2014 IEEE Symposium on Security and Privacy

Bootstrapping Privacy Compliance in Big Data Systems

Shayak Sen*, Saikat Guha†, Anupam Datta*, Sriram K. Rajamani†, Janice Tsai‡ and Jeannette M. Wing‡
*Carnegie Mellon University, Pittsburgh, USA
{shayaks,danupam}@cmu.edu
†Microsoft Research, Bangalore, India
{saikat,sriram}@microsoft.com
‡Microsoft Research, Redmond, USA
{jatsai,wing}@microsoft.com

***Abstract*—With the rapid increase in cloud services collecting and using user data to offer personalized experiences, ensuring that these services comply with their privacy policies has become a business imperative for building user trust. However, most compliance efforts in industry today rely on manual review processes and audits designed to safeguard user data, and therefore are resource intensive and lack coverage. In this paper, we present our experience building and operating a system to automate privacy policy compliance checking in Bing. Central to the design of the system are (a) LEGALEASE—a language that allows specification of privacy policies that impose restrictions on how user data is handled; and (b) GROK—a data inventory for Map-Reduce-like big data systems that tracks how user data flows among programs. GROK maps code-level schema elements to datatypes in LEGALEASE, in essence, annotating existing programs with information flow types with minimal human input. Compliance checking is thus reduced to information flow analysis of big data systems. The system, bootstrapped by a small team, checks compliance daily of millions of lines of ever-changing source code written by several thousand developers.**

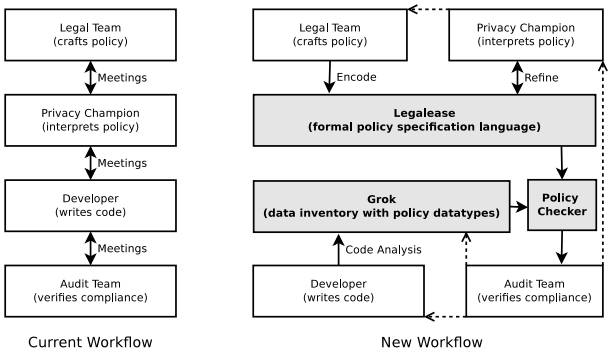


Fig. 1. Privacy Compliance Workflow. Manual reviews and audits are highly time-consuming, and resource-intensive. By encoding policy in LEGALEASE using the GROK data inventory, we decouple interactions so policy specification, interpretation, product development, and continuous auditing can proceed in parallel.

I. INTRODUCTION

Web services companies, such as Facebook, Google, and Microsoft, that use personal information of users for various functions are expected to comply with their declared privacy policies. Companies in the US are legally required to disclose their data collection and use practices, and the US Federal Trade Commission (FTC) has the mandate to enforce compliance, which it exercises by imposing penalties on companies found to violate their own stated policies [1], [2], [3]. In practice, these legal requirements translate to companies creating review processes and conducting internal audits to ensure compliance [4], [5]. Manual reviews and audits are time-consuming, resource-intensive, lack coverage, and, thus, inherently do not scale well in large companies; indeed, there have been cases where internal processes have not caught policy violations [6]. In this paper we take the first steps toward automated checking of large-scale Map-Reduce-like big data systems for compliance with privacy policies that restrict how various types of personal information flow through these systems. Our deployed prototype reduces compliance checking time and improves coverage by orders of magnitude, across the data analytics pipeline of Bing. Further, the human resources needs are small — the prototype is run by a small team, and scales to the needs of several thousands of developers working on Bing.

To contextualize the challenges in performing automated privacy compliance checking in a large company with tens of thousands of employees, it is useful to understand the division of labor and responsibilities in current compliance workflows [4], [5]. Privacy policies are typically crafted by lawyers in a corporate legal team to adhere to all applicable laws and regulations worldwide. Due to the rapid change in product features and internal processes, these policies are necessarily specified using high-level policy concepts that may not cleanly map to the products that are expected to comply with them. For instance, a policy may refer to "IP Address" which is a high-level policy concept, and the product may have thousands of data stores where data derived from the "IP Address" is stored (and called with different names) and several thousand processes that produce and consume this data, all of which have to comply with policy. The task of interpreting the policy as applicable to individual products then falls to the tens of privacy champions embedded in product groups. Privacy champions review product features at various stages of the development process, offering specific requirements to the development teams to ensure compliance

DOI 10.1109/SP.2014.28

with policy. The code produced by the development team is expected to adhere to these requirements. Periodically, the compliance team audits development teams to ensure that the requirements are met.

We illustrate this process with a running example we use throughout this paper. Let us assume that we are interested in checking compliance for an illustrative policy clause that promises "full IP address will not be used for advertising.". The privacy champion reviewing the algorithm design for, say online advertisement auctions, may learn in a meeting with the development team that they use the IP address to infer the user's location, which is used as a bid-modifier in the auction. The privacy champion may point out that this program is not compliant with the above policy and suggest to the development team to truncate the IP address by dropping the last octet to comply with the policy, without significantly degrading the accuracy of the location inference. The development team then modifies the code to truncate the IP address. Periodically, the audit team may ask the development team whether the truncation code is still in place. Later, the advertising abuse detection team may need to use the IP address. This may result in a policy exception, but may come with a different set of restrictions, e.g., "IP address may be used for detecting abuse. In such cases it will not be combined with account information." The entire process (Fig. 1 left panel) is highly manual, with each step sometimes taking weeks to identify the right people to talk to and multiple meetings between different groups (lawyers, champions, developers, auditors) that may as well be communicating in different languages.

Our central contribution is a workflow for privacy compliance in big data systems. Specifically, we target privacy compliance of large codebases written in languages that support the Map-Reduce programming model [7], [8], [9]. This focus enables us to apply our workflow to current industrial-scale data processing applications, in particular the data analytics backend of Bing, Microsoft's web search engine [10]. This workflow leverages our three key technical contributions: (1) a language LEGALEASE for stating privacy policies, which is usable by policy authors and privacy policy champions, but has precise semantics and enables automated checking for compliance, (2) a self-bootstrapping data inventory mapper GROK, which maps low level data types in code to high-level policy concepts, and bridges the world of product development with the world of policy makers, and (3) a scalable implementation of automated compliance checking for Bing. We describe each of these parts below.

The LEGALEASE **language**. LEGALEASE is an usable, expressive, and enforceable privacy policy language. The primary design criteria for this language were that it (a) be *usable* by the policy authors and privacy champions; (b) be *expressive* enough to capture real privacy policies of industrial-scale systems, e.g., Bing; (c) and should allow *compositional reasoning* on policies.

As the intended users for LEGALEASE are policy authors and privacy champions with limited training in formal languages, enabling usability is essential. To this end, LEGALEASE enforces syntactic restrictions ensuring that encoded policy clauses are structured very similarly to policy texts. Specifically, building on prior work on a first order privacy logic [11], policy clauses in LEGALEASE allow (resp. deny) certain types of information flows and are refined through exceptions that deny (resp. allow) some sub-types of the governed information flow types. This structure of nested allow-deny rules appears in many practical privacy policies, including privacy laws such the Health Insurance Portability and Accountability Act (HIPAA) and the Gramm-Leach-Bliley Act (GLBA) (as observed in prior work [11]), as well as privacy policies for Bing and Google. A distinctive feature of LEGALEASE (and a point of contrast from prior work based on first-order logic and first order-temporal logic [12], [11]) is that the semantics of policies is compositional: reasoning about a policy is reduced to reasoning about its parts. This form of compositionality is useful because the effect of adding a new clause to a complex policy is locally contained (an exception only refines its immediately enclosing policy clause). Section III presents the detailed design of the language. To validate the usability of LEGALEASE by its intended users, we conduct a user study among policy writers and privacy champions within Microsoft. On the other hand, by encoding Bing and Google's privacy policies regarding data usage on their servers, we demonstrate that LEGALEASE retains enough expressiveness to capture real privacy policies of industrial-scale systems. Section VI presents the results of the usability study and the encoding of Bing and Google's privacy policies.

The GROK **mapper**. GROK is a data-inventory for Map-Reduce-like big data systems. It maps every dynamic schema-element (e.g., members of a tuple passed between mappers and reducers) to datatypes in LEGALEASE. This inventory can be viewed as a mechanism for annotating existing programs written in languages like Hive [7], Dremel [8], or Scope [9] with the information flow types (datatypes) in LEGALEASE. Our primary design criteria for this inventory were that it (a) be *bootstrapped* with minimal developer effort; (b) reflect *exhaustive and up-to-date* information about all data in the Map-Reduce-like system; and (c) make it easy to *verify* (and update) the mapping from schema-elements to LEGALEASE datatypes. The inventory mappings combine information from a number of different sources each of which has its own characteristic coverage and quality. For instance, syntactic analysis of source code (e.g., applying pattern-matching to column names) has high coverage but low confidence, whereas explicit annotations added by developers has high confidence but low coverage. Section IV details the design of the system, and Section V presents how the automated policy checker performs conservative analysis while minimizing false positives over imperfect mappings.

By using automated data-inventory mapping and adding precise semantics to the policy specification, we reduce time-consuming meetings by decoupling the interactions between the various groups so policy specification, policy interpreta-

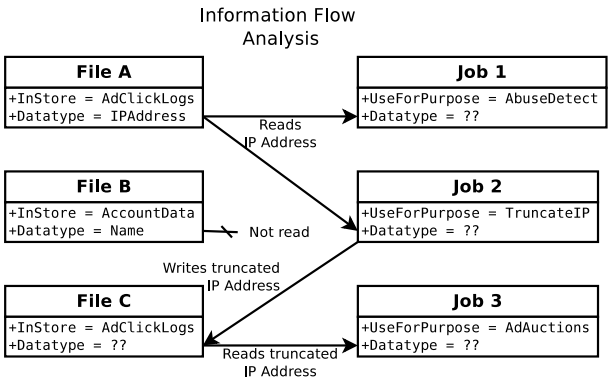


Fig. 2. Example scenario showing a partially-labeled data dependency graph between three files and programs.

tion, product development, and continuous auditing can proceed in parallel. Since we use automation to bridge code-level details to policy concepts, meetings are needed only when our automated privacy compliance checker (conservatively) detects potentially sensitive scenarios, and are hence more focused, especially on actionable items (dotted lines in Fig. 1).

