uses abstract labels as taints. In contrast, Thoth uses declarative policies as taints. This results in two fundamental differences. First, Flume relies on trusted application components to map system access policies to abstract labels and for all declassification. In contrast, in Thoth, the reference monitor enforces all access conditions (specified in the **read** and **update** rules) and all declassification conditions (specified in the **declassify** clauses). Application components are trusted only to install correct policies on ingress and egress nodes. Second, Thoth policies describe the policy configuration completely. In Flume, the policy configuration is implicit in the *code* of the trusted components that declassify and endorse data, and map access policies to labels (although mapping can be automated to some extent [21]).

Resin [46] enforces programmer-provided policies on PHP and Python web applications. Unlike Thoth's declarative policies, Resin's policies are specified as PHP/Python functions. Resin tracks flows at object granularity. Thoth tracks flows at process granularity, which matches the pipelined structure of data retrieval systems and reduces overhead significantly. Hails [25] is a Haskell-based web development framework with statically-enforced IFC. Thoth offers IFC in the kernel, and is independent of any language, runtime, or framework used for developing applications. COWL [39] confines JavaScript browser contexts using labels and IFC. Thoth addresses the complementary problem of controlling data flows on the server side. Both Hails and COWL use DC-labels [38] as policies. DC-labels cannot express content-dependent policies like our censorship, mandatory access logging and ONLY_CND_IDS policies.

Declarative policies Thoth's policy language is based on Datalog and linear temporal logic (LTL). Datalog and LTL are well-studied foundations for policy languages (see [30, 18, 20] and [15, 16, 23], respectively), known for their clarity, conciseness, and high-level of abstraction. The primary innovation in Thoth's policy language is its two-layered structure, where the first layer specifies access policies and the second layer specifies declassification policies. Some operating systems (Nexus and Taos [37, 45]), file systems (PFS and PCFS [41, 24]), and at least one cyber-physical system (Grey [17]) and one storage system (Guardat [40]) enforce access policies expressed in Datalog-like languages. Thoth can enforce similar policies but, additionally, Thoth tracks flows and can enforce declassification policies that these systems cannot enforce. Like Guardat, but unlike the other systems listed above, Thoth's policy language supports data-dependent policies. The design of Thoth's reference monitor is inspired by Guardat's monitor. However, Thoth's monitor tracks data flows, supports declassification policies, and intercepts memcached I/O and network communication, all of which Guardat's monitor does not do.

8 Ongoing work

In this section, we briefly describe ongoing work related to Thoth.

Lightweight isolation Information flow control requires the isolation of computations that handle different users' private data. In general-purpose operating systems, this means that separate processes must be used to handle user sessions. Thoth, for instance, requires that front-end processes be exec'ed for each new session. We are working on an operating system primitive that provides isolation among different user sessions within the same process with low cost.

Database-backed retrieval systems Thoth includes conduit interceptors for files, named pipes, network connections and a key-value store (memcached). In current

work, we are building a system to ensure compliance of SQL database queries with declarative policies associated with the database schema. The system can be used as a conduit interceptor, thus extending Thoth's protection to database-backed data retrieval systems.

Policy testing Assigning policies to internal conduits in Thoth, and making sure that they permit all data flows compliant with the ingress and egress policies, can be a tedious task in a large system. In current work, we are developing a tool that generates internal conduit policies semi-automatically using a system's dataflow graph and the ingress/egress policies as inputs. Moreover, the tool performs systematic testing to ensure all compliant dataflows are allowed, and helps the policy developer generate appropriate declassification policies as needed.

9 Conclusion

Efficient policy compliance in data retrieval systems is a challenging problem. Thoth is a kernel-level policy compliance layer to address this problem. The provider has the option to associate a declarative policy with each data source and sink. The policy specifies confidentiality and integrity requirements and may reflect the data owner's privacy preferences, the provider's own data-use policy, and legal requirements. Thoth enforces these policies by tracking and controlling data flows across tasks through kernel I/O interception. It prevents data leaks and corruption due to bugs and misconfigurations in application components (including misconfigurations in policies on internal conduits), as well as actions by unprivileged operators.

Our technical contributions include a declarative policy language that specifies both access (read/write) policies and how those access policies may change. The latter can be used to represent declassification policies. Additionally, the language supports content-dependent policies. Thoth uses policy sets as taint, which eliminates the need to trust application processes with access checks at the system boundary and with declassification. Our Linux-based prototype shows that Thoth can be deployed with low overhead in data retrieval systems. Among other things, this demonstrates the usefulness and viability of coarse-grained taint tracking as a basis for policy enforcement.

