A Data Usage Control System using Dynamic Taint Tracking

Julian Schütte, Gerd Stefan Brost

Abstract

Applications processing an increasing volume of structured personal information are on the rise, fostered by trends like "Quantified Self", "Smart Home", and "Future Internet". As a result, users who want to take advantage of such data analytics services are confronted with the challenge of keeping their personal data and business secrets secure, while still providing the information required for the analytics service to operate. Traditional access and usage control do not solve this problem, as they only take binary access decisions, but do not enforce specific views on data sets. We propose a mechanism to control the ways in which data may be processed, thereby limiting the information which can be gained from data sets to the specific needs of a service. The core of our approach is to model data analytics as a data flow problem and to apply dynamic taint analysis for monitoring the processing of individual records. We propose a policy language to state requirements on the way how data is processed and enforce measures to ensure that critical data is not revealed. Our approach is based on the query evaluation of a complex event processing engine, which is thereby turned into a policy-controlled privacy-preserving data analytics service.

1 Introduction

Keeping personal information under control has been a challenge in computing ever since. In the case of personal data, privacy concerns must be taken into consideration and in the case of business data it must be ensured that no information relevant to business secrets can be extracted. However, collecting and providing such critical data is inevitable, if users want to take advantage of the benefits of data analytics. The goal is thus to make data available exactly to the degree and in the perspective that is required for specific types of analysis, while preventing all other ways of extracting information from it. In the context of this paper, we will refer to data analytics as *data usage* and understand it as the application of operations to sets of data records.

Today, mostly access control concepts are found in practice. Users can control who may access their data and choose third party services they want to share data with. However, access control is a very limited concept when it comes to data usage, because it constitutes only a single precondition which must be fulfilled before data may be accessed. For example, possession of a specific token, key, or password will allow to retrieve data and use it in any way

and for any time span. There is no restriction on how data may be used, in which way it has to be treated, to whom it may be forwarded, or how long it may be stored.

Some of these aspects are addressed by the concept of usage control which has been subject to extensive research in the last decade [1, 2, 3, 4], but has barely found adoption in practice.

Usage control allows to determine under which conditions resources such as files, network connections, or databases may be used. It is thus an extension of the traditional access control paradigm in that it does not refer to a single point in time before access in granted, but rather aims at continuously enforce requirements while resources are in use. However, just like access control, usage control refers to *resources* in general and does not address the actual processing of data.

Being able to control the actual way in which data may be used is a predominant requirement in all scenarios which deal with analytics of personal or business-critical data, where different views on the data set allow to extract different information. In the case of personal data, for example, concepts from privacy research like k-anonymity [5], l-diversity [6], and t-closeness [7] deal with inference of critical information by correlating uncritical data attributes and taking into account public background knowledge. This problem is caused by quasi-identifiers, i.e. attributes when combined, allow to uniquely identify an individual within a data set. In this case, it must thus be possible to prevent combined retrieval of respective quasi-identifiers to avoid extraction of critical information.

We propose a solution to this problem by means of a *data usage control* system, which is orthogonal to traditional usage control in that it does not continuously monitors access decisions on resources like files or sockets, but rather controls the way *how* data is used, i.e. how it is processed, transformed, and combined. Our data usage control system consists of a domain specific policy language (DSL) which allows users to limit data usage according to their requirements and an enforcement mechanism which ensures that data is processed in accordance with the policy or otherwise takes counteractions.

The core of our approach is to model data usage as a data flow problem to which we apply a dynamic taint analysis to trace data records as different operations are applied to them. When data record enter the processing, they are initially marked with a taint flag indicating their criticality or type (such as `PERSONAL` or `LOCATION`). As they are passed between different operations, their taint state is tracked and may change, i.e. a record may become untainted or additional taint flags may be added. When data is about to leave the processing its taint state is evaluated against the policy, possibly leading to removal or modification of the record in the result set.

Summarizing, our contribution is in two aspects:

1. a refinement of the traditional usage control concept based on resources (files, network, etc.) to fine-granular data analytics control
2. a specific enforcement mechanism which controls data *processing*, instead

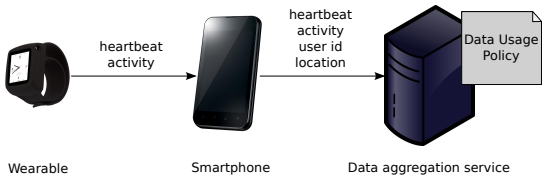

Fig. 1: Aggregating health data from IoT devices

of data *access and handling* like in previous work.

The paper is structured as follows: section 2 motivates our work by introducing a use case example and deriving requirements on the usage control system from it. In section 4 we introduce the data usage control model and discuss its implementation based on a taint analysis integrated in the evaluation of complex event processing (CEP) queries. In section 6 we discuss the results of our prototype evaluations and section 7 concludes the paper and outlines future work.

