On Quantitative Dynamic Data Flow Tracking

Enrico Lovat, Johan Oudinet, Alexander Pretschner
Technische Universität München, Germany
{lovat,oudinet,pretschn}@in.tum.de

ABSTRACT

We present a non-probabilistic model for dynamic quantitative data flow tracking. Estimations of the amount of data stored in a particular representation at runtime — a file, a window, a network packet — enable the adoption of fine-grained policies which authorize or prohibit partial leaks of data. We prove the correctness of the estimations, provide an implementation that we evaluate w.r.t. precision and performance, and analyze one instantiation at the OS level.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

Keywords

Information flow; runtime monitoring; usage control

1. INTRODUCTION

Information flow tracking can be used to support access and usage control [29]. In order to enforce real-world usage control requirements on data, one must take into account that data exist in multiple representations. This requires tracking the data flow from one representation to another within and across different abstraction layers [29]. For example, a picture to be protected may be at the same time a file on the disk (operating system (OS) level), a browser object (application level) and/or a set of pixels on the screen (window manager level). It is therefore convenient to distinguish between abstract data (e.g., a picture) and concrete representations of data at different levels of abstraction, which we also call *containers* (e.g., a file at the OS level, a record at the database level, a set of pixels at the windowing level). Using this model, one can track the distribution of data among different containers in the system at any time and, using such information, enforce advanced usage control requirements such as "every copy of this data item must be deleted in 30 days." However, the information this model can offer about a certain container c and a certain datum d is binary: either c potentially contains (possibly partially) d, or it does not, similarly to the notion of *tainting* in information flow analyses.

CODASPY'14, March 3–5, 2014, San Antonio, Texas, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2278-2/14/03...$15.00. http://dx.doi.org/10.1145/2557547.2557551
.

We propose a refinement of this model that keeps track of the (estimated) *amount* of data that flows to a container. We do not limit our analysis to whether or not data flowed, but answer the question of *how much data flowed?*

The benefit of this extension is the ability to express and enforce *quantity-based policies*, such as *if more than 5% of customer data is stored in a file, then that file must not leave the system and must be deleted on log out.*

Quantitative measurements are useful for preventive enforcement but also for *a posteriori* analyses. Consider a company's software querying a customer database and the policy *access to these data allowed during opening hours only.* Without preventive enforcement mechanisms, employees might take home sensitive data. Often, this is the only way to get the job done in time, in violation of the policy. For auditing purposes, it should then at least be recorded how much sensitive data have been disclosed. An auditor spotting a policy violation that concerns only 0.01% of customer data once a month may decide to ignore it, while daily disclosure, or disclosure of 30% of data, must be sanctioned.

This concept of *acceptable exception* is very important in security in practice. Even though confidential data should never be disclosed, this restriction is occasionally circumvented to accomplish specific tasks. While quantifying the threshold between an acceptable exception and a violation to report depends on the context-specific security goal (e.g., cost in case of disclosure), thanks to Quantitative Data Flow Tracking (QDFT) it is possible to measure quantitative flows of data and to enforce rules on them.

Research problem We study how to perform quantitative data flow measurements and how to incorporate these measurements into a data usage control framework.

Solution and contribution We define a formal model for dynamic quantitative information flow measurements; implement it; and integrate it into a usage control framework, thus allowing the specification and enforcement of *quantity-based* policies. We see our contribution in a generic model for quantitative data flows that can be instantiated to different layers of abstraction and that uses a non-probabilistic layer-specific quantitative measure for data (*units*). We provide an exemplary instantiation at the OS level.

Structure After an introduction to measuring data in § 2, § 3 presents our model for measuring data flow quantities. § 4 sketches quantitative policies. § 5 describes an experimental evaluation. § 6 puts our work in context. We discuss the underlying assumptions and some perspectives in § 7. Future work and conclusions are left to § 8.

Running example Alice works as an analyst for a smartphone manufacturer. All sensitive projects are stored in a central repository, which is accessible by each client machine, like Alice's. She regularly requires information about new models under development. Her job includes combining it with data from field experiments and from various public sources into reports for suppliers and for other departments. To prevent leakage of sensitive data, each enterprise machine implements measures such as forbidding the installation of third-party software. However, too restrictive measures, such as preventing sensitive data from being saved locally, proved not to be very effective in the past. This is because they slow down the business process too much and sometimes were circumvented on a regular basis. For this reason, each machine is equipped with an OS-level monitor that tracks the amount of sensitive data that is processed by each system call. This system aims at the enforcement of policies such as: *if a file contains more than 1MB of sensitive data, then it must be saved in encrypted form and may not be emailed.* Such a policy allows Alice to send to a supplier some details about a specific smartphone, like for example size, weight or screen resolution, but prevents her from disclosing too many details such as a high-resolution pictures of the circuit board. In order to prevent violations of the policy by splitting data in multiple different files and mails, the aggregated number of chunks mailed to the same destination is recorded and stored for a reasonable amount of time, e.g., until the phone is publicly released.

2. MEASURING DATA QUANTITIES

In the rest of this paper we use the following terminology: a *data item* is a representation-independent abstract content that we want to protect (e.g., a phone specification). Data (items) are abstract concepts that exist in the system only in the form of layer-specific representations, called *containers* (e.g., a file, a window, a database record). For example, if picture P is stored in file F, then we say that F is a (partial) representation of P or equivalently that container F (partially) contains data item P. Note that one container may contain more than one data item (e.g., one single file may contain the specifications of two phones).

With our model, we want to track the distribution of *data* across different *containers*. If we know that data are stored in one particular container, and an action (e.g., a copy command or a query) transfers half of the content of the container into a new one, intuition suggests that also half of the data is stored in the new representation. Thus, our model refines the so-called *tainting approach* (yes, data are stored in this container / no, they are not) by the notion of *quantity* (50% of data are stored in this representation). Given a specific container, such as a file or a database, we want to know *how much* different sensitive data are stored in it. We can then enforce policies such as *if more than 1MB of data related to a phone specification are stored in a file, then that file must be encrypted and deleted on log out.*

We define as *data unit* the smallest part of a container that can be addressed by an event of the system, like a system call or a query, and as *size* of a container the number of units that compose it. The size of an event is the number of units the event processes. Units may differ depending on the level of instantiation. For example, at the database level, records (or cells) are units ("database d.db contains 14 sensitive records") whereas at the file system level, containers are measured in bytes or blocks, depending on the granularity of the events that operate on them ("file f.doc contains 150KB of sensitive data"). If data flow across different layers, units at one layer must be converted into units of the other layer. We get back to conversion in § 7.