Acknowledgment

We would like to thank the anonymous reviewers for their helpful feedback. This research was supported in part by the European Research Council (ERC Synergy imPACT 610150) and the German Research Foundation (DFG CRC 1223).

References

[1] Adobe data breach more extensive than previoulsy disclosed. http://www.reuters.com/article/2013/10/29/us-adobe-cyberattack-idUSBRE99S1DJ20131029.

[2] Apache Lucene. http://lucene.apache.org.

[3] Apache Lucene bug report 1300. https://issues.apache.org/jira/browse/LUCENE-1300.

[4] Apache Lucene bug report 2756. https://issues.apache.org/jira/browse/LUCENE-2756.

[5] Apache Lucene bug report 3575. https://issues.apache.org/jira/browse/LUCENE-3575.

[6] Apache Lucene bug report 4511. https://issues.apache.org/jira/browse/LUCENE-4511.

[7] Apache Lucene bug report 49. https://issues.apache.org/jira/browse/LUCENE-49.

[8] Apache Lucene bug report 6503. https://issues.apache.org/jira/browse/LUCENE-6503.

[9] Apache Lucene bug report 6595. https://issues.apache.org/jira/browse/LUCENE-6595.

[10] Apache Lucene bug report 6832. https://issues.apache.org/jira/browse/LUCENE-6832.

[11] DataLossDB: Open Security Foundation. http://datalossdb.org.

[12] Memcached. http://memcached.org/.

[13] Privacy Rights Clearinghouse. http://privacyrights.org.

[14] Target breach worse than thought, states launch joint probe. http://www.reuters.com/article/2014/01/10/us-target-breach-idUSBREA090L120140110.

[15] Adam Barth, John C. Mitchell, Anupam Datta, and Sharada Sundaram. Privacy and utility in business processes. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, 2007.

[16] David A. Basin, Felix Klaedtke, and Samuel Müller. Policy monitoring in first-order temporal logic. In *Proceedings of the 22nd International Conference on Computer-Aided Verification (CAV)*, 2010.

[17] Lujo Bauer, Scott Garriss, and Michael K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 26th IEEE Symposium on Security and Privacy (S&P)*, 2005.

[18] Moritz Y. Becker, Cédric Fournet, and Andrew D. Gordon. Design and semantics of a decentralized authorization language. In *Proceedings of the 20th IEEE Computer Security Foundations Symposium (CSF)*, 2007.

[19] Michael Dalton, Christos Kozyrakis, and Nickolai Zeldovich. Nemesis: Preventing authentication & access control vulnerabilities in web applications. In *Proceedings of the 18th USENIX Security Symposium*, 2009.

[20] John DeTreville. Binder, a logic-based security language. In *Proceedings of the 23rd IEEE Symposium on Security and Privacy (S&P)*, 2002.

[21] Petros Efstathopoulos and Eddie Kohler. Manageable fine-grained information flow. In *Proceedings of the 3rd ACM SIGOPS European Conference on Computer Systems (EuroSys)*, 2008.

[22] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[23] Deepak Garg, Limin Jia, and Anupam Datta. Policy auditing over incomplete logs: theory, implementation and applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, 2011.

[24] Deepak Garg and Frank Pfenning. A proof-carrying file system. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P)*, 2010.

[25] Daniel B. Giffin, Amit Levy, Deian Stefan, David Terei, David Mazières, John Mitchell, and Alejandro Russo. Hails: Protecting data privacy in untrusted web applications. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[26] Havard D. Johansen, Eleanor Birrell, Robbert van Renesse, Fred B. Schneider, Magnus Stenhaug, and Dag Johansen. Enforcing privacy policies with meta-code. In *Proceedings of the 6th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys)*, 2015.

[27] Jayanthkumar Kannan, Petros Maniatis, and Byung-Gon Chun. Secure data preservers for web services. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, 2011.

[28] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard OS abstractions. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[29] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 1998.

[30] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the 5th Symposium on Practical Aspects of Declarative Languages*, 2003.

[31] Petros Maniatis, Devdatta Akhawe, Kevin Fall, Elaine Shi, Stephen McCamant, and Dawn Song. Do you know where your data are? secure data capsules for deployable data protection. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS)*, 2011.

[32] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, 1995.