Scale. Our scalability criteria are in (a) the amount of data over which we perform automated privacy compliance checking; (b) the time we take to do so; and (c) the number of people resources needed for the entire effort. As we quantify in Section VI, our deployed system scales to tens of millions of lines of source code written by several thousand developers storing data in tens of millions of files containing over hundred million schema-elements, of which a substantial fraction is changing or added on a day-to-day basis. Our data inventory takes twenty minutes (daily), and evaluating the complete LEGALEASE encoding of Bing's privacy policy over the entire data takes ten minutes. The entire effort was bootstrapped from scratch by a team of five people.

II. MOTIVATING EXAMPLE

In this section, we use an example to highlight salient features of our programming model and typical privacy policies that these programs have to respect. These features motivate the design of our privacy policy language LEGALEASE described in Section III, the data inventory GROK described in Section IV, and provide intuition on how privacy compliance checking is reduced to a form of information flow analysis.

Consider the scenario in Fig. 2. There are three programs (Jobs 1, 2, 3) and three files (Files A, B, C). Let us assume that the programs are expected to be compliant with a privacy policy that says: "full IP address will not be used for advertising. IP address may be used for detecting abuse. In such cases it will not be combined with account information." Note that the policy restricts how a certain type of personal information flows through the system. The restriction in this example is based on purpose. Other common restrictions include storage restrictions (e.g., requiring that certain types of user data are not stored together) and, for internal policies, role-based restrictions (e.g., requiring that only specific product team members should use certain types of user data). While our policy language is designed in a general form enabling domain-specific instantiations with different kinds of restrictions, our evaluation of Bing is done with an instantiation that has exactly these three restrictions—purpose, role, and storage—on flow of various types of personal information. We interpret information flow in the sense of non-interference [13], i.e., data not supposed to flow to a program should not affect the output of the program.

The data dependence graph depicted for the example in Fig. 2 provides a useful starting point to conduct the information flow analysis. Nodes in the graph are data stores, processes, and humans. Directed edges represent data flowing from one node to another. To begin, let us assume that programs are labeled with their purpose. For example, Job 1 is for the purpose of AbuseDetect. Furthermore, let us also assume that the source data files are labeled with the type of data they hold. For example, File A holds data of type IPAddress. Given these labels, additional labels can be computed using a simple static dataflow analysis. For example, Job 1 and Job 2 both acquire the datatype label IPAddress since they read File A; File C (and hence Job 3) acquires the datatype label IPAddress:Truncated. Given a labeled data dependence graph, a conservative way of checking non-interference is to check whether there exists a path from restricted data to the program in the data dependence graph. In a programming language such as C or Java, this approach may lead to unmanagable overtainting. Fortunately, the data analytics programs we analyze are written in a restricted programming model without global state and with very limited control flow based on data. Therefore, we follow precisely this approach. Languages like Hive [7], Dremel [8], or Scope [9] that are used to write big data pipelines in enterprises adhere to this programming model (Section IV provides additional details). Note, that for the search engine that we analyze, the data dependence graph does not come with these kinds of labels. Bootstrapping these labels without significant human effort is a central challenge addressed by GROK (Section IV).

III. POLICY SPECIFICATION LANGUAGE

We present the design goals for LEGALEASE, the language syntax and formal semantics, as well a set of illustrative policy examples.

A. Design Goals

As mentioned, we intend legal teams and privacy champions to encode policy in LEGALEASE. Therefore, our primary goal is usability by individuals with typically no training in first-order or temporal logic, while being sufficiently expressive for encoding current policies.

a) Usability: Policy clauses in LEGALEASE are structured very similarly to clauses in the English language policy. This correspondence is important because no single individual in a large company is responsible for all policy clauses; different sub-teams own different portions of the policy and

Policy Clause C	::=	$D \mid A$
Deny Clause D	::=	DENY $T_1 \cdots T_n$ EXCEPT $A_1 \cdots A_m$
		| DENY $T_1 \cdots T_n$
Allow Clause A	::=	ALLOW $T_1 \cdots T_n$ EXCEPT $D_1 \cdots D_m$
		| ALLOW $T_1 \cdots T_n$
Attribute T	::=	⟨attribute-name⟩ $v_1 \cdots v_l$
Value v	::=	⟨attribute-value⟩

TABLE I
GRAMMAR FOR LEGALEASE

any mapping from LEGALEASE clauses to English clauses that do not fall along these organizational bounds would necessitate (time-consuming) processes to review and update the LEGALEASE clauses. By designing in a 1-1 correspondence to policies in English, LEGALEASE clauses can be added, reviewed, and updated at the same time as the corresponding English clauses and by the same individuals.

b) Expressivity: LEGALEASE clauses are built around an attribute abstraction (described below) that allows the language to evolve as policy evolves. For instance, policies today tend to focus on access control, retention times, and segregation of data in storage, [14], [15], [16]. However, information flow properties [17] provide more meaningful restrictions on information use. Similarly, the externally-visible policy may be at a higher level while the internal policy may be more restrictive and nuanced. LEGALEASE allows transitioning between these policies with minimal policy authoring overhead, and provides enforcement techniques so as to enable stronger public-facing policy promises.

c) Compositional Reasoning: When the whole policy is stated as a monolithic logic formula, it may be more difficult to naturally reason about the effects of the policy, due to unexpected interactions between different parts of the formula [18]. LEGALEASE provides meaningful syntactic restrictions to allow compositional reasoning where the result of checking the whole policy is a function of reasoning on its parts.

B. LEGALEASE *Language Syntax*

A LEGALEASE policy (Table I) is rooted in a single top-level policy clause. A *policy clause* is a layered collection of (alternating) ALLOW and DENY clauses where each clause relaxes or constricts the enclosing clause (i.e., each layer defines an exception to the enclosing layer). Each clause contains a set of domain-specific attributes that restrict to which data dependency graph nodes the policy applies. Attributes are specified by their name, and one or more values. Attribute values are picked from a *concept lattice* [19] for that attribute (explained below). The absence of an attribute implies that there no restrictions for that attribute. The policy author defines new attributes by providing an attribute name and a lattice of values. In III-D, we describe the particular instantiation of attributes we use to specify information flow restrictions on programs.

Checking: LEGALEASE policies are checked at each node in the data dependency graph. Each graph node is labeled with the domain-specific attribute name and set of lattice values. For instance, in our setting, we assume that the data inventory phase labels programs with data that flows to it, a purpose attribute and a user attribute. Informally, an ALLOW clause permits graph nodes labeled with any subset of the attribute values listed in the clause, and a DENY clause forbids graph nodes labeled with any set that overlaps with the attribute values in the clause. The layering of clauses determines the context within which each clause is checked. We define the formal evaluation semantics in Section III-E.

C. LEGALEASE*, by example*

We illustrate LEGALEASE through a series of examples that build up to a complex clause. In the examples we use two user-defined attributes: *DataType* and *UseForPurpose* (our deployment uses two additional ones *AccessByRole* and *InStore*). We define the concept lattice for each of these four attributes in the next subsection.

The simplest LEGALEASE policy is DENY. The policy contains a single clause; the clause contains no exceptions and no attribute restrictions. The policy, rather uninterestingly, simply denies everything. We next add a restriction along the *DataType* attribute for graph nodes to which IP address flows.

```
DENY DataType IPAddress
```

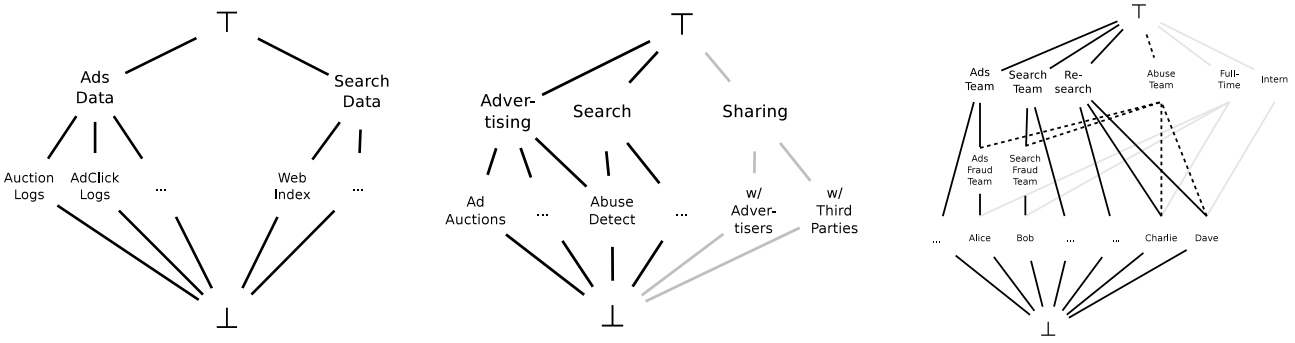
As discussed in our running example, there is often a need to capture some limited form of history of the data flow (e.g., that the IP address has been truncated before it can be used). We capture this notion of typestate in the concept lattice for the *DataType* attribute (described below). The lattice contains an element IPAddress:Truncated meant to represent the truncated IP address, and the lattice element for IP address IPAddress, such that IPAddress:Truncated $\leq$ IPaddress, where $\leq$ is the partial order for the lattice. We next add the exception that allows us to use the truncated IP address. The added lines are marked with ◁.

```
DENY DataType IPAddress
EXCEPT                                          ◁
  ALLOW DataType IPAddress:Truncated            ◁
```

The above policy contains a clause with an exception. The first disallows any use of IP address, while the exception relaxes the first allowing use when the IP address is truncated. Next, we restrict the policy to advertising uses only by adding a restriction along the *UseForPurpose* attribute for the value Advertising, while retaining the exception that allows the use of IP Address when truncated.

```
DENY DataType IPAddress
     UseForPurpose Advertising                  ◁
EXCEPT
  ALLOW DataType IPAddress:Truncated
```

The above policy corresponds to the English clause "full IP address will not be used for advertising". Note that since the first clause is restricted only to advertising use, and the second rule does not relax that attribute, the net effect is that the clause applies only to use of IP address for advertising and says nothing about non-advertising uses (consistent with the English clause).



(a) *InStore* lattice (b) *UseForPurpose* lattice (c) *AccessByRole* lattice

Fig. 3. Concept lattice [19] examples for three of the domain-specific attributes we define.