The goal of this work is to *estimate how many different* units of sensitive data are stored in a specific container. Multiple copies of the same unit stored in the same container do not make the container more sensitive than a single copy: if a document contains the same paragraph twice, it contains as much information as a document with only one instance of this paragraph. Given a data source (the initial representation), we want to know how many different units of this data item are stored in each container of the system at a specific moment in time. If a container c contains q units of data d, then, by looking at c, one may come to know up to q different units of (the initial representation of) d.

We assume each unit to be as informative as any other unit, i.e., no unit is more important than another. If this assumption cannot be justified, then we may assume that the data items are structured into parts of different informative value, and we separately cater to the different parts of the structure. We do not consider covert or side-channels (e.g., knowing whether a unit has been copied or not does not leak any information about its actual value). Compression and/or different encoding schemas conflict with the assumption of informative equality of units. However, in closed or semi-closed environments as in our example, it is reasonable to assume the behavior of every single event or process in the system to be known, and therefore also knowledge of where compression takes place. For simplicity's sake, we do not consider this case in our solution but discuss it in § 7.

Knowing *how many* data units are stored in a container does not necessarily imply knowledge of *which* data units. Knowing exactly which units of data are transferred requires monitoring and storage capabilities that may not be available in every scenario. For example, in a database system, it may be reasonable to log the *number* of records retrieved by a query (size of the result table), but not to log the *complete content* of each query (values in the result table).

The precision of our quantitative approach is in-between coarse-grained and fine-grained tainting approaches. Coarse-grained tainting tracks data at the level of containers: if one data unit flows to a container, the whole container is considered as sensitive as if the complete data item had been transferred to this container. Fine-grained tainting tracks each data unit independently and computes exactly which data unit is stored in each container. Our quantitative method records the size of each event and infers that the destination container of a transfer operation of size n contains at most n different data units, *but not which ones.* If tracking every data unit is not possible or requires too many storage or computational resources, the quantitative approach can be a good compromise between security and usability.

3. QDFT MODEL

We now describe how to compute for every container, at each moment in time, an upper bound for the number of different units of a sensitive data item in this container. This is done on the grounds of a quantitative data flow model.

The model relies on three abstractions: data, containers, and events. Three types of events affect the amount of data in a container: container initializations, transfers of data units from one container to another (e.g., copying or appending parts of a file), and deletion of data from a container (e.g., deleting some records from a database table). These abstract events need to be instantiated according to the instantiation of the model (e.g., system calls for an OS, queries for a database).

Initially, each container in the system contains no unit of sensitive data. If a new sensitive data item is introduced into the system, this event is modeled by mapping the number of sensitive units in this data item to its initial representation(s). Usually, the initial amount of sensitive data corresponds to the size of the initial representation, but we allow for the case where a data item is only partially sensitive, i.e., the number of sensitive units may be strictly smaller than the size of the initial representation.

After the initialization of a data item d, every event in the system that corresponds to a transfer or deletion of units of d in some container is monitored. We will show how different events lead to different upper bounds for the number of sensitive units in each container.

The amount of sensitive data transferred by an event is bounded by the amount of sensitive data stored in the source container. This bound can sometimes be improved. Consider two transfers of 6 units each from a container A of size 10 to a new container B. Simply adding units to the destination after every transfer would assign to B an upper bound of 12, even though A is bounded by 10 (hence no more than 10 different units could have flowed from A to B).

A similar observation can be made if a container receives data from multiple related sources. Consider a scenario where 6 units are transferred from A of size 10 to B. An additional transfer of 6 units from A to C and a subsequent transfer of 6 units from C to B would lead to the same result of B containing 12 different units—but all the units in B originate in A of size 10. To increase precision, we hence need to keep track of previous transfers. This historic information is stored in a *provenance graph.*

Provenance graphs For simplicity's sake, we assume that there is only one data item d in this section. If there is more than one item, we need more than one provenance graph, which we discuss in § 4. Data provenance is recorded by a flow graph $\mathcal{G} = (\mathcal{N}, \mathcal{E})$. Each node $n \in \mathcal{N}$ in this graph represents a container at a specific moment in time, encoded by a natural number: $\mathcal{N} \subseteq C \times \mathbb{N}$, where C is the set of containers. We assume the existence of a special external source node $(S, 0) \in \mathcal{N}$ for the data item for which the graph is built. Edges $\mathcal{E} \subseteq \mathcal{N} \times \mathbb{N} \times \mathcal{N}$ represent events that add data to or remove data from containers. The label of an edge is the actual amount (number of units) of data that flowed from the source container to the destination container in the considered event. As a consequence, it also is an upper bound for the amount of *sensitive* data units that flowed from the source node's container to the destination node's container. Its precise value is determined by the event, e.g., 100 bytes or 10 records have been copied.

Goal We want to compute an upper bound for the number of different sensitive units of data item d in each container in the system. For reasons that will become apparent in step 6 of the example below, we do this on the grounds of a provenance graph. We will define a function κ that computes such an upper bound for those nodes of the provenance graph that correspond to the containers in the current (that is, latest) time step.

Construction at runtime A provenance graph is incrementally built at runtime. This gives rise to a sequence of graphs $\mathcal{G}_0, \ldots, \mathcal{G}_t$ for each moment in time, t. Let $\mathcal{G}_t = (\mathcal{N}_t, \mathcal{E}_t)$ for all t. For a node $n \in \mathcal{N}_t$, let $\kappa(n)$ be the above mentioned upper bound for the maximum number of sensitive units in the container that corresponds to the node (the computation of κ will be discussed below). For every event at time t, at most one new node as well as one or two new edges are created. This evolves $\mathcal{G}_{t-1}$ into $\mathcal{G}_t$ where $\mathcal{G}_{t-1}$ is a subgraph of $\mathcal{G}_t$. Possible graph evolutions are the following:

I step The *initialization* of data is modeled by copying all data from the data source node to an initial container c_i. Assuming the event at time t is the initialization, we add a node (c_i, t) to $\mathcal{N}_{t-1}$ and, for m units of sensitive data contained in data item d, an edge $\big((S, 0), m, (c_i, t)\big)$ to $\mathcal{E}_{t-1}$, which yields $\mathcal{G}_t = (\mathcal{N}_t, \mathcal{E}_t)$.