2 Motivation

We motivate the need for data usage by giving three example use cases from practice, one for IoT wearables, one for smart manufacturing, and one for traditional data processing.

2.1 Personal data in IoT applications

Consider a typical IoT setup consisting of a wearable device, a smartphone, and an only data aggregation services. The wearable device comprises sensors to collect different types of data on the health status of the user, such as the heart rate and the current activity as one of *sleeping*, *sitting*, *walking*, or *running*. The wearable communicates via Bluetooth with a smartphone which receives the transmitted data and extends it with a unique user id and the current location. The combined data record is then forwarded to a data analytics services which collects and processes it and offers the result to consumers. In Figure 1, the analytics service is depicted outside of the smartphone, but is is of course perfectly feasible to assume it resides within the user's immediate domain of control, i.e. as a service on the smartphone, for example.

The challenge here is to make use of interesting analytics, such overall improvements in the user's exercise program, circadian rhythm, and fitness level, while at the same time preventing the analytics service from gaining further possibly intimate information such as the specific location at a certain time

2.2 Predictive maintenance in manufacturing processes

Industrial production lines include various sensors to monitor and control the manufacturing process, such as flow meters measuring the exact volume of in-

gredients in biochemical production. For the most part, these sensors are highly complex integrated devices with the ability to communicate with remote endpoints. For the sensor producer it is highly interesting to get access to the measurements and status information of the devices, as it allows him to early detect drifts and errors in the device and thereby predict breakdowns and maintenances. On the other hand, raw data from sensors in the manufacturing process allows to reverse engineer precise formulas and recipes, which are the heart of a manufacturer's business secret.

The challenge in this case is thus to allow the sensor producer to run exactly the analytics required to detect failures in the devices, without revealing raw data from which business-critical information could be extracted.

2.3 Processing and selling of anonymized personal data

When patients receive a prescription from a GP and buy respective pharmaceuticals from a pharmacy, accounting records for billing the insurance are created and processed by trusted data centers. These records are however also highly interesting for statistical analysis on types of diseases, correlated with locations or doctors, as well as statistical information on medical sales. Data centers are allowed to offer such statistics, but they must ensure that they stick to data protection regulations like HIPAA in the U.S. or BDsG in Germany, which require them anonymize the data. This includes not only removal of first-class identifiers, but also ensuring that the identity of individual patients or doctors cannot be reconstructed by combining quasi-identifiers.

The challenge here is thus to provide only specific statistical perspectives on the data set where each perspective does not contain enough attributes to impose a quasi-identifier and to ensure that attribute values are blurred in a way that multiple perspectives cannot be combined into one overall data set.

Our data usage control system is designed to address these challenges and allow users to precisely state the ways how data may be processed. This includes the definition of a *policy model*,including a language which allows users to express valid ways of processing. By means of the policy model, *sources* of sensitive data can be declared. Further, it allows to state restrictions on data processing such as a minimum or maximum *number of aggregated* items, the *retrieval frequency*, or *untainting operations* that if applied to data records will turn them uncritical. Also, operations like projection and selection are controllable, in order to prevent specific combinations or correlations of data records which are considered critical. Finally, the data usage control system allows users to specify *counteractions*, i.e. actions to be taken just before critical data is about to be revealed to the consumer. Typically, a counteraction would be to block the request, but it is also conceivable to forcibly blind attributes in order to anonymize records.

3 Related work

Usage control has been subject to extensive research for more than a decade. Various models for usage control policies have been proposed, whereas $UCON_{ABC}$, originally introduced by Park and Sandhu [2], is the most prominent one and has undergone different extensions in the course of time, such as [8] which incorporates post-obligations. It comprises Authorizations (A), oBligations (B), and Conditions (C), referring to attributes of subjects and resources. Moreover, attributes are mutable (e.g., they can change over time) and continuity of access decisions is formalized. In this way, UCON A, B and C can be defined to be evaluated before (pre) or during usage (on). $UCON_{ABC}$ leaves open how to design a specific architecture and mechanisms for usage control. Other approaches focus on specific languages rather than abstract models, such as the Obligation Specification Language (OSL) [9]. Much work has been done on the formalization of usage control policies and the formal analysis of policy properties. In [1], a formalization of $UCON_{ABC}$ in Lamport's Temporal Logic of Actions (TLA) is given, in [10, 11] Basin et al. give an approach on analyzing usage control policies formalized in first-order temporal logic (MFOTL), in [12] a Linear Time Logic (LTL) dialect is used for the sake of analyzing policies, and in [13] an analysis of dynamically changing usage control policies is described, based on Action Computation Tree Logic (ACTL). While our work is based on this basic research on usage control, our approach is more specific, in that it focuses on the application of usage control to data processing only.