[33] Yogesh Mundada, Anirudh Ramachandran, and Nick Feamster. Silverline: Preventing data leaks from compromised web applications. In *Proceedings of the 29th Annual Computer Security Applications Conference*, 2013.

[34] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *The 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 1999.

[35] Ben Niu and Gang Tan. Efficient user-space information flow control. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, 2013.

[36] Shayak Sen, Saikat Guha, Anupam Datta, Sriram K. Rajamani, Janice Tsai, and Jeannette M. Wing. Bootstrapping privacy compliance in big data systems. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P)*, 2014.

[37] Alan Shieh, Dan Williams, Emin Gün Sirer, and Fred B Schneider. Nexus: a new operating system for trustworthy computing. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[38] Deian Stefan, Alejandro Russo, David Mazières, and John C. Mitchell. Disjunction category labels. In *Proceedings of the 16th Nordic Conference on Information Security Technology for Applications*, 2011.

[39] Deian Stefan, Edward Z. Yang, Petr Marchenko, Alejandro Russo, Dave Herman, Brad Karp, and David Mazières. Protecting users by confining JavaScript with COWL. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[40] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Aastha Mehta, Deepak Garg, Peter Druschel, Rodrigo Rodrigues, Johannes Gehrke, and Ansley Post. Guardat: Enforcing data policies at the storage layer. In *Proceedings of the 3rd ACM SIGOPS European Conference on Computer*

Systems (EuroSys), 2015.

[41] Kevin Walsh and Fred B. Schneider. Costs of security in the PFS file system. Technical report, Computing and Information Science, Cornell University, 2012.

[42] Wikimedia Foundation. Image Dump. `http://archive.org/details/wikimedia-image-dump-2005-11`.

[43] Wikimedia Foundation. Static HTML dump. `http://dumps.wikimedia.org/`.

[44] Wikipedia. Data breach: Major incidents. `http://en.wikipedia.org/wiki/Data_breach#Major_incidents`.

[45] Edward Wobber, Martín Abadi, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. *ACM Transactions on Computer Systems (TOCS)*, 12(1), 1994.

[46] Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*, 2009.

[47] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[48] Qing Zhang, John McCullough, Justin Ma, Nabil Schear, Michael Vrable, Amin Vahdat, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Neon: System support for derived data management. In *Proceedings of the 6th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2010.

A Thoth policies for data flows in a search engine

In this Appendix we provide details of the policies used in our Thoth-compliant search engine. All policies are represented in the **read**, **update** and **declassify** rules on source conduits (documents that the search engine indexes, the user profile, etc.). We describe these rules incrementally: We start from a set of base rules, which we refine to include more policies.

Base rules Our base rules allow anyone to read, update or destroy the source conduit they are attached to.

read :- *TRUE*
update :- *TRUE*
destroy :- *TRUE*
declassify :- isAsRestrictive(**read**, `this`.**read**)
  until *FALSE*

The **read**, **update** and **destroy** rules have condition *TRUE*, which always holds, so these rules do not restrict access at all. The **declassify** rule insists that the **read** rule on any conduit containing data derived from the source conduit be at least as restrictive as the **read** rule above, which will always be the case (because the **read** rule above is the most permissive read rule possible). This base policy is pointless in itself, but it serves as the starting point for the remaining policies.

A.1 Client policies

First, we describe policies to represent client privacy preferences.

Private data policy A user Alice may wish that her private files (e.g., her e-mails) be accessible only to her. This can be enforced by requiring that accesses to Alice's private files happen in the context of a session authenticated with Alice's key. Technically, this is accomplished by replacing the conditions in the base **read**, **update** and **destroy** rules as shown below and attaching the resulting rules to Alice's private files. The predicate $\mathsf{sKeyIs}(k)$ means that the current session is authenticated using the public key k.

read :- $\mathsf{sKeyIs}(k_{\texttt{Alice}})$
update :- $\mathsf{sKeyIs}(k_{\texttt{Alice}})$
destroy :- $\mathsf{sKeyIs}(k_{\texttt{Alice}})$

The **declassify** rule remains unchanged. It ensures that any conduit containing data derived from Alice's private files is subject to a **read** rule that is at least as restrictive as the revised **read** rule above. Hence, no such conduit can be read by anyone other than Alice.