C steps If, at time t, container c_1 may contain some sensitive units and the event *copies*, without knowledge of whether or not they are sensitive, ℓ units of data from container c_1 to c_2, then we add a node (c_2, t) and the first or both of the following two edges to $\mathcal{G}_{t-1}$:

- **C1 step** $\big((c_1, t'), \ell, (c_2, t)\big)$ for the node $(c_1, t') \in \mathcal{N}_{t-1}$ with $t' < t$ such that there is no $(c_1, t'') \in \mathcal{N}_{t-1}$ with $t' < t'' < t$ (this ensures that we copy data from the "latest representation" of container c_1).
- **C2 step** If c_2 already exists (copying then is appending), we also need to consider its content at time $t - 1$. In this case, there is a $t' < t$ such that $(c_2, t') \in \mathcal{N}_{t-1}$ and there is no t'' with $t' < t'' < t$ such that $(c_2, t'') \in \mathcal{N}_{t-1}$. To make sure that the sensitive content of c_2 at time t' (which is the same as at $t-1$) is not forgotten at time t, we add the edge $\big((c_2, t'), \kappa((c_2, t')), (c_2, t)\big)$ to $\mathcal{E}_{t-1}$. Replacing the label $\kappa((c_2, t'))$ by ∞ would not impact correctness nor precision; our choice of $\kappa((c_2, t'))$ is motivated by presentation concerns. See the introductory remark in the appendix.[1]

T step If the event at time t is a *truncation* of c, we need to compare the amount of sensitive data in c (i.e., $\kappa((c, t'))$ such that $(c, t') \in \mathcal{N}_{t-1}$ and $(c, t'') \in \mathcal{N}_{t-1} \Rightarrow t'' \le t'$) to the new size of c, m. Since a container cannot store more sensitive data than its actual size, if $\kappa((c, t')) > m$, we add a node (c, t) and an edge $\big((c, t'), m, (c, t)\big)$ to $\mathcal{G}_{t-1}$. Otherwise, no new node is added.

Example Before we explain how to compute κ, consider the example in Figure 1.

1. First, container A is initialized with the content from the external data source (an I step). This yields node $(A, 1)$ as well as the edge labelled 20 from $(S, 0)$ to $(A, 1)$, indicating that the data item in question contains 20 units of sensitive data (and possibly more non-sensitive data). The only useful estimation is $\kappa((A, 1)) = 20$. Note that this only measures the number of sensitive units; similar to S, A may well contain more non-sensitive units of data.

[1] Note that a C step can be performed with more than one source container. This is modeled as a C1 step for each source container and only one C2 step if the destination container already exists.

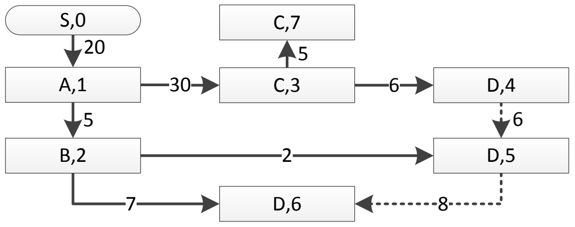

Figure 1: Example provenance graph at time $t = 6$. Dashed arrows are C2 steps.

2. In the second step, 5 units of data are copied from A to B, which results in node $(B, 2)$ and the edge labelled 5 from $(A, 1)$ to $(B, 2)$ (a C1 step). Since only five units of data have been moved, $\kappa((B, 2)) = 5$ is reasonable.
3. The third event copies 30 units of data from container A to container C (another C1 step), resulting in an edge with label 30 from node $(A, 1)$ to a newly created node $(C, 3)$. Note that it is possible to copy 30 units because the size of A may well exceed the number of sensitive units. Still, since A contains at most 20 units of sensitive data, 20 is also an upper bound for the number of sensitive units in C, yielding $\kappa((C, 3)) = 20$.
4. The fourth event copies 6 units from C to D (another C1 step). This yields a new node $(D, 4)$ and an edge labelled 6 from node $(C, 3)$ to $(D, 4)$. We know that C contains at most 20 sensitive units and have hence to assume that all data copied from C to D is sensitive. We estimate $\kappa((D, 4)) = 6$.
5. The fifth event copies 2 units from B to D (a C1&C2 step). This results in a new node $(D, 5)$, an edge labelled 2 from $(B, 2)$ to $(D, 5)$ and, in order to reflect the content previously contained in D, another new edge from $(D, 4)$ to $(D, 5)$ labelled 6 (which is the κ value of $(D, 4)$). The maximum number of sensitive units in D now is $\kappa((D, 5)) = 8$, the sum of at most 2 units from B and at most 6 units from the earlier instance of D.
6. The sixth event copies another 7 units from B to D (another C1&C2 step). We create a new edge labelled 7 from $(B, 2)$ to a new node $(D, 6)$, and an edge labelled $\kappa((D, 5)) = 8$ from $(D, 5)$ to $(D, 6)$. This is where the computation of a precise value for κ becomes non-trivial. A first estimate for the upper bound is $\kappa((D, 6)) = 8 + 7$, a result of the flows from B and an earlier version of D to D, similar to what happened in step 5. However, we know that the actual flows are upper bounds for flows of sensitive data, and since $\kappa((B, 2)) = 5$ and $\kappa((D, 5)) = 8$, it is impossible to have more than 13 distinct sensitive units in D. A better estimate is hence $\kappa((D, 6)) = 5 + 8$. Yet, it is impossible that 13 *different* units flowed from A to D: The upper bound of $\kappa((D, 5)) = 8$ units of sensitive data includes 2 units received from B (step 5). Since we do not need to count these units twice, D cannot contain more than $\kappa((D, 6)) = 5 + 6$ different sensitive units.
7. The seventh event truncates the size of container C to the new size of 5. This results in a new node $(C, 7)$ and an edge labelled 5 from $(C, 3)$ to $(C, 7)$. $\kappa((C, 7)) = 5$, because C is now composed of 5 units only, i.e. C contains at most 5 different sensitive units.

Motivation for provenance graphs Step 6 of the example motivates the use of provenance graphs: The fact that $(D, 6)$ necessarily contains duplicates of two units of sensitive data is something that we can know only by considering the *history* of data flows (in this case, step 5 that appended two units of data to container D—and the transfer of sensitive data in step 5 depends on steps 2 and 4, as we will see below). It is this historical knowledge that allows us to reduce the upper bound $\kappa((D, 6))$ from 13 to 11 units.