The enforcement of usage control policies has always been a challenge as it requires system-specific approaches and trust relationships with enforcement points, possibly located on remote platforms. In [14], Pretschner et al. propose to transform non-enforceable obligations into observations which at least indicate violations of the policy. Trustworthy system architectures for usage control enforcement have been proposed in [15], which allow usage control policies at the level of system calls, given that the trustworthiness of the enforcement point can be attested using hardware-based mechanisms. Our approach does not focus at usage control enforcement on remote platforms, but rather on a specific enforcement mechanism for data processing. Nevertheless, it could be combined with techniques from [14] or [15] in case the enforcement would have to take place on remote hosts.

Closer related to our work is [3], which introduces the idea of using data flow tracing at the level of system calls in order to enforce usage control policies. The authors show that based on an underlying data flow model, more realistic and expressive policy rules can be written, referring to states of a data flow system, rather than specific sequences of events. In [16], this approached is extended by tracing messages in the X11 environment, specifically copy & paste actions on sensitive data which is either blocked or replaced by meaningless data in case a policy is violated. Similar to [3, 12], we understand usage control as enforcing conditions in data flows. Our approach differs from the aforementioned in that we do not aim at actual flows of data within a system, but rather use data flows to model complex queries in a data processing system, which allows

fine-granular control of the way how data must be aggregated or modified to be used. Also, our policy model is not based on the states of a data flow system, but on conditions over the propagation of data, which supports an efficient implementation and more expressive policies in the context of data processing.
The approach presented herein relies on an abstract representation of queries in terms of an abstract complex event processing language. Concrete representatives of such languages are the Continuous Query Language (CQL) [17], StreamQL [1], and the Esper Processing Language (EPL)[2], which we used in our prototype implementation.

Finegranular data flow tracking for databases has been done by applying taint tracking in [18] for specific applications. A similar approach was followed by [19], providing an API that allows to introduce taint tracking for legacy web applications without major code changes. This is also based on taint tracking at database level and uses hooks that are placed in legacy code to enable security policy enforcement.

4 Data Usage Control

From the use case in the previous section it becomes clear that a binary access decision in terms of *deny* or *permit* is not suitable and thus neither is access control. Also the ability to bind access decisions to obligations the data consumer has to fulfill, is only a prerequisite and the interesting question is how obligations must be modeled in order to regulate details of the data processing. We thus strive for a model which allows users to express gradual restrictions on how sensitive data may be used so that it is still possible to provide it in a form that allows third party services to operate, while at the same time it prevents misuse cases in which conclusions on personal preferences can be drawn.

4.1 Complex event processing as a generic way of data processing

In the context if this paper we will regard data processing as information retrieval from complex event processing (CEP) queries. Complex event processing is a technique to select patterns from continuously incoming streams of events. In that respect it is similar to querying a static relational database system (RDBMS), whereas the data is not persisted in tables, but rather provided in the form of temporary events which are only considered when relevant for the queries registered in the system.

We will decompose CEP queries into data flows and operate on the resulting data flow graph, which makes our approach applicable to all data analytics that can be mapped into a respective flow model. Referring to CEP as a processing technique has several advantages: first, it is highly relevant for practical use cases dealing with the analysis of large volumes of data records and is generic

[1] `http://www.sqlstream.com/`
[2] `http://www.espertech.com/`

enough to include traditional RDBMS queries, as well as Big Data algorithms like Map Reduce. Second, it allows us to integrate data usage control in a single confined query engine, rather than scattering it across different components.

Consider the following query on a user's location and activity data to identify the user's favorite run tracks within the last month:

```
SELECT avg(loc.location) FROM
  Health:win.time(1m)[activity='running'] as h,
  Location:win.time(1m) as l
 WHERE h.userid=l.userid
```

Here, two streams of incoming `health` and `location` events are expected, whereas both event types include a field `userid` which is used in the equi-join condition. Events from the `Health` stream are filtered to remove all events not referring to the activity of running. Finally, as not every single position is of interest, the 1-minute-average location of the relevant activities is calculated. While the query is very simple and would require significant tuning to reveal precise results in practice, it illustrates the concepts of complex event processing. The syntax corresponds to EPL, but it is not suited for our following discussion, as it contains syntactic sugar and disguises the actual semantics of the query. For example, the semantics of the SELECT clause, combines the concept of *selection* (select some $v \in V$) with that of *projection* (select an attribute) and aggregation (calculate average of event attributes).

We thus model the query in an abstract syntax which allows us to refer to well-defined semantics for each operation. As a general assumption, we regard queries as a mapping from sets of input streams of events to an output set of events. An event stream is again a set of events, where each event bears a timestamp so that a complete order over events by the time of their creation is possible. The mapping is achieved by concatenated *operations*, where we assume that each operation is free of side-effects, i.e. operations may create new virtual events streams but cannot insert, update, or remove events from an existing event stream.