Friends only policy Alice might want that her blog and online social network profile be readable by her friends. To do this, she could add a disjunctive ("or"-separated) clause in the read rule requiring that read accesses happen in the context of a session authenticated with a key k_X of one of Alice's friends. Alice's friends are assumed to be listed in the file Alice.acl, which contains an entry of the form $\mathsf{isFriend}(k_X, X_{ACL})$ for each public key k_X that belongs to a friend of Alice. The isFriend entry also states the file X_{ACL} which lists the friends of the key k_X's owner. Note that the isFriend entry format presented in the paper was slightly simplified for readability.

read :- $\mathsf{sKeyIs}(k_{\texttt{Alice}}) \vee$
$[\mathsf{sKeyIs}(k_X) \wedge$ ("Alice.acl", *off*) says $\mathsf{isFriend}(k_X, X_{ACL})]$

The predicate $(($'Alice.acl", *off*) says $\mathsf{isFriend}(k_X, X_{ACL}))$ checks that k_X exists in the list of Alice's friends (file "Alice.acl") at some offset *off*.

Friends of friends policy To additionally allow read access to friends of friends, the policy would require read accesses to happen in the context of an authenticated session whose key is present in the friend list of any of Alice's friends.

read :- sKeyIs($k_{\texttt{Alice}}$) ∨
[sKeyIs(k_X) ∧ ("Alice.acl", *off*) says isFriend(k_X, X_{ACL})]
∨
[sKeyIs(k_Y) ∧ ("Alice.acl", off_1) says isFriend(k_X, X_{ACL})
∧ (X_{ACL}, off_2) says isFriend(k_Y, Y_{ACL})]

The predicate (('Alice.acl', off_1) says isFriend(k_X, X_{ACL})) checks that k_X exists in the list of Alice's friends (file "Alice.acl") at some offset off_1. It also binds the variable X_{ACL} to the friend list of the key k_X's owner. Next, the predicate ((X_{ACL}, off_2) says isFriend(k_Y, Y_{ACL})) checks that the public key that authenticated the session k_Y exists in the list of friends for the k_X's owner at some offset off_2.

A.2 Provider policies

Next, we describe two policies that a provider may wish to impose, possibly to comply with legal requirements.

Mandatory Access Logging (MAL) The MAL policy allows an authorized employee of the provider read access to a source conduit F if the access is logged. The log entry must have been previously written to the file k.log, where k is the public key of the employee. The log entry must mention the employee's key, the ID of the accessed conduit and the time at which the conduit is accessed with a tolerance of 60 seconds. To enforce these requirements, a new disjunctive condition is added to the last **read** rule above. The ... in the rule below abbreviate the conditions of the last **read** rule above.

read :- ... ∨
sKeyIs(k) ∧ cIdIs(F) ∧
("auth_employees", *off*) says isEmployee(k) ∧
(LOG_k = concat(k, ".log")) ∧
(LOG_k, off_1) says readLog(k, F, T) ∧ timeIs($curT$) ∧
gt($curT, T$) ∧ sub($diff, curT, T$) ∧ lt($diff$, 60)

The predicate sKeyIs(k) binds the public key that authenticated the session (i.e., the public key of the employee) to the variable k, and cIdIs(F) binds the name of source conduit to F. Next, the predicate (("auth_employees", *off*) says isEmployee(k)) checks that k exists in the list of authorized employees (file "auth_employees") at some offset *off*, to verify that the source conduit's reader is really an employee. Next, LOG_k is bound to the name of the employee's log file, k.log. The predicate ((LOG_k, off_1) says readLog(k, F, T)) checks that the log file contains an appropriate entry with some time stamp T and the remaining predicates check that the current time, $curT$, satisfies $T \leq curT \leq T + 60$s.

Every log file has a **read** rule that allows only authorized auditors to read the file (the public keys of all authorized auditors are assumed to be listed in the file "auditors"). It also has an **update** rule that allows appends only, thus ensuring that a log entry cannot be removed or overwritten.

read :- sKeyIs(k) ∧ ("auditors", *off*) says isAuditor(k)
update :- sKeyIs(k) ∧
("auth_employees", *off*) says isEmployee(k) ∧
cCurrLenIs($cLen$) ∧ cNewLenIs($nLen$) ∧
gt($nLen, cLen$) ∧ (this, 0, $cLen$) hasHash (h) ∧
(this, 0, $cLen$) willHaveHash (h)

In the append-only policy (rule **update** above), the predicate cCurrLenIs($cLen$) binds the current length of the log file to $cLen$ and the predicate cNewLenIs($nLen$) binds the new length of the log file to $nLen$. Next, the predicate gt($nLen, cLen$) ensures that the update only increases the log file's length. *(c, off, len)* hasHash (or willHaveHash) is a special mode of using says (or willsay) which allows the policy interpreter to refer to the hash of the conduit c's content (or updated content in a write transaction) from offset *off* with length *len*. In the **update** rule, hasHash and willHaveHash are used to verify that the existing file content is not modified during an update by checking that the hashes of the file from offset 0 to $cLen$, originally and after the prospective update, are equal.