Attribute Name	Concept	Defined In	Example Lattice
InStore	Storage	Section III-D1	Fig. 3a
UseForPurpose	Use for purpose	Section III-D2	Fig. 3b
AccessByRole	Access control	Section III-D3	Fig. 3c
DataType	Information	Section III-D4	Fig. 4c

TABLE II
ATTRIBUTES USED IN OUR DEPLOYMENT

Finally, consider the English policy "full IP address will not be used for advertising. IP address may be used for detecting abuse. In such cases it will not be combined with account information." This policy is encoded in LEGALEASE below. The first, second, and third sentences correspond respectively to lines 1–4, 5–6, and 7–8.

```
DENY DataType IPAddress
    UseForPurpose Advertising
EXCEPT
  ALLOW DataType IPAddress:Truncated
  ALLOW DataType IPAddress                    ◁
      UseForPurpose AbuseDetect               ◁
  EXCEPT                                      ◁
    DENY DataType IPAddress, AccountInfo      ◁
```

The last clause (in lines 7-8) mentions that the combination of IPAddress and AccountInfo is denied, but these elements can be used individually. It turns out that giving formal semantics to such exceptions where combinations are disallowed whereas individual elements are allowed is non-trivial. We revisit this issue when we give formal semantics to LEGALEASE.

In Section VI, we encode the entirety of the externally-visible privacy policies for Bing and Google, as it pertains to backend data processing and storage. Therefore, LEGALEASE satisfies our goal of being able to express current policy in a way that there is a natural 1-1 correspondence with the policy. We also show through a user study that LEGALEASE is easy for privacy champions to learn and use.

D. Domain-Specific Attributes

LEGALEASE derives its expressiveness and extensibility through domain-specific attributes that can be suitably instantiated to specify policy restrictions specific to the application at hand. The attribute values must be organized as a *concept lattice* [19], which is a complete lattice[1] of values that the attribute can take.

The concept lattice serves three purposes in LEGALEASE: first, it abstracts away semantics (e.g., policy datatype and typestate, use for purpose, access control) in a way that the rest of LEGALEASE and language tools do not have to be modified as long as a new concept can be encoded as a lattice, and the GROK data mapper can label nodes in the data dependency graph with the label corresponding to the semantic meaning of that attribute. In practice we have found the need for only four concepts (Table II), all of which can be encoded as a lattice and labeled by the GROK mapper. Second, the lattice structure allows the user to concisely define (and refer to) sets of elements through their least upper bound. Finally, the lattice structure allows us to statically check the policy for certain classes of errors (e.g., exceptions that have no effect as they do not relax or constrict the enclosing clause).

LEGALEASE does not assign any particular semantic meaning to these attributes. In our context, however, we instantiate the language to specify restrictions on information flow in our programming model. In particular, the policy datatype labels denote the information that flows to a particular node in the data dependency graph, and the typestate labels record a limited history of declassification actions during that flow.

We now define the four domain-specific attributes and their corresponding concept lattices that we use in our deployment.

1) InStore attribute: We define the *InStore* attribute to encode certain policies around collection and storage of data. For instance, consider the policy "Ads will not store full IP address". A privacy champion may interpret that policy as forbidding storing the entire IP address on any data store designated for Ads. The concept lattice, illustrated in Fig. 3a, contains all data stores in the bottom half (e.g., AuctionLogs,

[1]Recall, a complete lattice is a structure of the form $(L, \leq, \wedge, \vee, \top, \bot)$ where L is a partially ordered set of elements under the ordering relation $\leq$, meet ($\wedge$) and join ($\vee$) are operators that result respectively in the least upper bound and the greatest lower bound, and top ($\top$) and bottom ($\bot$) are the maximal and minimal element.

AdClickLogs, WebIndex) and a coarser grained classification in the top half (e.g., AdsData, SearchData). The policy-writer can use any lattice element in LEGALEASE clauses. As we describe in the formal semantics section, the policy then applies to all elements below the mentioned element. Thus by choosing the AdsData element the policy-writer would cover all data stores designated for Ads.

The *InStore* attribute adds a storage restriction in the policy clause as follows:

DENY *DataType* IPAddress
  InStore AdsData ◁
EXCEPT
  ALLOW *DataType* IPAddress:Truncated

The policy above disallows storing IP addresses in any data store designated as Ads unless it has been truncated. The GROK mapper labels all data stores (e.g., files) with the *InStore* attribute value (e.g., AdClickLogs) so that the above clause is triggered whenever the non-truncated IP address flows to a file designated for storing advertising data.

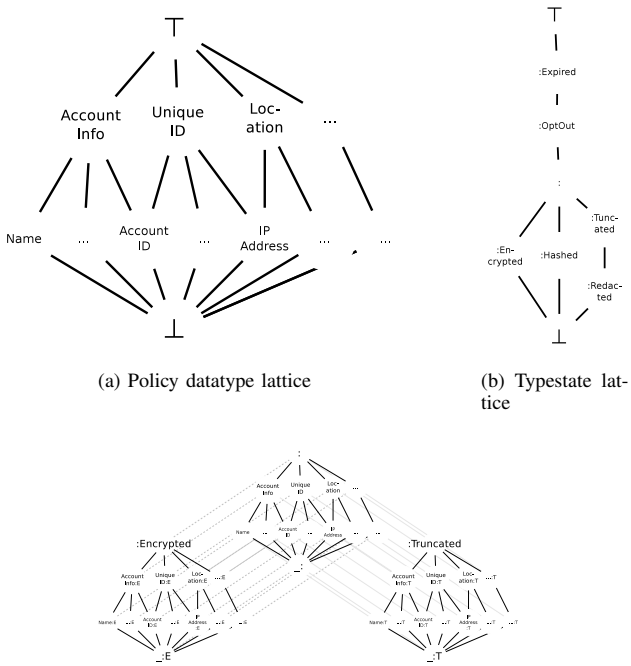
In LEGALEASE, a use-oriented policy is expressed very similarly by changing *InStore* AdsData in the policy above to *UseForPurpose* Advertising. With this low editing overhead, LEGALEASE satisfies our expressivity design goal of enabling seamless transitions towards meaningful restrictions on information use.

2) UseForPurpose Attribute: In earlier examples we show how the *UseForPurpose* attribute helps encode certain policies around use of data. In examples so far we have discussed using data in various products (Advertising) or product features (AbuseDetect). The use, however, need not be restricted purely to products. Indeed, policy often seeks to disallow sharing certain data with third-parties; here Sharing is another use of the data. All these uses are captured in the concept lattice for the *UseForPurpose* attribute (Fig. 3b).

3) AccessByRole Attribute: Internal policies often further restrict data use based on which team is accessing the data. These access-control oriented roles often do not show up in externally-facing policies because they make sense only within the organizational structure. To encode internal access-control based policies we define the *AccessByRole* attribute where the lattice (Fig. 3c) is the role-based hierarchy which includes the org-reporting hierarchy (solid black lines in the figure), virtual-teams that may span different parts of the organization (dotted lines), as well as job-titles (gray lines). The lattice is populated from the organizational directory service. An example internal policy may be:

DENY *DataType* IPAddress
EXCEPT
  ALLOW *AccessByRole* AbuseTeam ◁
  EXCEPT ◁
    DENY *AccessByRole* Intern ◁

The above policy allows the abuse detection team to use the IP address, which from the lattice in Fig. 3c includes a team in the ads organization, one in the search organization, and some individuals in the research organization. The policy then explicitly denies access to interns. Note that the layered specification (explicitly) breaks ties — both the allow and the nested deny clause apply to data dependency nodes labeled with *AccessByRole* Dave, but the more deeply nested DENY clause takes precedence. We show in the formal semantics section how all LEGALEASE policies can be interpreted unambiguously.



(a) Policy datatype lattice

(b) Typestate lattice

(c) *DataType* lattice (product of 4a and 4b); only part of lattice shown.

Fig. 4. Concept lattice construction for the *DataType* attribute.

4) DataType Attribute: Lastly, we define the *DataType* attribute and concept lattice. The interesting aspect is the notion of increasing or decreasing the sensitiveness of a datatype (e.g., encryption decreases sensitiveness, opting-out increases). Since policy may disallow use of data at one sensitiveness level and allow use at another, there is a need to track a limited history of the policy datatype. We track history with a notion we call typestate (defined below).

a) Policy datatypes: The policy datatypes are organized in a lattice (Fig. 4a). For example, IP address is both a unique identifier as well as a form of location information. The ordering relationship ($\leq_T$) defines $t \leq_T t'$ if t "is of type" t'. e.g., IPAddress $\leq_T$ UniqueID and IPAddress $\leq_T$ Location.

b) Limited typestate: The typestate is a limited way of tracking history. Consider the typestate :OptOut, which we use to refer to data from users that have opted-out of certain products, or :Expired that tracks data past its retention time and scheduled for deletion (highly sensitive). The GROK mapper determines the typestate of nodes in the data dependency graph as defined in Section V. Fig. 4b lists some other typestates that we use in our deployment. These typestates are organized in a lattice ordered by the "is less sensitive than" relation ($\leq_S$);

the sensitiveness levels are decided by privacy champions.

c) Combining policy datatypes and typestates: The concept lattice (D) for the *DataType* attribute is defined over tuples of the form $t{:}s$ where t is picked from the lattice of policy datatypes, and s is picked from the lattice of typestates. The ordering relationship ($\leq$) for set element $t{:}s \in D$ is defined as $t{:}s \leq t'{:}s'$ iff $t \leq_T t' \wedge s \leq_S s'$. Intuitively it is the lattice formed by flattening the result of replacing each element of one lattice with a copy of the other lattice. Fig. 4c shows a part of the *DataType* lattice formed by replacing the :, :Encrypted, and :Truncated elements from the typestate lattice with the policy datatype lattice; black lines encode the $\leq_T$ ordering relationship, and grey lines encode $\leq_S$.

E. Formal Semantics

Our third goal is to enable compositional reasoning of policies. We now present the formal semantics of LEGALEASE that satisfies this goal.

The semantics uses vectors of attributes. We use the notation T (with suitable superscripts) to denote a vector of sets of lattice elements (representing the label of a node in the data dependency graph or a clause during policy evaluation), and the notation T_x to denote the value of attribute x in T, which is a set of lattice elements. Recall that LEGALEASE policies are checked at each graph node. Each graph node G is labeled with a vector T^G. Similarly, policy clauses contain a vector T^C.