Operations can perform statistical computations of data records, gather multiple events in *windows*, which are either defined by a certain time span in which data is collected, or filter out records immediately before any further processing is applied. Although actual set of operators is not predetermined by our approach but may be extended at runtime by plugins. we list some typical operations supported in our implementation in 1. Table 1 lists the main operations of the query language, where V denotes the set of events provided by an input stream, $\mathcal{C}$ a boolean condition (predicate) over events $v \in V$, and f arbitrary functions (whose semantics are not part of the query language considered here).

Note that both *select* and *window* realize a selection, but are represented as separate operations, because *select* operates on the defined attributes of events, while *window* operates on the implicitly defined time of event creation.

The query from above can now be rewritten using the abstract syntax:

Tab. 1: Data processing operations of the abstract syntax

Operation	Description
$V \leftarrow select(\mathcal{C}, V)$	Selection of events in set V where $\mathcal{C}(v) = 1$
$V \leftarrow projection(a, V)$	Selection of attribute $e.a$ of events $e \in V$
$V \leftarrow join(V_1, V_2, \mathcal{C})$	Returns a subset of Cartesian product defined by the join condition, i.e. $V = V_1 \times V_2$
$v \leftarrow avg(V)$	Average of the set V
$v \leftarrow count(V)$	Number of elements in V, i.e. $\|V\|$
$C \leftarrow window(\mathcal{T}, V)$	Selection of events in a time window specified by $\mathcal{T}$

```
projection(a,
 a ← avg (
       select (
        join(
           h←select(activity='running', window(t>1m, Health)),
           l←window(t>1m,Location) ),
         h.userid=l.userid )
       )
 )
```

4.2 MapReduce and CEP

If we use CEP as a processing technique, we can show that it is not necessarily limited to data streams, but is generic and flexible enough to also cover related Big Data processing approaches like MapReduce. If we take a look at the definition of MapReduce, we see the projection (Map) of a set of key-value-tuples to another set, followed by a consolidation (Reduce) to a list of target values:

$$Map : K \times V \rightarrow (L \times W)^* : [(k, v) \rightarrow [(l_1, x_1), ..., (l_{rk}, x_{rk})]$$
$$Reduce : L \times W^* \rightarrow X^* : (l, [y_1, ..., y_{sl}]) \rightarrow [w_1, ..., w_{ml}]$$

MapReduce is triggered to perform calculations on large data sets. Still, the data sets change over time and can be modeled accordingly. So a CEP query can be used to trigger MapReduce:

A continuous re-triggering of a MapReduce cycle leads to a less dynamic data stream, but one that can still be modeled by our approach.

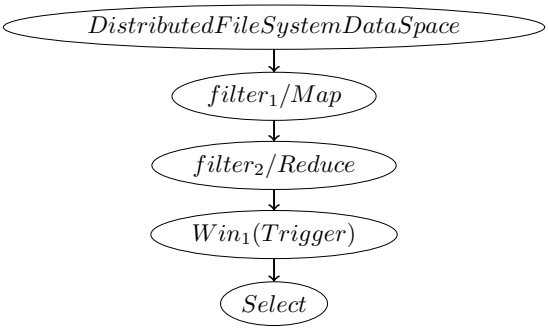

Fig. 2: CEP Query to control Map/Reduce

4.3 Data usage as a data flow problem

The model of formal operations from the previous section describes individual data processing steps, but does not say anything about how they are combined in a query. We thus extend it by a model which puts operations into order and formally describes queries in a way which allows to apply data usage policies to them.

The evaluation of a query can be regarded as a directed acyclic graph where nodes refer to individual operations and edges between two operations denote that one operation is applied to the output of another. This is not much different from the representation of a query plan which database systems use internally for the sake of optimizing the execution order of operations to speed up evaluation performance. In contrast to a query plan, our goal is however not to restructure the graph into a semantically equivalent but more rapid execution order, but rather to provide a model to describe restrictions on the way how data may be processed.

Stating a query in form of a data flow graph of operations allows us to treat the evaluation of a query as a single path through that graph. This path can then be verified against conditions which must hold along each query evaluation and are set by user-defined policies.

Figure 3 illustrates the flow graph of the example query from the previous section. It has two entry nodes as the query retrieves data from two different sources (i.e., tables or streams) and passes them on through a selection, two windows whose results are joined in a merge, and finally passed on through three further operations.

Pretschner et al. [12] represent data flows as a Kripke structure, which allows to state policies in temporal logic and apply model checking to verify them against the data flow model. While this is a straight-forward theoretical concept, PSPACE-hard model checking as a policy evaluation mechanism is inefficient in practice [20]. We thus do not go that route, as we aim at efficiently detecting policy violations at runtime using dynamic taint analysis. For this purpose, representing data flows in the form of flow equations is more useful.