Motivation for max-flow/min-cut In fact, *the number of units transferred from* $(B, 2)$ *to* $(D, 5)$ *(i.e., the 6th event) is not the only relevant influence for the computation of* $\kappa((D, 6))$; regardless of the number of units that had been transferred from B to D in step 5, we would still have an upper bound of at most $\kappa((D, 6)) = 11$. From this perspective, the provenance graph now reveals that $(B, 2)$ and $(D, 5)$ together cannot contain more than 5+6 units of sensitive data: historically, B received 5 units in step 2 and D received 6 units in step 4. *This information is stored in the form of the edge labels.* This motivates the computation of κ via the max-flow/min-cut theorem rather than via recursive computations on predecessors or dominators of the newly created nodes as follows. Remember that the *actual* flows of possibly non-sensitive data in-between containers (the edge labels) are *upper bounds* for the flow of *sensitive* data. We can hence interpret them as *capacities for sensitive data* in the flow graph. Then, the upper bound for the amount of different sensitive data in each container (corresponding to a node n) equals the maximum flow of sensitive data from the source node to this node n. In our example, the max flow to D in step 6 is determined by steps 2 and 4, corresponding to the edges from $(A, 1)$ to $(B, 2)$ and from $(C, 3)$ to $(D, 4)$.

Computation of κ The definition of κ then is simple: if $n \in \mathcal{N}_t$ is the new node created in time step t, then $\kappa(n) = \mathit{maxflow}_{\mathcal{G}_t}((S, 0), n)$. Since every container can be the destination of a copying event in the next step (a C step), we always need to know the current κ value of all nodes.

Algorithmically, we need to compute at most one max-flow/min-cut on a directed acyclic graph per event for the new node. This is because we never add incoming edges to existing nodes; their κ value is constant over time.

Correctness Our model is correct if, at each moment in time, the number of different units of a sensitive data item actually contained in a container is not greater than the κ value we have computed. We assume a function φ that provides this actual amount for a node (i.e., container and a moment in time). We do not say how to compute φ, we just assume its existence.

We need to assume that all events in the system are adequately reflected in the construction of the flow graph. Whenever an event moves data in-between containers, this event is used to construct the provenance graph as described above. Conversely, no edge or node is added if there is no corresponding event. This assumption connects the model (provenance graph) to the real system. Note that κ denotes a property of the model and φ a property of the real world.

The proof that $\forall\, t \in \mathbb{N}\, \forall\, c \in C : \varphi(c, t) \leq \kappa((c, t))$ indeed holds is provided in Appendix A.

Simplification Because the complexity of the computation of maximum flows depends on the size of the graph, it is desirable to keep graphs small. We are hence interested in reducing their size while maintaining correctness and precision of the algorithm presented above. Our simplification rules are motivated by the observation that a sequence of event can sometimes be shortened to another sequence that leads to a smaller provenance graph that provides the same upper bounds for every current and future container.

Intuitively, provenance graphs $\mathcal{G}_t$ and $\mathcal{G}'_t$ are equivalent if (i) for each container c, if n and n' are the most recent nodes created for c in $\mathcal{G}_t$ and $\mathcal{G}'_t$ respectively, then $\kappa(n) = \kappa'(n')$ where $\kappa'(n') = \mathit{maxflow}_{\mathcal{G}'_t}((S,0), n')$; and (ii) if the content of any set of containers (i.e., most recent nodes) is copied to a (possibly new) container after time t, then the evolutions of the two graphs yield the same upper bounds for the amount of sensitive data in this container. (i) stipulates that the two graphs yield identical upper bounds for every container. Since future evolutions of a provenance graph add edges to the respective most recent nodes only, (ii) stipulates that independently of the events that connect such nodes in the future, the maximum flows from the source to the new nodes must be identical in both graphs.

Formally, let $\mathit{cn}_{\mathcal{G}}(X)$, the *current nodes*, be the nodes in $\mathcal{G}$ that are the latest representation of each container in $X \subseteq C$: $\forall \mathcal{G} = (\mathcal{N}, \mathcal{E}), X \subseteq C : \mathit{cn}_{\mathcal{G}}(X) = \{(c,t) \mid c \in X \wedge (c,t) \in \mathcal{N} \wedge \forall t' : (c,t') \in \mathcal{N} \Rightarrow t' \leq t\}$. Assuming a set of containers X, $\mathcal{G}_{t,X}$ denotes the graph in which every node in $\mathit{cn}_{\mathcal{G}_t}(X)$ is connected to a dummy node dn that represents a virtual sink of all future operations on the nodes in $\mathit{cn}_{\mathcal{G}_t}(X)$ (and therefore all future evolutions of $\mathcal{G}_t$ on every possible set of containers): $\mathcal{G}_{t,X} = (\mathcal{N}_t \cup \{\mathit{dn}\}, \mathcal{E}_t \cup \bigcup_{n \in \mathit{cn}_{\mathcal{G}_t}(X)} \{(n, \infty, \mathit{dn})\})$. For a node n, let $\kappa_{t,X}(n) = \mathit{maxflow}_{\mathcal{G}_{t,X}}((S,0), n)$ and, similarly, $\kappa'_{t,X}(n) = \mathit{maxflow}_{\mathcal{G}'_{t,X}}((S,0), n)$. *Equivalence of provenance graphs* then reads as $\mathcal{G}_t \sim \mathcal{G}'_t \Leftrightarrow \forall X \subseteq C : \kappa_{t,X}(\mathit{dn}) = \kappa'_{t,X}(\mathit{dn})$.

We now describe several cases where we can obtain a smaller but equivalent provenance graph that will considerably increase performance in the experiments of § 5.3. We believe these to be the interesting cases but make no claim of completeness. The proof that these simplifications are *correct* in that the original provenance graph, $\mathcal{G}_t$, is equivalent to the modified graph, $\mathcal{G}'_t$ (i.e., $\forall t \in \mathbb{N} : \mathcal{G}_t \sim \mathcal{G}'_t$), is provided in Appendix B.

Removal of a truncation. If at time k there is a copying action of size q from c_a to a new container c_b, and a truncation of c_b to size m occurs at time $t > k$, and, between these two events, there is no other truncation or transfer to c_b, and the sum of transfers from c_b is lower than $q - m$, then this sequence leads to a provenance graph equivalent to the one yielded by the sequence that transfers exactly m data units to c_b at time k, copies data from c_a instead of c_b from time $k+1$ to $t-1$, and does not contain the truncation of c_b at time t. We hence modify the edge label of the copying step at time k, replace any further copying action from c_b by a copying action from c_a, and remove the final truncation.

Removal of a copy (case 1). If a copying action from c_a to c_b of size s_1 takes place at time k, and another copying action from c_c to c_b of size s_2 occurs at time $t > k$, and in between these two events there is no truncation or transfer to c_a or to/from c_b, then this sequence leads to a provenance graph that is equivalent to the one yielded by the sequence that directly transfers s_1 and s_2 data units at time k from c_a and c_c, respectively. We hence add c_c to the source containers of the copying at time k and remove the last copying step.