In order to define how policies are checked, we define the partial order $\sqsubseteq$ over vectors of sets of lattice elements, as pointwise ordering over all the attributes in the vector. More precisely, we define $T \sqsubseteq T'$ iff $\forall x.T_x \sqsubseteq_x T'_x$, where $\sqsubseteq_x$ is defined as $\forall_{v \in T_x} \exists_{v' \in T'_x}.v \leq_x v'$ and $\leq_x$ is the partial order associated with the attribute x. Using DeMorgan's law, we have that $T \not\sqsubseteq T'$ iff $\exists x.T_x \not\sqsubseteq_x T'_x$. Intuitively, a policy clause ALLOW T^C applies to a graph node labeled with a vector T^G if $T^G \sqsubseteq T^C$.

We also define $T \sqcap T'$ pointwise, where for each x, $T_x \sqcap_x T'_x$ is $\{\bigvee_{v \in T_x} v \wedge_x v' | v' \in T'_x\}$. We use the notation $\bot \in T$ as shorthand for $\exists x. \bot \in T_x$. A policy clause DENY T^C applies to a graph node labeled with a vector T^G if $\bot \notin T^G \sqcap T^C$, which intuitively means that the overlap between the denied elements T^G and the node labels T^C is not empty.

Missing attributes: If for some attribute x the set of lattice elements is not specified, it is taken to be a singleton set with top ($\top$), as missing attributes imply all.

Finally, the judgment C allows T^G, read as policy clause C allows data dependency graph node labeled with attribute set T^G, characterizes which nodes are allowed. Similarly, the judgment C denies T^G characterizes which graph nodes are denied. These two judgments, defined formally in Table III, provide a recursive procedure to check whether a data dependency graph node satisfies a policy, given its attributes. Intuitively, a graph node is allowed by an ALLOW clause if and only if the clause applies and is allowed by each exception (rules A_1-A_3 in Table III). A graph node is denied by a DENY clause if and only if the clause applies and is denied by each exception (rules D_1-D_3).

$$\frac{T^G \not\sqsubseteq T^C}{\text{ALLOW } T^C \text{ EXCEPT } D_1 \cdots D_m \text{ denies } T^G} \ (A_1)$$

$$\frac{T^G \sqsubseteq T^C \quad \exists_i D_i \text{ denies } T^G}{\text{ALLOW } T^C \text{ EXCEPT } D_1 \cdots D_m \text{ denies } T^G} \ (A_2)$$

$$\frac{T^G \sqsubseteq T^C \quad \forall_i D_i \text{ allows } T^G}{\text{ALLOW } T^C \text{ EXCEPT } D_1 \cdots D_m \text{ allows } T^G} \ (A_3)$$

$$\frac{\bot \in T^G \sqcap T^C}{\text{DENY } T^C \text{ EXCEPT } A_1 \cdots A_m \text{ allows } T^G} \ (D_1)$$

$$\frac{\bot \notin T^G \sqcap T^C \quad \exists_i A_i \text{ allows } T^G \sqcap T^C}{\text{DENY } T^C \text{ EXCEPT } A_1 \cdots A_m \text{ allows } T^G} \ (D_2)$$

$$\frac{\bot \notin T^G \sqcap T^C \quad \forall_i A_i \text{ denies } T^G \sqcap T^C}{\text{DENY } T^C \text{ EXCEPT } A_1 \cdots A_m \text{ denies } T^G} \ (D_3)$$

TABLE III
INFERENCE RULES FOR LEGALEASE

As an example, consider the following policy clause that allows everything except for the use of IPAddress and AccountID in the same program. Note that individually, using either may be allowed.

```
ALLOW DataType ⊤
EXCEPT
  DENY DataType IPAddress, AccountID
```

For ease of exposition, we demonstrate policy evaluation only for the *DataType* attribute here and elide *DataType* subscripts. The *DataType* attribute of top level clause is $\{\top\}$ and as for any T^G, $T^G \sqsubseteq \{\top\}$, one of rules A_2 or A_3 applies and we need to check the exception. The exception has *DataType* attribute $T^C = \{\text{IPAddress}, \text{AccountID}\}$. Now consider two nodes with its *DataType* attributes being $T_1^G = \{\text{IPAddress}\}$ and $T_2^G = \{\text{IPAddress}, \text{AccountID}\}$. In the first case $T^C \sqcap T^G = \{\text{IPAddress}, \bot\}$, and therefore, the node is allowed by the policy. In the second case, on the other hand, $T^C \sqcap T^G = \{\text{IPAddress}, \text{AccountID}\}$, and therefore, the node is denied by the policy.

F. LEGALEASE Properties

We now use the formal definition of LEGALEASE to state some of its properties. Appendix A of [20] contains a more detailed discussion and proofs.

The first two are safety properties that ensure that checking is defined uniquely for each policy and graph node.

Proposition 1 (Totality): For each vector of sets of lattice elements T, and policy clause C, C allows T or C denies T.

Proposition 2 (Unicity): For each vector of sets of lattice elements T, and policy clause C, C allows T and C denies T, are not both true.

We then show that LEGALEASE respects a syntactic notion of weakening. Our notion of weakening captures the standard

$$\frac{T_1^C \sqsubseteq T_2^C}{\textsf{ALLOW } T_1^C \preceq \textsf{ALLOW } T_2^C} \qquad \frac{C_1 \preceq C_2 \quad C_2 \preceq C_3}{C_1 \preceq C_3}$$

$$\frac{D_n \preceq D'_n}{\textsf{ALLOW } T^C \textsf{ EXCEPT } D_1 \cdots D_n \preceq \textsf{ALLOW } T^C \textsf{ EXCEPT } D_1 \cdots D'_n}$$

TABLE IV
SELECTED RULES FOR WEAKENING LEGALEASE POLICIES

modes of editing a policy in LEGALEASE i.e., relaxing a clause, weakening an exception or removing an exception. Table IV contains selected rules for defining the weakening relation $\preceq$. The intuitive idea that weakening a policy should make it more permissive, is stated as follows:

Proposition 3 (Monotonicity): If $C_1 \preceq C_2$, then for any T^G, C_1 allows T^G implies that C_2 allows T^G and C_2 denies T^G implies C_1 denies T^G.

IV. DATA INVENTORY

Inference rules of LEGALEASE assume a data dependency graph labeled with the domain-specific attributes. Constructing such a graph in reality is a difficult process. In this section we present GROK which constructs and labels a data dependency graph over big data pipelines with data flow, storage, access, and purpose labels with minimal manual effort.

A. Design Goals

Our primary goal in designing the GROK mapper is to scale to the needs of a large big data system, i.e., it must scale to millions of lines of ever-changing source code, storing data in tens of millions of data files.

a) Exhaustive and up-to-date: We target Map-Reduce-like big data systems that store data and run processing jobs. Such systems have complete visibility into all data, accesses, and processing. Any data entering (or leaving) the system can do so only though a narrow set of upload (or download) APIs that require users to authenticate themselves to the system. Similarly, all jobs run on the system are submitted by authenticated users.

b) Bootstrapping: At the scale in which we are interested, bootstrapping a GROK is highly non-trivial. We cannot assume extra effort on the part of the developer (e.g., labeling schema-elements with policy datatypes). There are thousands of developers and any change affecting more than a few tens requires creating a new organizational process (awareness, trainings, code reviews, process audits, and so on). Thus, we are constrained in using purely automated approaches or approaches involving a small team to bootstrap a labeled data flow graph.

c) Verifiable and robust: As a result of bootstrapping without any developer effort, it is inevitable that there will be false-positives and false-negatives in our attribute labels. At a minimum we must be able to verify the correctness of inferred labels. At the same time, it is unreasonable to assume that the team will be able to verify correctness of all labels. We therefore expose an explicit confidence-level associated with any attribute label. Our privacy policy checker uses the GROK confidence values to rank violations so that we can direct auditor attention to violations it is more certain about.

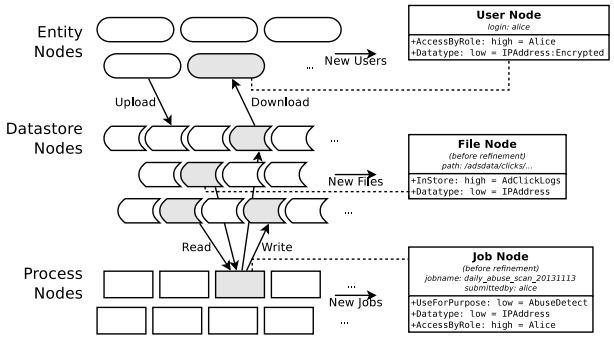


Fig. 5. A coarse-grained GROK data flow graph over users, files, and jobs.

B. GROK System

A GROK data flow graph (Fig. 5) contains a node for every principal (processes, data stores, entities) handling data in a system, and a directed edge between principals when data flows from one to another. Nodes are labeled with the domain-specific attributes mentioned in the previous section (callouts in the figure). The graph is updated with new nodes and edges as new principals and data flows are encountered. GROK associates a confidence score with each attribute label (labeled in the figure as high or low). Confidence values are based on how the attribute value was inferred.

Granularity: The finer the granularity of GROK, the more precision with which it can track data flows, but the higher the scalability cost of using that information. For example, at a fine granularity, there may be a process node for every line of executable code in every job; a data store node for every file; and an entity node for every human accessing the system. At coarse granularity there may be one process node for every job run on the cluster; one data store node for every logical separation of data (e.g., sub-directory); and one entity node for every functional team. Having one node per sub-directory is more scalable, but it conflates file level attributes that may not otherwise appear together on the same node thus trading off precision. Ultimately, the required precision is a function of the privacy policy — if the policy "We will not store account information with advertising data" is interpreted as not storing in the same sub-directory, then a coarse-grained GROK is precise enough.

Our GROK prototype is at a finer granularity than the examples above. There is a data node for each individual column in each file that contains tabular data, a process node for every field in every sub-expression in a statement of code, and entity nodes for each computer a user connects to the system with. For scalable use of the graph, we dynamically materialize a coarse-grained view by combining related columns (at the cost of precision), but allow any algorithm access to the underlying fine-grained graph as needed. By scalably targeting the finest granularity, we allow policy interpretations to change over time without having to change GROK.

```
Clicks =
        EXTRACT GUID, ClientIP
        FROM "/adsdata/clicks/20131113";
UserAgents =
        EXTRACT GUID, UserAgent
        FROM "/adsdata/uadata/20131113";

Suspect =
        SELECT Encrypt(ClientIP, "...") AS EncryptedIP
        FROM Clicks INNER JOIN UserAgents ON GUID
        WHERE MaybeFraud(UserAgent);

OUTPUT Suspect TO "/users/alice/output";
```

Fig. 6. Example of big data analysis code written in Scope [9].