A query is modeled as a flow graph $G(V, I, E, \tau, \mathcal{L})$, where

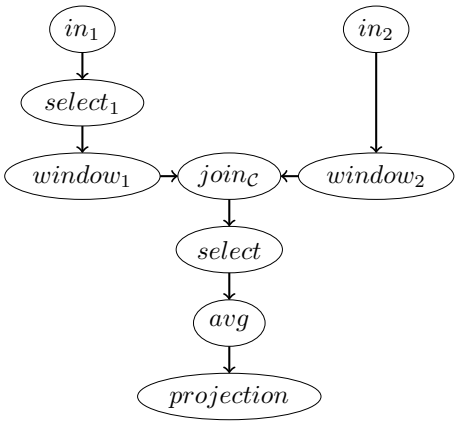

Fig. 3: Query evaluation as flow of data between operations

V: set of nodes representing atomic steps in the query evaluation. In each step, a single operation $o \in OP$ is applied to incoming events.
$I \subseteq V$ is a set of entry nodes providing the input event for a query evaluation.
$E : V \times V$: set of edges indicating that data is passed from one node to another.

Each node v may receive input events from it predecessor, denoted by the set $IN(v)$ and $OUT(v)$ denotes the set of output events of each node, respectively. Further, $GEN(v)$ denotes the events newly created in v and $KILL$ denotes deleted events. The flow of data through the query evaluation graph is consequently described by the following flow equations:

$$IN(v) = \bigcup_{p \in pred(v)} OUT(p)$$

$$OUT(v) = GEN(v) \cup [IN(v) \setminus KILL(v)]$$

The specific definition of $GEN(v)$ and $KILL(v)$ depends on the operation executed in v. For example, operation *in* in the two entry nodes in Figure 3 will create a GEN set containing all events recorded from the underlying event streams and will leave $KILL$ empty. The *select* operation in the subsequent node will leave GEN empty and instead add all events to $KILL$ which do not match the selection criterion.

In order to label events with respect to their criticality, we introduce a set of *taint labels* $\mathcal{L}$. In the simplest case, $\mathcal{L}$ can be $\{0, 1\}$, indicating whether an event is considered critical or not, but also more complex label sets are conceivable, for instance to indicate the type of labeled data (e.g., $\{Identifier, Location, Medical, ...\}$). A *taint propagation logic* $\tau_{op} : Events \rightarrow 2^{\mathcal{L}}$ assigns a set of taint states to events, depending on the taint state of the input events and the operation *op* applied to them.

4.4 Taint propagation for policy enforcement

The data flow model allows data usage control in the form of a dynamic taint analysis (DTA) [21]. This technique is traditionally used in program analysis to detect data leaks and lack of input sanitizations at runtime. Some programming languages such as Perl[3] include concepts for tainting variables and debugging tools like Valgrind are able to provide specific runtime environments for taint tracking[4]. Although the concept of DTA is clear and mature, its application in the context of program analysis comes with its own challenges. For example, overtainting is a common problem, which leads to large memory portions of a process being tainted, for example caused by spreading the taint state over globally accessible static variables. Also determining a precise and correct taint propagation logic can be difficult in some cases – for example a tainted integer being used as index in an array write operation, which may or may not be considered as tainting the overall array.

Fortunately, some of the problems from program analysis do not apply to our case, as we regard evaluations of data processing queries which have a reduced semantics and do not need to deal with global variables, threads, pointer arithmetic, and other border cases occurring in programs. As a consequence, much of the fuzziness of dynamic program taint analysis is removed in our case and the flow model captures much more precisely the semantics of the actual data propagation, in contrast to program analysis where this is only achieved approximatively. The approach is however similar, in that we track taint states of individual events during execution of the query and apply a taint propagation logic to transfer taint states from one event to another where appropriate. When data is about to leave the query evaluation, i.e. it ends up in OUT_{exit} of the exit node, the taint state of all respective events is checked against a user-defined policy. Depending on the policy, tainted events violating the rules will either be removed from OUT_{exit} or replaced by a sanitized version.

Policies are discussed in detail in section 5. It is however important to note that the taint propagation logic and the user-defined policy are distinct. The taint propagation logic determines how taint flags are propagated as operations are applied to set of events. For each operation, a respective taint propagation function τ_{op} is given, which may be configurable by a set of parameters. The policy classifies data sources with taint labels, determines unwanted data flows, and specifies counteractions to be taken when an unwanted data flow has taken place. For this purpose, it sets the parameters for the taint propagation logic.

For the sake of space we refrain from discussion the complete taint propagation logic here. The formulas given for the following operations should however give an impression of the taint semantics and the configurable parameters which can be set by the policy. Each row refers to a single operation and states the taint function which takes events from IN as input and sets the taint state of events in OUT.