Removal of a copy (case 2). If at time k there is a copying action from c_a to a new container c_b of size s_1, and another copying action also from c_a to c_b of size s_2 occurs at time $t > k$, and in between these two events there is no truncation or transfer to c_a or c_b, and the sizes of all transfers from c_b after time k sum to a value that is below s_1, then this sequence leads to a provenance graph that is equivalent to the one yielded by the sequence that copies $s_1 + s_2$ data units at time k and does not contain the final transfer.

4. QUANTITATIVE POLICIES

Semantic Model We now use provenance graphs to generalize the model presented in [29] to combine QDFT with data usage control. In that model, possibilistic data flows are captured by sequences of states that map data items to their representations, i.e. containers. The considered systems are modeled as tuples $(P, \mathit{Data}, \mathit{Event}, C, \Sigma, \sigma_i, \varrho)$, where P are principals, Data are data items, Event are events happening in the system that are triggered by principals' actions, C are data containers, Σ are the states of the system with σ_i being the initial state ($\varnothing$), and ϱ is the state transition function. We first show how to compute a provenance graph for each data item at each moment in time and then use this definition for the specification of usage control policies that restrict a provenance graph's evolution.

States $\Sigma = \mathit{Data} \to \mathit{Graph}$ associate each data item with a provenance graph. Provenance graphs consist of nodes, $\mathit{Nodes} \subseteq C \times \mathbb{N}$, i.e., container-timestamp pairs. We require Nodes to contain a reserved identifier $(S, 0)$ that stands for the external source of data. Provenance graphs are modeled as a (partial) function of type $\mathit{Graph} = \mathit{Nodes} \times \mathit{Nodes} \to \mathbb{N}$ which associates each edge with the corresponding event's size, i.e., the number of flowed data items.

Given a provenance graph and an event, function $\mathit{step} : (\mathit{Graph} \times \mathit{Event}) \to \mathit{Graph}$ updates the graph according to the rules presented in § 3. We hence assume that each concrete event in the system is (1) mapped to one of the abstract events and therefore possible graph evolutions (init, transfer, truncation) and (2) associated with a size, possibly 0. Then, $\varrho : (\Sigma \times \mathit{Event}) \to \Sigma$ where $\varrho(\sigma, e) = \sigma'$ and $\sigma'(d) = \mathit{step}(\sigma(d), e)$ for each data item $d \in \mathrm{dom}(\sigma)$.

Quantitative usage control policies are defined over traces that map abstract time points to events. For simplicity's sake, we assume it is always possible to linearly sort events chronologically, and therefore only one event per timestep; $\mathit{Trace} : \mathbb{N} \to \mathit{Event}$. Then, given a trace tr, function $\mathit{states}_q : (\mathit{Trace} \times \mathbb{N}) \to \Sigma$ computes the information state at a given moment in time t via $\mathit{states}_q(\mathit{tr}, 0) = \sigma_i$ and $t > 0 \Rightarrow \mathit{states}_q(\mathit{tr}, t) = \varrho(\mathit{states}_q(\mathit{tr}, t-1), \mathit{tr}(t-1))$.

Policies Policies describe situations to be avoided or enforced. In usage control contexts, this is often specified by sequences of allowed/disallowed events or states (mappings from data to containers) using temporal logic formulae. We do not discuss general usage control policies here, since this has been done elsewhere [40, 29], but concentrate on one possible way of restricting data quantities. To this end, we need to capture function κ introduced in § 3. Given a graph $\mathcal{G}_t$ and a container c, function $K : (\mathit{Graph} \times C) \to \mathbb{N}$ returns the maximum amount of different sensitive units stored in c according to the provenance graph $\mathcal{G}_t$. K corresponds to the application of $\kappa()$ to the most recent version of c in $\mathcal{G}_t$. We then define one *state-based operator* ($\Phi_q ::= \mathit{atMostInSet}(\mathit{Data}, \mathbb{N}, \mathbb{P}(C))$) to specify quantitative policies: `atMostInSet(d, q, Cs)` limits the combined capacity of a set of containers. For example, the policy "No more than 1MB of customer data can be saved on a removable device" could be checked by the proposition `at-`

MostInSet(d,1MB, REMOVABLE), where REMOVABLE is the set of containers that represent files on removable devices. The policy "no more than 10MB may be sent over the network" is specified as atMostInSet(d,10MB,CNET) where CNET is the set of all network sockets. Lack of space prevents us from presenting other operators in Φ_q here, for instance, atMostInEach(d, q, Cs) with its intuitive semantics.
Given a trace and a time point, we define the semantics $\models_q \subseteq (\mathit{Trace} \times \mathbb{N}) \times \Phi_q$ of this quantitative data usage operator by $\forall\, tr \in \mathit{Trace}, \forall\, t \in \mathbb{N}, \forall\, \phi \in \Phi_q, \forall\, \sigma \in \Sigma \bullet (tr, t) \models_q \phi \Leftrightarrow$
$\sigma = \mathit{states}_q(tr, t) \wedge \exists\, d \in \mathit{Data}, \mathit{Cs} \subseteq C, Q \in \mathbb{N} \bullet$
$\phi = \mathit{atMostInSet}(d, Q, \mathit{Cs}) \wedge \sum_{c \in \mathit{Cs}} K(\sigma(d), c) \leq Q.$

Using Φ_q we can specify simple information flow policies only. In order to express more complex policies like those presented in our reference scenario in § 1, we need a more expressive policy language. For reasons of space, we do not describe here how to embed Φ_q into a full temporal logic language such as the ones defined in [29, 40]. However, by leveraging $\models_q$ and states_q, it is possible to define the semantics of a new language $\Phi \supset \Phi_q$ for the specification of quantitative policies that capture temporal and propositional operators as well. Using this language Φ, it is then possible to specify policies such as $\mathit{always}(\neg \mathit{atMostInSet}(d, q, \mathit{Cs}) \Rightarrow \mathit{notify})$ that issue a notification whenever the amount of data d stored in the specified set of containers Cs exceeds q. This kind of policies will be used in § 5.

5. EXPERIMENTAL RESULTS

We conducted two experiments to evaluate the adequacy of the QDFT model in terms of the scenario in § 1. First, we measured precision and performance of the model alone, i.e., at an abstract level. Then, we instantiated the model to the OS level, extending previous work [19] and refining the abstract events from the first experiment to system calls.