C. Language Restrictions in Big Data Languages

Before we can describe how we bootstrap GROK, we explain semantics common to the three languages commonly used in industry for big data analysis — Hive, Dremel, and Scope. All three languages have the same basic data abstraction of a table. A *table* is a rectangle of data with a fixed number of columns, and an arbitrary number of rows. Each *column* has a name and a type (e.g., int, string). Jobs are a sequence of SQL-like expressions. Each *expression* of the language operates on one or more tables, and returns a resultant table that may be used in other expressions. Thus, the result of every expression (or sub-expression) in the program is also a table with named columns. The languages provide a mechanism to read a flat file (e.g., tab delimited) into a table; and a mechanism to write a table out as a flat file.

Implicit flow of data in these languages is very limited. There is no global state. User-defined functions (UDFs) are restricted to using only their input parameters columns, and their output is restricted only to the output column(s). UDFs cannot directly access the data store. Implicit flows due to `WHERE` clauses are made explicit by considering the columns referenced in the clause as input columns. The language itself does not have data dependent loops. The pre-processor provides syntactic sugar to write repetitive or conditional code driven by compile-time macros. The resulting code is straight-line code that explicitly tracks all flow of data.

Fig. 6 lists the SQL-like Scope source code for an analysis job we'll use as a running example in this section. The first statement reads in a flat file *"/adsdata/clicks/20131113"*, which contains tabular data, into the *Clicks* variable using the column schema (*GUID*, *ClientIP*) provided. The second line similarly reads another file into *UserAgents* with the given schema (*GUID*, *UserAgent*). The third statement joins the two tables on the *GUID* column, retains the rows where the UDF call *MaybeFraud(UserAgent)* returns true, computes the output table containing a single schema column named *EncryptedIP* populated with the result of the UDF call *Encrypt(ClientIP, "...")*, and binds it to the *Suspect* variable. The last statement outputs this table to *"/user/alice/output"* as a flat file.

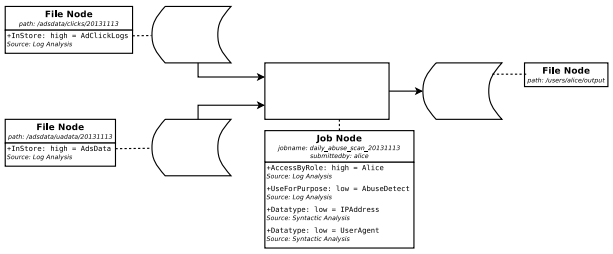


Fig. 7. Coarse-grained labeled data flow graph nodes for Fig. 6.

D. Data Flow Edges and Labeling Nodes

We use a variety of complementary approaches to construct the fine-grained data flow graph, and label it. We begin by constructing a coarse-grained data flow graph with *InStore* and *AccessByRole* attributes by analyzing logs. We then use extensive syntactic analysis of programs to add limited *DataType* and *UseForPurpose* attributes. Next, we use semantic analysis of programs to replace coarse-grained process and data store nodes with fine-grained internals. In the process we also use static data flow analysis to expand the coverage of *DataType* attributes. Finally, we identify a small set (few hundred) of bottleneck nodes that, if verified manually, allows us to increase the confidence score of the majority of the nodes in the graph. We describe each approach in detail.

1) Log Analysis: Inferring data flows at a coarse-granularity (job, file, user) is trivial given a log that contains all jobs that were run on the cluster, all the files the job accessed for read and write, and all users that downloaded (or uploaded) files from (to) the cluster. If this log is exhaustive and updated regularly, the corresponding GROK is also exhaustive and up-to-date, satisfying our first design goal.

In our deployment we use one such log to bootstrap the coarse-grained data flow graph. We also use this log to label file nodes with the *InStore* attribute, and entity nodes (and job nodes run by a user) with the *AccessByRole* attribute. We associate a high confidence score with these labels since the corresponding information is tracked explicitly; e.g., mapping between data store names and directories is created by the cluster admin, and as mentioned, only authenticated users can run jobs or access data. Lastly, since our org-hierarchy usually reflects the functional hierarchy (i.e., jobs for purpose AbuseDetect are run (only) by the AbuseTeam and vice versa), we associate a *UseForPurpose* attribute for each job based on the role of the user running the job. We associate a low confidence value for *UseForPurpose* attributes since it is based on a heuristic.

Fig. 7 illustrates the data flow information we glean just through log analysis. The data flow information includes the file and job nodes and the edges between them, and the non-*DataType* attributes attached to these nodes. We next discuss how we label *DataType* attributes.

2) Program Analysis (Syntactic): A scalable way of labeling nodes with the *DataType* attribute is to syntactically analyze the source code of the job that read or wrote data.

By syntactic analysis we mean inferring *DataType* attributes for job nodes based on the identifiers (e.g., tuple field names, column names) used in the source code to refer to data. Good coding practices enforced rigorously in engineering teams through code reviews, variable naming conventions, etc. require the developer to use comprehensible variable names in their programs. We use a set of regular expressions to infer a limited set of policy datatypes (and sometimes typestate) from *identifiers* in the source code. As before, we associate a low confidence to such inferences.

From our example in Fig. 6, we would extract the identifier names *Clicks, GUID, ClientIP, UserAgents, UserAgent, EncryptedIP*. Using regular expression patterns (see Section V) we may associate the *DataType* labels IPAddress with *ClientIP*, IPAddress:Encrypted with *EncryptedIP*, and UniqueID with *GUID*. All with low confidence.

Using regular expressions to label identifiers is a heuristic borne out of the necessity of bootstrapping GROK without access to the underlying data, without requiring developer effort, and in an environment where variable names, while comprehensible to humans, are not standardized; nevertheless it is a heuristic. Fortunately, a small set of patterns (3200) curated manually in a *one-time* effort allows us to label tens of millions of schema elements daily for which we would otherwise have no information. Having bootstrapped the GROK, we discuss in Section V how we reduce our dependence on this bootstrapping approach using highly targeted developer annotations going forward.

3) Program Analysis (Semantic): Next we leverage program semantics to refine coarse-grained file nodes to a collection of column nodes for that file, and refine coarse-grained job nodes to a sub-graph of nodes over the columns in the sub-expressions in the job.

a) File to schema refinement: Given the language semantics to read/write files into/from tables, we infer the columns in the file from the column names in the table. By applying the syntactic technique above on the column names, we ascribe low-confidence *DataType* labels to these columns. We then refine the file node in the catalog with the inferred columns, and update the edges in the graph so there is an edge only from the columns read to the job (or from the job to columns written).

b) Job to expressions, expressions to columns refinement: We refine a job node by including a node for each expression in the job. We then refine each expression node into a collection of columns, and analyze the source code to identify the other columns that are used to compute the current column and add the corresponding edges. We make conservative assumptions for UDFs — we conservatively add edges from all inputs to the output. The sub-graph representing the job reflects a conservative data flow through all columns in all sub-expressions in the job.

This refinement step is illustrated in Fig. 8. The file nodes have been replaced with multiple column nodes, and the job node has been replaced with a sub-graph of columns in the three expressions (from Fig. 6). The edges track data from which column (file or expression) flows into which column. Finally, we apply the syntactic analysis for each column to label it with low confidence *DataType* attribute values.

Note that output columns from one job is another job's input columns. Thus, this semantic analysis step allows us to construct a complete data flow graph at the granularity of columns that tracks the flow of all data across all jobs and all files in the big data system.

E. Data Flow Analysis

Next, we perform data flow analysis over the entire graph. We do this by copying the *DataType* attribute on one node to all nodes that data flows to (as long as the destination doesn't already have a higher confidence attribute). If the destination already has an attribute with the same confidence value, we replace it with the lattice join of the two attributes. If the data flow is through a UDF, we look for patterns in the UDF name to infer if the UDF modifies the typestate of the policy datatype (e.g., the UDF *Encrypt(...)* converts an input IPAddress to IPAddress:Encrypted); if a typestate transition is performed we force the confidence value to low.

Assuming our initial labels are correct, we compute the attributes along any path conservatively. Therefore, after the the data flow analysis, we know that if an input column with policy datatype attribute t, and confidence c interferes in state s with an output column with *DataType* attribute $t'{:}s'$ and confidence less than c, then we must have that $t{:}s \leq t'{:}s'$ in the *DataType* lattice.

Fortunately, the restrictive programming model helps us side-step the most common problem with information flow analysis where everything quickly gets saturated. Recall that data flows in Scope are very restricted (no global state or data driven loops) and UDFs are allowed in very specific settings and confined to operate only on the input columns and their results captured only in the output column. As a result, even with conservative treatment of UDFs (where all inputs flow to all outputs), *DataType* labels do not get saturated. Furthermore, the limited verification step below allows us to add high confidence labels at key points that limit the low confidence data flows, thus further containing the saturation effect.