The result of a $JOIN$ operation is the union of the taint states of all input

[3] `http://perldoc.perl.org/perlsec.html#Taint-mode`
[4] `https://github.com/wmkhoo/taintgrind`

Tab. 2: Taint propagation logic (parameters in bold)

Operation	**Taint function**
$JOIN(IN_1, IN_2, \mathcal{C})$	$\tau(v_1) \vee \tau(v_2)$ for $v_1 \in IN_1, v_2 \in IN_2, \mathcal{C}(v_1, v_2) = true$
$SELECT(v, IN)$	$\tau(v)$
$\tau(avg(V))$	$\bigvee_{v \in IN} \tau(v)$ for $\lvert IN \rvert > \boldsymbol{t}$
$\sigma(IN)$	$\bigvee_{v \in IN} \tau(V)$ for $\lvert IN \rvert > \boldsymbol{t}$
$count(IN)$	$\bigvee_{v \in IN} \tau(v)$ for $\lvert IN \rvert > \boldsymbol{t}$

events to the join. The $SELECT$ rule is similarly simple, keeping the taint states of the selected input event intact. For aggregation operations like *avg* and *count*, the taint propagation logic may be worth discussing. Whether the number of tainted events in a certain window should be regarded as tainted or not depends on the specific application and on the overall volume of events. In the context of this paper, the result of an aggregation function is regarded as tainted when the number of input events exceeds a certain threshold. This prevents a possibly malicious data user from retrieving individual data records by carefully crafting query attributes to narrow down the result set to a single record. Also, it ensures that data can only be retrieved at a certain level of aggregation, which is relevant in the case of industrial sensor data, where raw records might reveal too much information about actual production processes. We acknowledge however that the choice of a taint policy is not universally feasible and might differ in other applications.

5 Data usage control policies

Making use of dynamic taint analysis weaved into the query evaluation of CEP queries requires that users must be able to express their requirements in terms of a policy. The policy determines the configurable parameters of the taint propagation logic, taint flags, data sources, and countermeasures to be taken when the requirements are violated. The language is thus orthogonal to existing languages like the traditional usage control policy languages like the $UCON_{ABC}$ model, OSL [22], or even the Obligation element of XACML [23]. While these languages define elements for expressing obligations for a data user, our language extends on this and provides a way of stating restrictions on data usage, which is a specific form of obligation.

A policy consists of a set of rules, which are processed in the order they are specified. The processing is not limited to a single rule, but it is perfectly valid to have multiple rules sequentially applied to the same data record. The

definition of data source does however avoid multiple rules from accidentally overwriting each other, as it would be necessary to include the outputs of one rule in the inputs of another rule to apply both rules likewise to the same data record.

```
Rule: 'rule' name=ID
      r=TaintStmt | ActionStmt;

TaintStmt: 'mark' (dataSource+=DataType)+
           'with' (tFlags+=TaintType ','?)+
            untaint+=AllowStmt*;

AllowStmt: 'allow' op=Operation
           'with' (arg+=Parameter ','?)+;

ActionStmt: 'where' 'type' dataType=DataType
             'action' ('block'| ('sanitize'
                 ↪ f=Function | l=STRING));

Function: name=ID;
DataType: name=ID;
Operation: name=ID;
TaintType: name=ID;
```

Listing 1: XText grammar for data usage control policy

A rule is uniquely identified by an *ID*, chosen by the user. It defines a set of *dataSource* elements denoting the name of the considered input streams (or, in the case of relational databases, names of tables) and the taint flags assigned to them. Further, the *allow* element denotes operations which will untaint an event and sets the parameters for it, which are defined in the taint propagation logic. This is where parameter t for the aggregation operations from logic given in Table 2 is set. Parameters have names to make them more understandable for the user, such as *min_count* or *min_time_window* to denote a minimal number of events to untaint the result of the aggregation function or a minimal time frame, respectively. By an *action* statement, the user can control the countermeasures when a tainted event arrives at an exit node. The action is either dropping the event and removing it from the output (*block*) or a sanitization, which replaces the respective event with either a fixed literal or the result of an external function which takes the current event as input argument.

Referring to the example scenario from above, a policy to require that data may only be accessed at a certain level of aggregation would look as follows:

```
rule avgHealthAllow
     mark Health with T_PERSONAL
     allow avg with min_count=60, min_time=60
rule avgLocationAllow
     mark Location with T_PERSONAL
     allow avg with min_count=60, min_time=60
rule blockPersonalData
     where type T_PERSONAL
     action block
```

Listing 2: Policy for example scenario

```
rule avgHealthAllow
    mark  Health with T_PERSONAL
    allow avg with min_count="60", min_time="60"

rule avgLocationAllow
    mark  Location with T_PERSONAL
    allow avg with min_count="60", min_time="60"

rule blockPersonalData
    where type T_PERSONAL
    action block

tainttype T_PERSONAL
```

Fig. 4: Example policy in Eclipse IDE

This policy marks both data source 'Health' and 'Location' with a taint flag T_PERSONAL and declares that the taint flag will be removed if more than 60 records within 60 seconds are aggregated by the `avg` operation. If this has not been the case and a record marked with T_PERSONAL arrives at the exit node of the query evaluation, the record will be blocked, i.e. it will silently be removed from the output.