5.1 Implementation and methodology

Remember that Alice performs a sequence of report generation/ update actions in the phone scenario presented in § 1 and that she is subjected to a set of policies of the kind $\mathit{always}(\neg \mathit{atMostInSet}(d, 1MB, \mathit{MAIL}) \Rightarrow \mathit{notify})$ as introduced in § 4. An action is a transfer of some units of data from a specification or from an existing report to another (possibly new) report. We generated random sequences of actions of different lengths, and observed the evolution of our model during their execution. In each step, Alice can choose between creating a new and updating an existing report with probability *PN*. We ran 100 sequences for each length and used average and median values for the analyses in the remainder of this section.

In the first set of experiments (*atomic*, denoted by **A**), we modeled each action after the initialization as one single abstract event, either a T or a C step (see § 3), where specifications and reports are containers. In the second set of experiments (*syscalls*, denoted by **S**), we instantiated the model to the OS level and performed QDFT at this level. In this refined context, events are system calls, and containers are files, pipes, memory locations, and message queues. Specifically, the MAIL set from the above policy is a set of sockets for email communication. Specifications and reports are modeled as files, and each abstract action from the first set of experiments corresponds to one system call or a sequence of system calls: initialization is done in the beginning by extracting the data item d to be protected from the policy and mapping it to the initial representation; truncation is modeled as open() with the overwrite flag set, as truncate(), ftruncate(), or unlink(); transfer is modeled by mmap(), as read() from file to process, and as write() from process to file or socket. These are the concrete events used to define the ϱ relation in § 4. They also refine, by disjunction, the abstract events specified in the above policy.

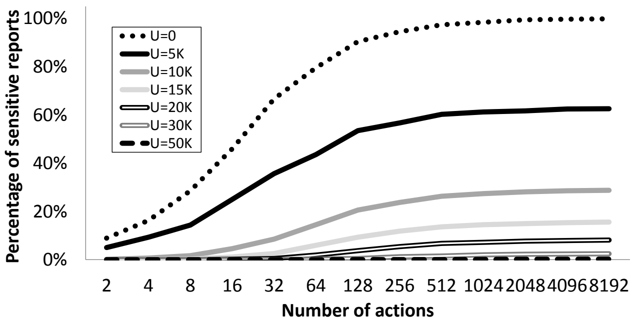

Figure 2: Percentage of reports considered sensitive by a quantitative policy vs. number of actions (PN=50%)

The length of an event sequence depends on the size of the files and of the transfers involved. In our tests, one abstract action corresponds, on average, to 40 system calls. We started each experiment from the same arbitrary yet fixed amount of (sensitive) specifications and (initially non-sensitive) reports. For both experiments, we evaluate our model with (denoted **s**) and without (denoted **n**) simplification. For the *syscalls* tests without simplification (i.e., the Sn configuration), we considered sequences of at most 1024 actions because the time required to perform 100 executions longer than 1024 is prohibitive without simplification.

For the second set of experiments, we extended an existing usage-control framework for system call interposition based on the Systrace tool [30]. The amount of bytes that a process tries to read/write from/to a file corresponds to the event size used by our tracking framework. If any of the defined policies does not allow the respective system call to be executed, the system call is denied. Otherwise, the system call is dispatched to the kernel and executed as usual.

We study the following questions: **(i)** How precise is our model (i.e., how far is the estimated value from the exact amount of different sensitive data units)? **(ii)** What is the overhead of QDFT w.r.t native execution?

5.2 Precision

One problem with possibilistic data flow tracking is usability: because of the involved over-approximations, systems quickly become unusable because very many data items are quickly tainted ("label creep"). For a typical run of the system, Figure 2 shows the relative number of reports considered sensitive (tainted) for the policy "do not distribute a report if it contains more than U units of sensitive data," for varying U (50K is the number of sensitive units in the system), as a function of the number of hitherto executed actions. The figure suggests that usability may indeed be increased because reports are considered sensitive less quickly and thus not blocked when sent, depending on the value of U. Note that all curves are well below the uppermost reference line $U = 0$ (possibilistic estimation).

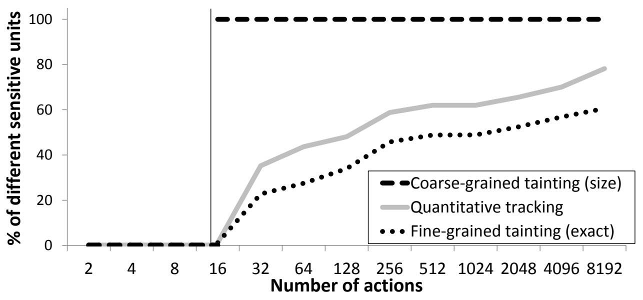

Figure 3: Estimation of different sensitive units in an exemplary container during one execution (PN=50%). Before the 16th action, no sensitive data is stored in the container.

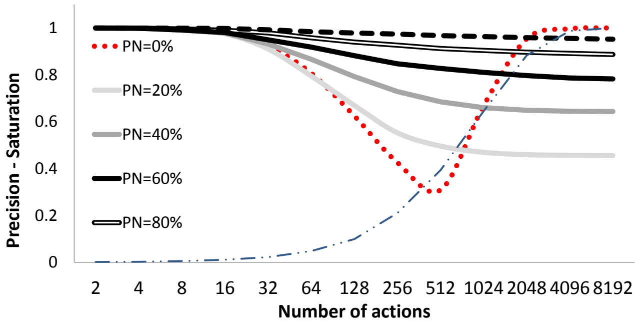

Figure 4: Median precision vs. number of events and percentage of new reports for *syscalls* tests

A second perspective is provided by Figure 3, which shows how many units are considered sensitive according to different tracking methods for one typical container. As expected, the quantitative estimation is in-between coarse-grained and fine-grained tainting.

To more precisely answer the first research question we want to know in general how close the quantitative tracking curve is to the fined-grained tainting curve; the smaller this gap is, the more precise our model is. We therefore empirically measure the precision of our model, for a container c at a specific moment in time, by letting $\mathit{Prec}(c) = (\mathit{Size}(c) - \mathit{Estim}(c))/(\mathit{Size}(c) - \mathit{Exact}(c))$ if $\mathit{Size}(c) \neq \mathit{Exact}(c)$, and 1 otherwise, where $\mathit{Size}(c)$ is the total number of data units in c (including non-sensitive data and duplicates), $\mathit{Exact}(c)$ is the precise number of different sensitive units in c, and $\mathit{Estim}(c)$ is the number computed by our model, i.e., the value of κ. Thus, $\mathit{Exact}(c) \leq \mathit{Estim}(c) \leq \mathit{Size}(c)$. The estimation is most accurate if $\mathit{Estim}(c) = \mathit{Exact}(c)$ (i.e., $\mathit{Prec}(c) = 1$), and the worst case ($\mathit{Prec}(c) = 0$) is $\mathit{Estim}(c) = \mathit{Size}(c)$. In the special case $\mathit{Size}(c) = \mathit{Exact}(c)$, our estimation is still correct (i.e., $\mathit{Prec}(c) = 1$). The proportion of non-sensitive units (or duplicates of sensitive units) in a container c that are incorrectly considered additional different sensitive units by our model corresponds to $1 - \mathit{Prec}(c)$.