F. Verifying Labels

While the approaches above give us high coverage for *DataType* attributes, they are all with low confidence due to the heuristics involved. Contacting the developer who wrote a piece of code for ultimate verification is usually a time-consuming process. Using the following greedy algorithm, we use GROK to minimize the number of developers we need to contact for verification. For each low confidence connected component in the data dependency graph where the syntactic analysis labeled at least one column, we identify all source code files (including shared code modules) that contributed a column node to that connected component. We then invert the mapping to determine the aggregate size of connected components a given source code file contributed

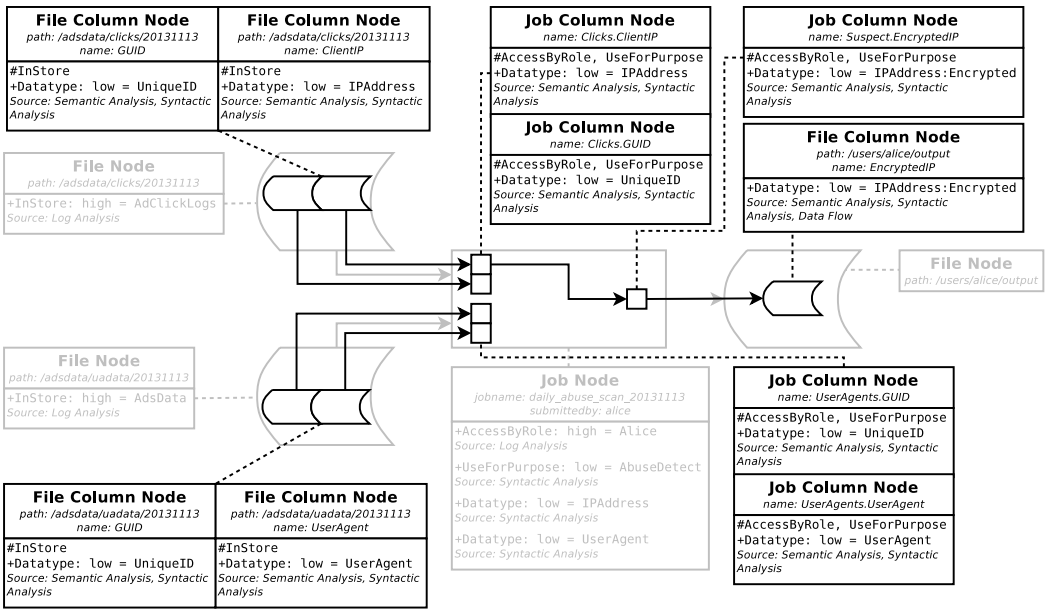


Fig. 8. Fine-grained labeled data flow graph nodes for Fig. 6.

columns to. We then contact the author of the highest-ranked source code file, verify and update the *DataType* attributes with high confidence and use the data flow analysis to label the connected components. We then repeat the process for the remaining low confidence connected components until we hit diminishing returns. We show during validation (Section VI) that by contacting only 12 developer teams and having them verify, on average, 18 nodes each we are able to attribute high confidence *DataType* labels to 60% of the data dependency graph (28 million nodes daily).

V. Praxis

We describe in this section our implementation of LEGALEASE and GROK, and lessons learned from deployment of our tool in practice.

A. Implementation

GROK*:* Is implemented as two components: a massively-parallel standalone static semantic analyzer for the Scope language, and a massively-scalable data flow analyzer. Both components run on the big data system itself. The semantic analyzer processes individual jobs from the cluster log into the nodes and edges in the data dependency graph without any attributes. This component is stateless, and executed in a massively parallel manner processing tens of thousands of jobs per minute (we present scalability numbers in the validation section). The second component collates all the graph nodes (for any arbitrary number of days past) at the granularity desired for checking the privacy policy, performs syntactic analysis and conservative data flow analysis over the entire graph, and outputs graph nodes augmented with the *DataType* and other attributes. The two components together comprise 6143 lines of C# code, 988 lines of Scope code, and take as input a 3203 line configuration file that contains the regular expression patterns used for the syntactic analysis phase, and the manual verification results from prior runs.

LEGALEASE*:* The policy checker is implemented in 652 lines of C# code, takes as input the LEGALEASE privacy policy specification (Table V), evaluates it over GROK's output, and outputs a ranked list of graph nodes for subsequent manual verification. The output is ranked based on the GROK confidence values for the labels that resulted in the violation.

B. Experience and Lessons Learned

We discuss three lessons we learned during the process of bootstrapping the complete system. First, defining patterns for syntactic analysis, while laborious, has a tremendous payoff. Second, our light-weight solution to checking simple temporal properties like data retention. And third, our solution to minimizing the verification effort through developer annotations.

a) Defining patterns for syntactic analysis: To define patterns for syntactic analysis, we manually analyzed around 150K unique column and variable names (from a day's worth of jobs). We identified on the order of 40 regular expressions for roughly as many lattice elements for policy datatypes (e.g., *%email%* for columns that might contain an Email), and on the order of 400 exact matches based on domain knowledge. We found that these regular expressions had some false-positives, for instance, labeling the column *emailResponseRate*, a floating-point value, as an email address. We enforced type restrictions (available during semantic analysis). While this helped reduce false-positives, it did not eliminate them (e.g., column *emailProvider*, which is a string). We manually examined the column names to identify obvious false-positives, and defined a set of around 2500 negative exact matches (across all policy datatypes). Finally, during the first

manual verification we found cases where the inference was correct (i.e., the column *entityEmail* did indeed have email), but that they were for business listing in publicly crawlable web data (and that the team followed a naming convention). We added such conventions we discovered through developer interactions to the set of negative patterns. Thus in our current deployment, a column is labeled Email if it matches any positive pattern or exact match defined for email, and does not match any negative pattern or negative exact match.

Overall this process was laborious, taking one person one full week to construct the GROK configuration file. Having spent that one-time effort, however, the 3203 lines in the configuration file today label with high precision (based on verifying a random sample) on the order of millions of graph nodes daily.

b) Retention and limited temporal properties: While the big data system offers developers three mechanisms for ensuring that data is deleted after the retention period elapses, the underlying log data on which GROK is built gives us visibility into developers using only one of those mechanisms. For coverage, we use the data dependency graph to trace back the origin of any piece of data and when it was first seen in GROK. We automatically compute the day when that data should be deleted, subtract a two week buffer, and update GROK setting the typestate to :Expired; any subsequent use of this near-expiry data is output as a verification task item. Labelling data with :Expired allows the audit team to identify teams using data that is near-expiry and ensure that the teams are using one of the other two mechanisms to delete the data on time.

c) Reducing verification time through developer code annotations: Our current bottleneck is auditor bandwidth since following up with (even a small number of) developers is time-consuming. Instead of a manual audit process, we are currently piloting code annotations that a small number of developers can add (proactively) to disambiguate the policy datatypes they access.

VI. VALIDATION

We experimentally validate our approach along two axes: first, the scalability and coverage of the GROK data inventory, and second, the usability and expressiveness of the LEGALEASE language.

A. Scale

Fig. 9 shows the number of new nodes added to the GROK each day over a 100 day period for our operational system. On average, we process over 77 thousand jobs each day, submitted by over 7 thousand entities in over 300 functional units. We process daily, on average, 1.1 million unique lines of code (including generated code), 21% of which changes (on average) on a day-to-day basis, covering 46 million dynamic table schemas. These jobs process tables persisted to 32 million files. Building the fine-grained column-level GROK data dependency graph takes, on average, 20 minutes daily on our production cluster. Performing data flow analysis over all data use on the cluster in a four week period takes 10 minutes.

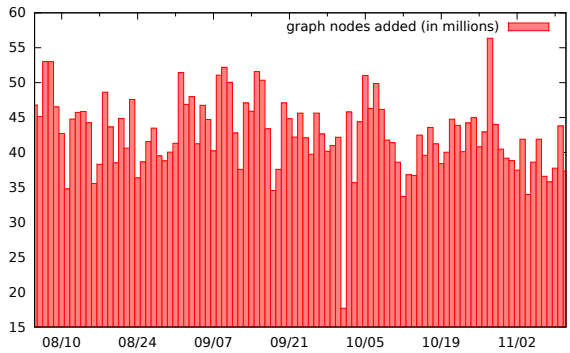


Fig. 9. Number of GROK data flow graph nodes added each day

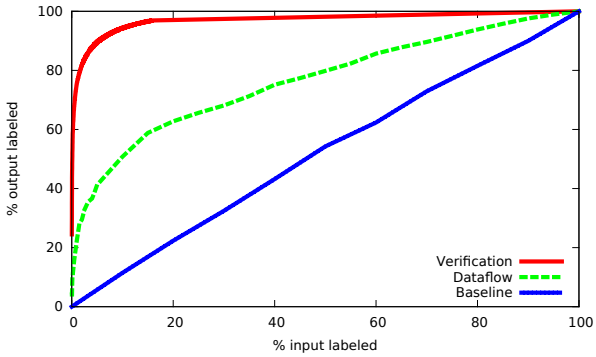


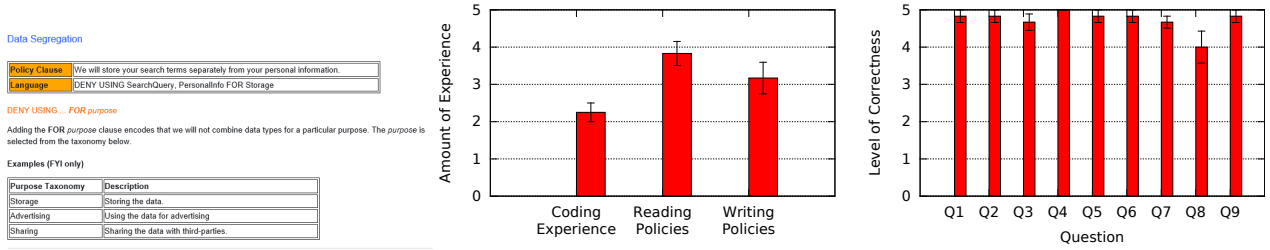
Fig. 10. Coverage of labeling by successive phases of GROK bootstrapping.

Note that this last number, 10 minutes, is the time it takes the system to take an unlabeled data dependency graph over the past several weeks, label it with attributes based on syntactic analysis and past verification, perform data flow analysis, and evaluate the configured policy over historical data. This quick turnaround allows us to perform rapid *what-if analysis for proposed policy changes*; a capability that is unattainable with manual reviews and audits that operate at the time-scale of months.

B. Coverage

We seek to understand the overall coverage of accurate *DataType* labels in GROK. The overall coverage depends on the coverage of the bootstrap syntactic analysis, improvements we get from data flow analysis, and boost in coverage and confidence values we get from manual verification. Fig. 10 plots how data flow analysis, and targeted manual verification improve the GROK coverage relative to the baseline.

We establish a baseline by simulating a syntactic analysis with varying degrees of coverage on our real-world data dependency graph. Specifically, we first pick $x\%$ of all *unique* column names uniformly at random, and flag them in our simulation as correctly labeled. We note first that a linear baseline is not a trivial result since the overall graph nodes labeled correctly is a function of the popularity of column



(a) A snapshot from the pre-survey training

(b) Coding and Policy experience levels

(c) Overview of Survey Performance

Fig. 11. Summary of the Usability Survey

names. There is no knee or shoulder that would imply a sweet-spot for the coverage vs. effort trade-off in syntactic analysis. Improvements in coverage of accurate labels in syntactic analysis translate linearly to overall improvement.

The overall coverage is improved using data flow analysis. This improvement is because correctly labeling a node in a connected component allows the entire connected component to be labeled. The dataflow line in Fig. 10 shows overall coverage as a function of connected components, ordered by size, that are labeled correctly using syntactic analysis and then the label flowed using data flow analysis. We find that by focusing on only 10% of the top connected components, we can boost overall coverage to 50%. However, we observe that labeling more connected components leads to diminishing returns.