A more efficient implementation of the append-only policy could rely on a specialized predicate unmodified(*off*, *len*), which checks that the conduit contents from offset *off* with length *len* were not modified. The **update** rule could then be simplified to:

update :- sKeyIs(k) ∧
("auth_employees", *off*) says isEmployee(k) ∧
cCurrLenIs($cLen$) ∧ cNewLenIs($nLen$) ∧
gt($nLen, cLen$) ∧ unmodified(0, $cLen$)

Region-based censorship Legal requirements may force the provider to blacklist certain source files in certain regions. Accordingly, the goal of the censorship policy is to ensure that content from a document F can only reach users in regions whose blacklists do not contain F. The policy relies on a mapping from IP addresses to regions and a per-region blacklist file. The blacklist file is maintained in a sorted order to efficiently lookup whether it contains a given document or not.

The censorship policy is expressed by modifying the **declassify** rule of every source conduit cndID as follows:

declassify :- isAsRestrictive(**read**, `this`.**read**) until
(CENSOR(cndID) ∧ isAsRestrictive(**read**, `this`.**read**))

The rule says that the **read** rule on any conduit to which cndID flows must be as restrictive as cndID's **read** rule *until* a conduit at which the condition CENSOR(cndID) holds is reached. CENSOR(cndID) is a macro defined below. The predicate sIpIs(IP) checks

that the IP address of the connecting (remote) party is *IP* and the predicate $\mathsf{IpPrefix}(IP,R)$ means that *IP* belongs to region *R*. The blacklist file for region *R* is *R*.BlackList. In words, CENSOR(cndID) means that the remote party's *IP* belongs to a region *R* and cndID lies strictly between two two consecutive entries in *R*'s blacklist file (and, hence, cndID does not exist in *R*'s blacklist file).

$$\begin{array}{l}\mathsf{sIpIs}(IP) \wedge \mathsf{IpPrefix}(IP,R) \wedge \\ (F_{BL} = \mathsf{concat}(R, \text{".BlackList"})) \wedge \\ (F_{BL}, off_1)\ \mathsf{says}\ \mathsf{isCensored}(cnd_1) \wedge \\ \mathsf{add}(off_2, off_1, \mathsf{CENSOR\_ENTRY\_LEN}) \wedge \\ (F_{BL}, off_2)\ \mathsf{says}\ \mathsf{isCensored}(cnd_2) \wedge \\ \mathsf{lt}(cnd_1, \mathsf{cndID}) \wedge \mathsf{lt}(\mathsf{cndID}, cnd_2)\end{array}$$

A.3 Search engine flows

Indexing flow The indexer reads documents with possibly contradictory policies and, in the absence of a dedicated provision for declassification, the index (and any documents derived from it) cannot be served to any client. To prevent this problem, searchable documents allow typed declassification. The **declassify** rule for each searchable document is modified with a new clause that allows complete declassification into an (internal) conduit whose **update** rule allows the conduit to contain only a list of object ids. The modified **declassify** rule of each source document has the form:

$$\begin{array}{l}\textbf{declassify}\ \text{:-}\ \ldots\ \mathsf{until}\ (\ldots \vee (\mathsf{clsIntrinsic} \wedge \\ \quad \mathsf{isAsRestrictive}(\textbf{update}, \mathsf{ONLY\_CND\_IDS})))\end{array}$$

The macro ONLY_CND_IDS stipulates that only a list of valid conduit ids can be written and it expands to:

$$\begin{array}{l}\mathsf{cCurrLenIs}(\mathtt{cLen}) \wedge \mathsf{cNewLenIs}(\mathtt{nLen}) \wedge \\ \mathsf{each\ in}(\mathsf{this}, \mathtt{cLen}, \mathtt{nLen})\ \mathsf{says}(\mathtt{cndId}) \\ \quad \{\mathsf{cIdExists}(\mathtt{cndId})\}\end{array}$$

In the macro above, the predicate $\mathsf{cNewLenIs}(nLen)$ binds the new length of the output file to *nLen*. The predicate willsay checks that the content update from offset 0 and length *nLen* is a list of conduit IDs, and the predicate $\mathsf{cIdExists}(cndId)$ checks that *cndId* corresponds to an existing conduit.