Precision obviously depends on the sequence of actions considered. For instance, if the user always copies the complete content of one container to another, then the quantity-based approach will not be more precise than any coarse-grained tainting approach. Similarly, if the user always copies the same few specific units from a container that contains a lot of other sensitive units, the quantity-based approach will be significantly less precise than a fine-grained tainting approach. However, it is interesting to see how the precision evolves in scenarios in-between these two extremes.

Figure 4 shows the median precision for all containers after Alice performed a number of transfers. Since she can choose between creating a new report or updating an existing one, we varied the probability PN that she creates a new report in each step from 0, where no new report is created but only existing ones are updated, to 1, where every action creates a new report. Precision monotonically decreases for each value of PN but 0. For $PN = 0$, only updates of existing reports are possible: Alice keeps transferring data to and in-between the same fixed set of reports, with the consequence that after a while all data in the specifications is transferred to every report. Let max be the maximum number of different sensitive units in the specifications (and thus, in the system). If a report contains all specification data ($\mathit{Exact}(c) = max$), we call it *saturated*. Considering that max is an upper bound for our estimation, precision for saturated containers is equal to 1 ($max = \mathit{Exact}(c) \leq \mathit{Estim}(c) \leq max \Rightarrow \mathit{Exact}(c) = \mathit{Estim}(c)$), and the closer to saturation a container is, the higher its precision will be. Figure 4 shows the positive correlation, after some time, between the median precision (dotted line) and the average report saturation (average ($\mathit{Exact}(c)/max$) over every report container c, dashed line) for $PN = 0$. For other values of PN, median precision asymptotically decreases to a limit that depends on PN: the higher PN, the higher precision.

5.3 Simplification

Without simplification (§ 3), the provenance graph grows by one node and one or two edges per event on a sensitive container. Memory and also time consumption may then become critical because max-flow computation time is quadratic in the size of the graph. In particular, monitoring Alice's behavior at the OS level requires observing many events ($\sim$ 40 system calls per action), which significantly impacts the size of the provenance graph.

Figure 5 shows the growth of the provenance graph as the number of actions increases. We compare the monitoring of each action as an atomic transfer (1 action = 1 event) with the instantiation at the OS level (1 action $\sim$ 40 events), with and without simplification. By construction, the graphs grow by at most one node after each event. As expected, monitoring a fourty-fold number of events is reflected in a graph fourty times larger (Sn). However, simplification (Ss) made the graph as small as the atomic one (As), with direct implications in terms of time required to capture a new event, as explained in the next section. After 336248 events, Ss contains 5460 nodes only, which is less than 2% of the non-simplified graph's size.

5.4 Performance

In addition to precision, we are also interested in evaluating the storage and the time required to update our model when receiving an event. As for the graph's size, performance depends on the sequence of actions performed by the user. Our implementation requires a fixed amount of $\sim$12Mb of RAM plus $\sim$285 bytes per node and $\sim$110 bytes per edge, i.e. less than 17Mb in total for traces of more than 330K

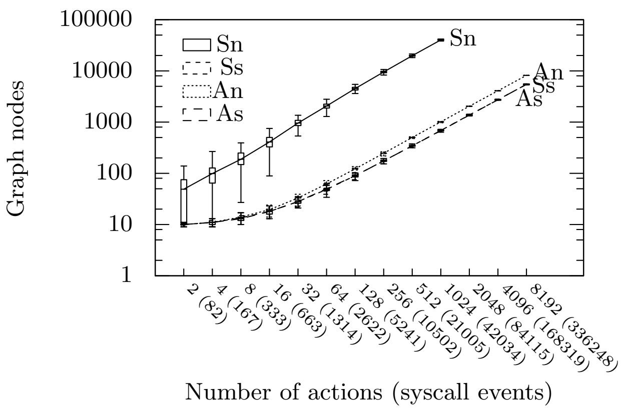

Figure 5: Graph size vs. number of actions (PN=50%), level of abstraction (**A**tomic/**S**yscall), with (**s**) and without (**n**) simplification. Note As and Ss overlap.

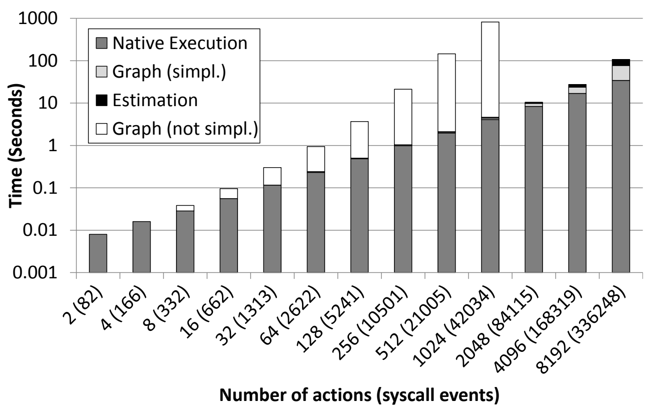

Figure 6: Time overhead for the *syscalls* tests (PN=50%) with and without QDFT, with and without optimizations.

events. Because these numbers are negligible on modern machines, we focus our analysis on the dimension of time.
Worst-case scenario We observe that it is possible for every **C2** step to replace the edge's capacity by ∞ without impacting correctness or precision. This allows us to build a provenance graph at runtime *without* computing max-flow: only if a usage event (i.e. an event whose execution is constrained by a quantitative policy) addressing a container c is observed, max-flow from source to c is computed (*lazy evaluation*). Concretely, in our scenario this means computing max-flow only if we observe a `write()` system call to one of the containers in *MAILS*, and not for every `read()` and `write()` in the trace. For instance, if only 1 in every 10 reports is actually sent via email (the rest is sent to other departments), then the median overhead after 8192 actions (336248 events) is 43 seconds instead of 72 (124% vs 211%).