The biggest improvement to overall coverage comes from limited manual verification. As mentioned, we analyze jobs to identify columns in sub-expressions in shared code modules that, if verified, allow us to flow the labels most broadly. The verification line in Fig. 10 shows that manual verification of only 0.2% of code modules (maintained by 12 teams) and adding code annotations to 182 lines of source code (out of several millions), combined with data flow and syntactic analysis, increases overall coverage of accurate *DataType* labels to 60%.

C. Usability

To develop a preliminary understanding of the ability for non-technical subject matter experts in privacy to understand and use the LEGALEASE language, we conducted an online survey targeting privacy champions in Microsoft. In the survey, we described the LEGALEASE language and asked participants to encode clauses of a privacy policy that we found online.

Survey design: We provided a 1-page definition of LEGALEASE terms, example clauses, and lattice elements (Fig. 11a); this single page of text and tables was our sole training tool[2]. After reading through the training information, participants were presented with 9 policy clauses to encode. The clauses increased in complexity as they progressed through the survey. For each question, participants were provided with a set of lattice elements to choose from (so that participants would not be required to memorize the lattice presented on the training page), and a text box to type in the LEGALEASE policy clause.

Participants: Participants ($n = 12$) were recruited via a company mailing list and were not provided with compensation. They were primarily privacy champions who had been in their position from 2 weeks to over 6 years. As shown in Fig. 11b, in general, based on their ratings of how much coding experience they had ranging from "No experience at all" (1) to "Expert" (5), they were not experienced in coding (mean 2.25)[3]. As privacy champions, they did have more experience reading privacy policies (mean 3.83)[4], and were neither experienced nor inexperienced in writing privacy policies (mean 3.17)[5] on the same scale. After the coding tasks, participants were neutral about the difficulty of the task (mean 3.17)[6] on the scale of "Very Difficult" (1) to "Very Easy" (5).

Results: After reading the training information (the average time spent on the tutorial page was 2.4 minutes), the majority of participants were able to code each policy clause with the correct answer (see Fig. 11c for a question by question breakdown of correctness, ranging from "Incorrect" (1) to "Correct" (5).) The overall correctness rating for all participants was 4.65 (standard deviation 0.48). The time spent on encoding clauses was 14.3 minutes on average. Overall, our sample of privacy champions was able to use LEGALEASE to code policy clauses at a high level of correctness with very little training in a short amount of time.

D. Expressiveness

To demonstrate the expressiveness of LEGALEASE, We now present a complete encoding of externally-visible privacy policies[7] of Google and Bing that applies to data storage and use. We also demonstrate the LEGALEASE goal of usability through 1-1 correspondence with the English policy clauses by presenting a side-by-side view (Tables V and VI).

Note that the policies in Table V were part of the survey above. The LEGALEASE clauses for them are the actual (ma-

[2]Particpants had the ability to open the page of training information in a new window while completing the encoding tasks.

[3]M=2.25, $t(11)$=3, p=0.01, as compared to the midpoint in a one-sample t-test

[4]M=3.83, $t(11)$=2.59, p=0.03, as compared to the midpoint in a one-sample t-test

[5]M=3.17, $t(11)$=0.39, p=0.7, as compared to the midpoint in a one-sample t-test

[6]M=3.17, $t(11)$=0.56, p=0.59, as compared to the midpoint in a one-sample t-test

[7]As of 14th October 2013

ALLOW EXCEPT	
DENY *DataType* IPaddress:Expired	◁ "we remove the entirety of the IP address after 6 months"
DENY *DataType* UniqueIdentifier:Expired	◁ "[we remove] cookies and other cross session identifiers, after 18 months"
DENY *DataType* SearchQuery, PII *InStore* Store DENY *DataType* UniqueIdentifier, PII *InStore* Store	◁ "We store search terms (and the cookie IDs associated with search terms) separately from any account information that directly identifies the user, such as name, e-mail address, or phone numbers."
DENY *DataType* BBEPData *UseForPurpose* Advertising	◁ "we do not use any of the information collected through the Bing Bar Experience Improvement Program to identify, contact or target advertising to you"
DENY *DataType* BBEPData, PII *InStore* Store	◁ "we take steps to store [information collected through the Bing Bar Experience Improvement Program] separately from any account information we may have that directly identifies you, such as name, e-mail address, or phone numbers"
DENY *DataType* BBEPData:Expired	◁ "we delete the information collected through the Bing Bar Experience Program at eighteen months."
DENY *DataType* UserProfile, PII *InStore* Store	◁ "we store page views, clicks and search terms used for ad targeting separately from contact information you may have provided or other data that directly identifies you (such as your name, e-mail address, etc.)."
DENY *DataType* PII *UseForPurpose* Advertising DENY *DataType* PII *InStore* AdStore	◁ "our advertising systems do not contain or use any information that can personally and directly identify you (such as your name, email address and phone number)."
DENY *DataType* SearchQuery *UseForPurpose* Sharing EXCEPT ALLOW *DataType* SearchQuery:Scrubbed	◁ "Before we [share some search query data], we remove all unique identifiers such as IP addresses and cookie IDs from the data."

TABLE V
AN ENCODING OF PRIVACY PROMISES BY BING AS OF OCTOBER 2013

ALLOW EXCEPT	
DENY *DataType* PII *UseForPurpose* Sharing	◁ "We do not share personal information with companies, organizations and individuals outside of Google unless one of the following circumstances apply:"
EXCEPT ALLOW *DataType* PII:OptIn	◁ "We require opt-in consent for the sharing of any sensitive personal information."
EXCEPT ALLOW *AccessByRole* Affiliates	◁ "We provide personal information to our affiliates or other trusted businesses or persons to process it for us"
EXCEPT ALLOW *UseForPurpose* Legal	◁ "We will share personal information [if necessary to] meet any applicable law, regulation, legal process or enforceable governmental request."
DENY *DataType* DoubleClickData, PII EXCEPT ALLOW *DataType* DoubleClickData, PII:Optin	◁ "We will not combine DoubleClick cookie information with personally identifiable information unless we have your opt-in consent"

TABLE VI
AN ENCODING OF PRIVACY PROMISES BY GOOGLE AS OF OCTOBER 2013

jority) response provided by the surveyed privacy champions, who are the intended target users of LEGALEASE.

VII. DISCUSSION

In this section we discuss some non-goals of LEGALEASE and GROK, some limitations and mitigating factors.

Expressiveness: LEGALEASE does not support expressing policies based on first-order temporal-logic. It supports a restricted form of temporal policies, implemented with help from the GROK. LEGALEASE is intended as a bridge between developers and policy makers in Web service companies like Bing and its expressiveness is restricted to policy elements encountered in practice that apply to the big data system. In particular, policies such as those related to cookie management and the use of secure communication channels are beyond the scope of our analysis.

Inference of Sensitive Data: Sensitive data can often be inferred from non-sensitive data [21], [22]. Unless explicitly labeled, GROK cannot detect such inferences. A careful choice of the *DataType* lattice may help reduce some of these risks by classifying together data that can be reasonably inferred from each other.

Precision: The information flow analysis in GROK is conservative, but not necessarily precise. A major source of imprecision is our overly conservative treatment of user defined functions. In the future, we hope to leverage static code analysis of C# user defined functions in the flavor of [23], [24] to make GROK more precise.

False Negatives: The semantics of LEGALEASE are precise and the information flow analysis in GROK is conservative. Therefore, bootstrapping that leads to more coverage of the graph with labels, would generally imply a reduction in false positives. However, due to the lack of ground truth for labels, we are unable to characterize the exact nature of false negatives in our system.

Assurance: Our system provides weak guarantees in the face

of adversarial developer behavior such as incorrect annotations and intentional flouting of naming conventions to mislead our bootstrap analysis. However, we expect independent redundant annotations in conjunction with the data flow analysis to significantly enhance correctness and confidence of these labels over time.

VIII. Related Work

Recall that we focus on privacy policies that impose restrictions on how various types of personal information flow amongst programs. There are two main lines of work that are closely related to ours—*information flow analysis of programs* and *privacy policy enforcement over executions*. Furthermore, we also describe related work on *usable policy languages*.

a) Information flow analysis of programs: There has been significant research activity in restricting information flows in programs over the last three decades [25] and on language-based methods that support these restrictions, including languages like Jif [26], which augments Java with information flow types, and Flow Caml, which augments ML [27] (see [17] for a survey of these and other language-based methods). These languages can enforce information flow properties like non-interference with mostly static type checking. Taking Jif as one example language, we note that prior work has shown that Jif principals can be used to model role-based [26] and purpose-based [28] restrictions on information flow. Additionally, recognizing that non-interference is too strong a requirement, the theory of relaxed non-interference through declassification [29], [30], [31], allows expressing policies that, for instance, do not allow disclosure of individual ages, but allow the disclosure of average age. This line of work also includes techniques for automated inference of declassification policies [32], [33] with minimal programmer annotations. While these ideas have parallels in our work, there are also some significant differences. First, our policy language LEGALEASE enables explicit specification of policies separately from the code whereas in language-based approaches like Jif the policies are either expressed implicitly via typed interface specifications or explicitly via conditionals on program variables. The separation of high-level policy specification from code is crucial in our setting since we want the first task to be accessible to privacy champions and lawyers. Second, since our goal is to bootstrap compliance checking on existing code, we do not assume that the code is annotated with information flow labels. A central challenge (addressed by GROK) is to bootstrap these labels without significant human effort. Once the labels are in place, information flow analysis for our restricted programming model is much simpler than it is for more complex languages like Jif. Note that we (as well as Hayati and Abadi [28]) assume that programs are correctly annotated with their purposes. A semantic definition of what it means for an agent (a program or human) to use information for a purpose is an orthogonal challenge, addressed in part in other work [34].

b) Privacy policy enforcement over executions: A second line of work checks executions of systems (i.e., traces of actions produced by programs or humans) for compliance with privacy policies that restrict how personal information may flow or be used. This line of work includes auditing, run-time monitoring, and logic programming methods for expressive fragments of first-order logic and first-order temporal logics [12], [35], [36], [37] applied to practical policies from healthcare, finance and other sectors. These results are different from ours in two ways. First, their language of restrictions on information flow is more expressive than ours—they can encode role-based and purpose-based restrictions much like we do, but can express a much larger class of temporal restrictions than we can in LEGALEASE with our limited typestates on data. Second, since their enforcement engines only have access to executions and not the code of programs, they can only check for direct flows of information and not non-interference-like properties. Such code analysis is also a point of difference from enforcement using reference monitors of access control and privacy policy languages—an area in which there is a large body of work, including languages such as XACML [38] and EPAL [39].

c) Usable policy languages: To author policy statements, several interfaces and tools have been tested for their usability. Rarely have the raw languages been evaluated on their own for ease of use without some kind of UI-based authoring tool. Thus, a direct comparison with our work is difficult. Nevertheless, we mention three efforts along these lines since our goals are similar to theirs. The Expandable Grids interface was used to test the ability for people to author P3P policies (P3P (Platform for Privacy Preferences) is the W3C standard for creating XML-based machine-readable privacy policies). An empirical study found that Expadable Grids did not improve the usability beyond users' abilities to express policies using natural language statements [40]. Another example is SPARCLE [41], a web-based policy authoring tool that generates XML based on the users' selection of user categories, actions, data categories, purposes, and conditions based on natural language policy clauses, with promising usability results. We view the need for authoring tools such as SPARCLE as being complementary to a language that can be used by policy authors, as such automated translation tools usually entail inaccuracies, and therefore, need to be validated. In a language such as LEGALEASE, one can hope that the translation is verified by the authors themselves.