6 Prototype

For the evaluation of our prototype implementation, we collect data by a provider component and retrieve it by a consumer component through a CEP query interface. All components are realized as OSGi services and may be distributed in a network, though performance tests have been done with a local setup to rule out network latency. Data is represented by events, which are simple POJO Java beans. The provider component feeds them to different event streams where each stream corresponds to one type of event. As for complex event processing, the provider component uses the Esper[5] engine and provides interfaces for registering queries to the consumer. The consumer discovers the provider component and registers a CEP query, along with a callback function which is invoked when results for the respective query are available.

The data usage control mechanism consists of an extension on the Esper query engine in order to track taint states of events (or event attributes) during evaluation, along with a policy language for setting parameters of the taint analysis. For the policy language, a domain specific language (DSL) has been created by means of an Xtext[6] grmmar, from which the Xtext framework generates an API for accessing the elements of the policy model. The parameters provided that way are read by the dynamic taint analysis weaved into the query engine. Figure 4 shows a screen shot of the policy DSL editor.

Esper provides an instrumentation mechanism which can be switched on for development builds and provides an architecture to hook into different steps of the query evaluation. As this mechanism is designed for debugging and not for taint analysis, further extensions of the hooking API had to be made. We

[5] `http://www.espertech.com/`
[6] `http://www.eclipse.org/Xtext/`

Tab. 3: Single query round trip performance (n=1,000)

Profile	**Mean (ms)**	σ
Original	0.6258	0.5596
Data Usage Monitoring	0.9360	0.6789

leverage the fact that events in Esper are immutable, i.e. it is guaranteed that a single `Event` object will not be deep copied or modified in the course of a query evaluation. This allows us to keep track of taint states by maintaining a hash table of event instances and taint states, which is continuously updated by the taint propagation logic at each step of the query evaluation. Before result data is passed to the consumer, the provider checks the taint state of all result events and looks up the policy for countermeasures to take if tainted events are contained in the output set. Only after the respective events have been removed (*block*) from the result set or blinded (*sanitize*), the provider returns the query result.

The footprint of our implementation is small and does not add significant overhead to a normal CEP set up. The overall set up comprising consumer, provider, and CEP engine sums up to about 5 MB. Our taint tracking extension to the Esper engine is about 3500 lines of code, whereas only a fraction of it is executed, depending on the specific operations in a query.

Further, we were interested in the performance overhead that is added by the dynamic taint analysis. Retrieval speed is crucial in data-heavy applications and it would be problematic if the integrated taint analysis would have a significant impact. We measured the round-trip time of a query, i.e. the time from the initial event entering the query evaluation to the final event being written in the result set. The actual blinding or removal of tainted events was not included in the measurement as simply removing an item from a hash table is negligible and the time required for blinding depends on the actual blinding function. Table 3 shows the mean time for 1,000 query evaluations with and without dynamic taint analysis. As can be seen, the dynamic taint analysis adds only 0.31 ms to the evaluation, which makes it applicable even in high-volume event processing applications.

Measuring memory consumption of the dynamic taint analysis is difficult, as queries with artificially large memory requirements would have to be created in order to generate notable impact over the overall fluctuations from the OSGi layer and garbage collection. We therefore discuss memory consumption only at a theoretical level. Memory consumption of the taint analysis grows linearly with the amount of tainted events maintained during a query evaluation. It must however be noted that not only events from input streams may be written to the taint table, but also new events generated at intermediate steps during the query evaluation. As events are immutable, each calculation of an aggregate function will generate a new event bearing its result. Although this may outdate previously calculated aggregate events which may consequently be removed from memory, the taint table will still hold a reference to them, thereby preventing their removal from memory until the query has been evaluated. That is, the

amount of required memory will scale with the amount of input events and the number of operations which generated new events are applied during the query evaluation. Obviously, more complex queries will thus require more memory, but especially queries requiring frequent re-calculation of aggregates will add to memory consumption. In absolute numbers however, we do not expect the taint analysis to have a relevant impact on the applicability of our data usage control system, as for each tainted event, merely its address (8 Byte) plus a bitmap of its taint states in the order of 1 Byte must be stored.

7 Conclusion

We presented a data usage control system which allows to restrict *how* sensitive data may be used – an approach which orthogonal to traditional access control and integrates into generic usage control models like UCON$_{ABC}$. The core idea of our approach is to model data usage as a data flow, to state restrictions on how data may be used by setting parameters of a taint propagation logic, and to enforce them by a dynamic taint analysis which operates at the level of individual data records and is hooked into the original query engine. We proposed a domain-specific language for writing policies which are translated into parameters for the taint propagation logic and discussed the results of a prototype implementation based on a complex-event processing engine. From the evaluation of our implementation we conclude that the performance overhead is small enough to support real-world applications. Directions of further research are an abstraction of our policy language to support more high-level anonymitcy concepts like k-anonymity or l-diversity. Also, a further examination of over- and undertainting effects of the current approach and possible remedies is worthwile pursuing.