So far we have assumed that the conduit ids are not themselves confidential. If the presence or absence of a particular conduit id in the search results may leak sensitive information, then the source declassification policy can be augmented to require that the list of conduit ids is accessible only to a principal who satisfies the confidentiality policies of all listed conduits. Then, the macro ONLY_CND_IDS can be re-written to:

$$\begin{array}{l}\mathsf{cCurrLenIs}(\mathtt{cLen}) \wedge \mathsf{cNewLenIs}(\mathtt{nLen}) \wedge \\ \mathsf{each\ in}(\mathsf{this}, \mathtt{cLen}, \mathtt{nLen})\ \mathsf{willsay}(\mathtt{cndId}) \\ \{\mathsf{cIdExists}(\mathtt{cndId}) \wedge \mathsf{hasPol}(\mathtt{cndId}, \mathtt{P}) \wedge \\ \ \mathsf{isAsRestrictive}(\textbf{read}, \mathtt{P}.\textbf{read}) \wedge \\ \ \mathsf{isAsRestrictive}(\textbf{declassify}, \mathtt{P}.\textbf{declassify})\}\end{array}$$

Additionally in the macro above, the predicate $\mathsf{hasPol}(\mathtt{cndId}, \mathtt{P})$ binds P to the policy of the conduit cndId, and the predicate $\mathsf{isAsRestrictive}(\textbf{read}, \mathtt{P}.\textbf{read})$ requires that the confidentiality of the list of conduit ids is as restrictive as the confidentiality requirements of the source conduit ids themselves.

Profile aggregation flow Since raw user activity logs are typically private, a declassification is required that enables a profile generator to produce a user preferences vector (a vector of fixed length) from the activity logs. However, this preferences vector must further be restricted so that it can be used to produce only a list of conduit ids (the search results). Further, the user might also want to ensure that only activity logs generated in the past 48 hours be used for personalization. This can be achieved by allowing the declassification into the fixed-size vector to happen only within 172800 seconds of the log's creation. Suppose an activity log is created at time *t* and that the preferences vector has length *n*. Then, the relevant policy rules on the activity log are the following (note that *t* and *n* are constants, not variables).

$$\begin{array}{l}\textbf{read}\ \text{:-}\ \mathsf{sKeyIs}(k_{\mathtt{Alice}}) \\ \textbf{declassify}\ \text{:-}\ [\mathsf{isAsRestrictive}(\textbf{read}, \mathtt{this}.\textbf{read})\ \mathsf{until} \\ \quad \mathsf{isAsRestrictive}(update, \mathsf{ONLY\_FLOATS}(n)) \wedge \\ \quad \mathsf{clsIntrinsic} \wedge \mathsf{timeIs}(curT) \wedge \mathsf{gt}(curT, t) \wedge \\ \quad \mathsf{sub}(diff, curT, t) \wedge \mathsf{lt}(diff, 172800)] \wedge \\ \quad [\mathsf{isAsRestrictive}(\textbf{read}, \mathtt{this}.\textbf{read})\ \mathsf{until}\ \mathsf{clsIntrinsic} \wedge \\ \quad \mathsf{isAsRestrictive}(\textbf{update}, \mathsf{ONLY\_CND\_IDS})]\end{array}$$

This policy ensures that the raw user logs can only be transformed into the user preferences vector, which in turn can only be declassified into the search results of the search engine.

The macro $\mathsf{ONLY\_FLOATS}(n)$ stipulates that only a vector of *n* floats can be written. It expands to:

$$\begin{array}{l}\mathsf{cNewLenIs}(\mathtt{nLen}) \wedge \\ \mathsf{each\ in}(\mathsf{this}, 0, \mathtt{nLen})\ \mathsf{willsay}(\mathtt{value}) \\ \{\mathsf{vType}(\mathtt{value}, \mathsf{FLOAT}) \wedge (\mathtt{Cnt}++)\} \wedge \\ \mathsf{eq}(\mathtt{Cnt}, n)\end{array}$$

In the macro above, the predicate $\mathsf{cNewLenIs}(nLen)$ binds the new length of the output file to *nLen*. The predicate willsay checks that the content update from offset 0 and length *nLen* is a list of *value*s, and the predicate $\mathsf{vType}(value, \mathsf{FLOAT})$ checks that each *value* in the list is of type FLOAT. The predicate $\mathsf{eq}(cnt, n)$ checks that the update contains *n* floats.