IX. Conclusion

In this paper, we demonstrate a collection of techniques to transition to automated privacy compliance compliance checking in big data systems. To this end we designed the LEGALEASE language, instantiated for stating privacy policies as a form of restrictions on information flows, and the GROK data inventory that maps low level data types in code to high-level policy concepts. We show that LEGALEASE is usable by non-technical privacy champions through a user study. We show that LEGALEASE is expressive enough to capture real-world privacy policies with purpose, role, and storage restrictions with some limited temporal properties, in particular

that of Bing and Google. To build the GROK data flow graph we leveraged past work in program analysis and data flow analysis. We demonstrate how to bootstrap labeling the graph with LEGALEASE policy datatypes at massive scale. We note that the structure of the graph allows a small number of annotations to cover a large fraction of the graph. We report on our experiences and learnings from operating the system for over a year in Bing.

ACKNOWLEDGEMENTS

We thank the policy authors and privacy champions at Microsoft who participated in our user study. We thank Leena Sheth, Carrie Culley, Boris Asipov and Robert Chen for their contributions to the operational system. We thank Michael Tschantz and the anonymous reviewers for useful feedback. This work was partially supported by the AFOSR MURI on "Science of Cybersecurity" and the National Science Foundation (NSF) grant CNS1064688 on "Semantics and Enforcement of Privacy Policies: Information Use and Purpose".

REFERENCES

[1] D. J. Solove and W. Hartzog, "The FTC and the New Common Law of Privacy," *Columbia Law Review (forthcoming 2014)*., vol. 114.

[2] (2012, Aug.) Ftc approves final settlement with facebook. Federal Trade Commission. [Online]. Available: http://www.ftc.gov/news-events/press-releases/2012/08/ftc-approves-final-settlement-facebook

[3] (2011, Mar.) Ftc charges deceptive privacy practices in google's rollout of its buzz social network. Federal Trade Commission. [Online]. Available: http://www.ftc.gov/opa/2011/03/google.shtm

[4] Data Protection Commissioner, Ireland, "Facebook ireland ltd, report of re-audit," 2012. [Online]. Available: http://www.dataprotection.ie/documents/press/Facebook_Ireland_Audit_Review_Report_21_Sept_2012.pdf

[5] Information Commissioner's Office, United Kingdom, "Google inc.: Data protection audit report," 2011. [Online]. Available: http://ico.org.uk/~/media/documents/disclosure_log/IRQ0405239b.ashx

[6] (2012, Apr.) Investigations of Google Street View. Electronic Privacy Information Center (EPIC). [Online]. Available: http://epic.org/privacy/streetview/

[7] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy, "Hive - a petabyte scale data warehouse using Hadoop," in *ICDE '10: Proceedings of the 26th International Conference on Data Engineering*. IEEE, Mar. 2010, pp. 996–1005.

[8] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: Interactive analysis of web-scale datasets," *PVLDB*, vol. 3, no. 1, pp. 330–339, 2010.

[9] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *Proc. VLDB Endow.*, vol. 1, no. 2, pp. 1265–1276, Aug. 2008.

[10] Bing. [Online]. Available: http://www.bing.org/

[11] H. DeYoung, D. Garg, L. Jia, D. Kaynar, and A. Datta, "Experiences in the logical specification of the hipaa and glba privacy laws," in *Proceedings of the 9th Annual ACM Workshop on Privacy in the Electronic Society*. New York, NY, USA: ACM, 2010, pp. 73–82.

[12] A. Barth, A. Datta, J. Mitchell, and H. Nissenbaum, "Privacy and contextual integrity: framework and applications," in *Security and Privacy, 2006 IEEE Symposium on*, 2006, pp. 15 pp.–198.

[13] J. A. Goguen and J. Meseguer, "Security policies and security models," in *IEEE Symposium on Security and Privacy*, 1982, pp. 11–20.

[14] Facebook. (2012, Dec.) Data use policy. [Online]. Available: https://www.facebook.com/full_data_use_policy

[15] Google. (2013, Jun.) Privacy policy. [Online]. Available: http://www.google.com/policies/privacy/

[16] (2013, Oct.) Bing privacy statement. Microsoft. [Online]. Available: http://www.microsoft.com/privacystatement/en-gb/bing/default.aspx

[17] A. Sabelfeld and A. Myers, "Language-based information-flow security," *Selected Areas in Communications, IEEE Journal on*, vol. 21, no. 1, pp. 5–19, 2003.

[18] M. C. Tschantz and S. Krishnamurthi, "Towards reasonability properties for access-control policy languages." in *SACMAT*. ACM, 2006, pp. 160–169.

[19] R. Wille, "Restructuring lattice theory: An approach based on hierarchies of concepts," in *Ordered Sets*, ser. NATO Advanced Study Institutes Series. Springer Netherlands, 1982, vol. 83, pp. 445–470.

[20] S. Sen, S. Guha, A. Datta, S. K. Rajamani, J. Tsai, and J. M. Wing, "Bootstrapping privacy compliance in a big data system," Microsoft Research, Tech. Rep. MSR-TR-2014-36.

[21] C. Dwork, "Differential privacy," in *in ICALP*. Springer, 2006, pp. 1–12.

[22] C. Farkas and S. Jajodia, "The inference problem: A survey," *SIGKDD Explor. Newsl.*, vol. 4, no. 2, pp. 6–11, Dec. 2002.

[23] R. Madhavan, G. Ramalingam, and K. Vaswani, "Purity analysis: An abstract interpretation formulation," in *Static Analysis*, ser. Lecture Notes in Computer Science, E. Yahav, Ed. Springer Berlin Heidelberg, 2011, vol. 6887, pp. 7–24.

[24] C. Gkantsidis, D. Vytiniotis, O. Hodson, D. Narayanan, F. Dinu, and A. Rowstron, "Rhea: Automatic filtering for unstructured cloud storage," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'13. Berkeley, CA, USA: USENIX Association, 2013, pp. 343–356.

[25] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Commun. ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[26] A. C. Myers and B. Liskov, "Protecting privacy using the decentralized label model," *ACM Trans. Softw. Eng. Methodol.*, vol. 9, no. 4, pp. 410–442, 2000.

[27] F. Pottier and V. Simonet, "Information flow inference for ml," in *POPL*, 2002, pp. 319–330.

[28] K. Hayati and M. Abadi, "Language-based enforcement of privacy policies," in *In Proceedings of Privacy Enhancing Technologies Workshop (PET)*. Springer-Verlag, 2004.

[29] S. Chong and A. C. Myers, "Security policies for downgrading," in *Proceedings of the 11th ACM conference on Computer and communications security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 198–209.

[30] P. Li and S. Zdancewic, "Downgrading policies and relaxed noninterference." in *POPL*. ACM, 2005, pp. 158–170.

[31] A. Sabelfeld and D. Sands, "Declassification: Dimensions and principles," *Journal of Computer Security*, vol. 17, no. 5, pp. 517–548, 2009.

[32] M. C. Tschantz and J. M. Wing, "Extracting conditional confidentiality policies," in *Proceedings of the 2008 Sixth IEEE International Conference on Software Engineering and Formal Methods*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 107–116.

[33] J. A. Vaughan and S. Chong, "Inference of expressive declassification policies," in *Proceedings of the 2011 IEEE Symposium on Security and Privacy*. Washington, DC, USA: IEEE Computer Society, 2011, pp. 180–195.

[34] M. C. Tschantz, A. Datta, and J. M. Wing, "Purpose restrictions on information use," in *Computer Security - ESORICS 2013*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, vol. 8134, pp. 610–627.

[35] D. A. Basin, F. Klaedtke, S. Müller, and B. Pfitzmann, "Runtime monitoring of metric first-order temporal properties," in *FSTTCS*, 2008, pp. 49–60.

[36] D. Garg, L. Jia, and A. Datta, "Policy auditing over incomplete logs: theory, implementation and applications," in *Proceedings of the 18th ACM conference on Computer and communications security*, ser. CCS '11. New York, NY, USA: ACM, 2011, pp. 151–162.

[37] D. A. Basin, F. Klaedtke, S. Marinovic, and E. Zalinescu, "Monitoring compliance policies over incomplete and disagreeing logs," in *RV*, 2012, pp. 151–167.

[38] T. Moses *et al.*, "Extensible access control markup language (xacml) version 2.0," *Oasis Standard*, vol. 200502, 2005.

[39] P. Ashley, S. Hada, G. Karjoth, C. Powers, and M. Schunter, "Enterprise privacy authorization language (epal 1.2)," *Submission to W3C*, 2003.

[40] R. W. Reeder, L. Bauer, L. F. Cranor, M. K. Reiter, K. Bacon, K. How, and H. Strong, "Expandable grids for visualizing and authoring computer security policies," in *CHI*, 2008, pp. 1473–1482.

[41] C. Brodie, C.-M. Karat, and J. Karat, "An empirical study of natural language parsing of privacy policy rules using the sparcle policy workbench," in *SOUPS*, 2006, pp. 8–19.