Acknowledgements

This work has been funded by the German Federal Ministry of Education and Research (BMBF) through the project InDaSpace – Digital Data Sovereignty.

References

[1] X. Zhang, F. Parisi-Presicce, R. Sandhu, and J. Park, "Formal model and policy specification of usage control," *ACM Transactions on Information and System Security (TISSEC)*, vol. 8, no. 4, pp. 351–387, Nov. 2005.

[2] J. Park and R. Sandhu, "The ucon abc usage control model," *ACM Transactions on Information and System Security (TISSEC)*, vol. 7, no. 1, pp. 128–174, 2004.

[3] M. Harvan and A. Pretschner, "State-based usage control enforcement with data flow tracking using system call interposition," in *Network and System*

Security, 2009. NSS '09. Third International Conference on, Oct 2009, pp. 373–380.

[4] G. Bai, L. Gu, T. Feng, Y. Guo, and X. Chen, "Context-aware usage control for android," 5 2012.

[5] L. Sweeney, "k-anonymity: A model for protecting privacy," *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, vol. 10, no. 05, pp. 557–570, 2002.

[6] A. Machanavajjhala, D. Kifer, J. Gehrke, and M. Venkitasubramaniam, "l-diversity: Privacy beyond k-anonymity," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 1, no. 1, p. 3, 2007.

[7] N. Li, T. Li, and S. Venkatasubramanian, "t-closeness: Privacy beyond k-anonymity and l-diversity," in *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*. IEEE, 2007, pp. 106–115.

[8] B. Katt, X. Zhang, R. Breu, M. Hafner, and J.-P. Seifert, "A general obligation model and continuity: enhanced policy enforcement engine for usage control," in *Proceedings of the 13th ACM symposium on Access control models and technologies*. ACM, 2008, pp. 123–132.

[9] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter, "A policy language for distributed usage control," in *Computer Security–ESORICS 2007*. Springer, 2007, pp. 531–546.

[10] D. Basin, F. Klaedtke, and S. MÃŒller, "Policy monitoring in first-order temporal logic," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, T. Touili, B. Cook, and P. Jackson, Eds. Springer Berlin Heidelberg, 2010, vol. 6174, pp. 1–18.

[11] D. Basin, M. Harvan, F. Klaedtke, and E. Zălinescu, "Monpoly: Monitoring usage-control policies," in *Proceedings of the Second International Conference on Runtime Verification*, ser. RV'11. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 360–364.

[12] A. Pretschner, J. Ruesch, C. Schaefer, and T. Walter, "Formal analyses of usage control policies," in *Availability, Reliability and Security, 2009. ARES '09. International Conference on*, March 2009, pp. 98–105.

[13] Y. Elrakaiby and J. Pang, "Dynamic analysis of usage control policies," in *11th International Conference on Security and Cryptography (SECRYPT)*, Vienna, Austria, Nov. 2014, pp. 88–100.

[14] A. Pretschner, M. Hilty, and D. Basin, "Distributed usage control," *Communications of the ACM*, vol. 49, no. 9, pp. 39–44, 2006.

[15] X. Zhang, J.-P. Seifert, and R. Sandhu, "Security enforcement model for distributed usage control," in *Sensor Networks, Ubiquitous and Trustworthy Computing, 2008. SUTC'08. IEEE International Conference on.* IEEE, 2008, pp. 10–18.

[16] A. Pretschner, M. Büchler, M. Harvan, C. Schaefer, and T. Walter, "Usage control enforcement with data flow tracking for x11," in *Proc. 5th Intl. Workshop on Security and Trust Management*, 2009, pp. 124–137.

[17] A. Arasu, S. Babu, and J. Widom, "The cql continuous query language: Semantic foundations and query execution," *The VLDB Journal*, vol. 15, no. 2, pp. 121–142, Jun. 2006.

[18] B. Davis and H. Chen, "Dbtaint: cross-application information flow tracking via databases," *Proc. of WebApps*, vol. 10, 2010.

[19] G. Chinis, P. Pratikakis, S. Ioannidis, and E. Athanasopoulos, "Practical information flow for legacy web applications," in *Proceedings of the 8th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems.* ACM, 2013, pp. 17–28.

[20] P. Schnoebelen, "The complexity of temporal logic model checking," in *Advances in Modal Logic*, vol. 4. World Scientific Publishing Co., 2002, pp. 1–44.

[21] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331.

[22] M. Hilty, A. Pretschner, D. Basin, C. Schaefer, and T. Walter, "A policy language for distributed usage control," in *Computer Security (ESORICS).* Springer, 2007, pp. 531–546.

[23] OASIS XACML TC, "eXtensible Access Control Markup Language (XACML) Version 3.0," OASIS Standard, Jan. 2013.