To answer the second research question, we measure the performance overhead of maintaining our provenance graph with and without applying optimizations (simplification rules and lazy evaluation). Figure 6 shows the median performance overhead compared to the native execution time. For the optimized case, we distinguish between the minimum overhead, introduced by the pure creation of the graph, without any max-flow computation (Graph (simpl.)) and the maximum overhead, required to compute max-flow after the execution of each event (Estimation). Although the time to maintain the non-simplified graph grows very quickly (13700% of the native execution time for 1024 actions), the overhead remains quite small for the optimized version (43 seconds, 124%, two orders of magnitude less than before for traces 8 times longer). In this example, a max-flow computation on such graphs always requires less than 115 msec.
Average-case scenario The numbers obtained from the previous experiments make scalability a concern in the worst case. However, these experiments were *designed* to stress this particular aspect. In order to test whether or not scalability is an issue *in an average real-world scenario*, we conducted another experiment: we instantiated the model at the level of MS Windows, where events are API calls (`WriteFile()`, `ReadFile()`, `CreateFile()`, etc.) and containers are files, pipes and memory locations. As in the previous experiments, we considered a set of sensitive source files and the creation of 64 reports with PN=50%. In this scenario, we simulated a user opening two files with a text editor (notepad or wordpad) and copy-pasting some content from each source to a third file, possibly adding some (non-sensitive) content before saving it. Note that the destination could be either a new report or an existing one.

In this scenario we noted two important differences: a) in addition to the API calls generated by the user interaction, we considered every other API call in the system, including those generated by background processes; b) some processes in Windows read/write files using many API calls with blocks of fixed size (e.g. 4KB for notepad), whilst other transfer large amounts of data in a single call, similar to the behaviors of, respectively, the *syscalls* experiment and the *atomic* experiment described before.

We observed a total of $60K$ API calls, out of which $48K$ were filtered using trivial heuristics (like ignoring certain system services). The remaining 12K were ca. 10% related to sensitive data and 90% unrelated "noise" calls created by other processes. All the 12K calls were sent to our model and translated into respective **I**, **C** or **T** actions, updating the graph and requiring for each event an estimation of the sensitive data in its target container. The total time needed by our model to handle all the updates was ca. 40msec, less than 17% of the native execution time of the 12K API calls. With respect to the total time required by a fast user to perform the task (at least 5 seconds per report), this overhead is not perceptible. This further instantiation of our model shows that it is indeed usable in a realistic scenario.

5.5 Discussion

The experimental results confirm that the precision of our QDFT lies in-between a coarse-grained and a fine-grained tainting approach. Although highly dependent on the properties of the sequences of actions analyzed, the precision of our quantitative estimation is usually quite good (i.e., the estimated amount of different sensitive units is close to the exact amount). In terms of performance, without the application of simplification rules we notice the expected quadratic dependence between time required to perform an operation on the graph and its size (which, in turn, is linearly growing with the number of actions performed). With the introduction of simplification and lazy evaluation, the size of the graph decreases by more than one order of magnitude and the performance of our model improves significantly.

The quadratic complexity of the max-flow algorithm will always be a limit to the scalability of this approach; however, in an average-case scenario the overhead introduced by our model may remain negligible. In several contexts, moreover, sensitivity of data is *time-bounded*, e.g., a phone specification may be sensitive only until the phone is released, hence no need to maintain a provenance graph afterward.

Finally, if our traces represent real-world documents creation, it is unlikely that Alice would anyway notice an overall overhead of 72 seconds during the preparation of more than 8000 reports ($\sim 9ms$ per report). Though this overhead does not take into account the delay introduced by the system call interposition framework (up to 270% [19], and 1-3 orders of magnitude for other levels of abstraction), note that the delay introduced by our model is absolute, i.e. *independent* of the time required to *perform* or *intercept* any action. This means that, for instance, $1ms$ to update the provenance graph may be an intolerable overhead if we are modeling the execution of a system call, but may at the same time be a negligible delay for a BPEL or ESB event [26].

6. RELATED WORK

Information flow tracking, in the context of security policy enforcement [32], is at least thirty years old [16]. The two main approaches are *static* [13, 25, 18] and *runtime* checking, based on dynamic tainting analysis. In terms of static checking, Denning [14] first proposed to quantitatively measure information flow, defining the amount of information transferred in a flow as "the reduction in uncertainty (entropy) of a random variable". Solutions in this area (e.g., [7, 8]) rely on a specific input distribution's entropy or universally quantify over all input distributions. In contrast, our analysis applies to runtime systems and is independent of any stochastic notion of input data distribution.

Approaches more similar to ours perform dynamic taint analysis (DTA). Several techniques have been proposed for DTA, mainly for detecting malware and unknown vulnerabilities in software [10, 34, 4], checking integrity (tainted data should not affect normal behavior of the program, where tainted means "possibly bad") [10, 11] and confidentiality (public output should not be influenced by tainted data, where tainted stands for "possibly secret") [6, 23, 38]. However, most of them are "hybrid" approaches, because they rely on static annotations to account for implicit flows [13, 36]. In contrast, our approach does not require any static annotation to perform the analysis. A common pattern in all these solutions is the idea that monitoring should be done "as close to the hardware as possible" [38]. We hence find solutions based on binary rewriting [5, 10, 9, 21, 27, 23], memory and pointer analysis [33, 34], partial- or full-system emulation [20, 24, 38, 6, 12] or on making information flow a first-class OS abstraction [15, 39, 22]. In contrast, we believe that high-level events such as "print" or "play" or "screenshot" are handled more conveniently at higher levels of abstraction because they can directly been observed there. For this reason, our abstract model is deliberately not bound to one specific architecture or platform.

Our work is inspired by [23] which introduces the idea of measuring information flow as a network flow capacity. Their tool estimates the amount of information flowing from inputs to outputs of a particular program for one (or some) specific executions. Our work generalizes the idea to a generic model that can be instantiated to multiple layers of abstraction, including the level of code considered by McCamant et al. We have commented on the benefits of data flow tracking at levels different from the machine code level above. Since our model is based on the observation of events and therefore does not consider control flow dependencies, it does not measure implicit flows. At the cost of manual instrumentation, these are, in contrast, considered in [23].

The idea of measuring information flows by considering information as an incompressible fluid flowing through a network appears also in [37], where it has been applied to the socio-information networks domain. This model uses data-flow risk estimations for access control purposes, assuming likelihood of information leakages (e.g., in-between subjects) to be given. Our work, in contrast, is domain-agnostic and relies only on the size of actions that actually took place.

Data provenance tracking has been thoroughly investigated, in particular in the context of databases [3]. A recent work of Demsky [12] relies on data provenance tracking to perform basic usage control enforcement, similar to what our system does. In addition to the more advanced types of policies our system can enforce, the goal of our provenance tracking is different: we use provenance to determine *how much* data comes from *where*, whereas Demsky's and other approaches in literature track *what* data comes from *where*.

Our model is a *quantitative* extension of the data-flow tracking model for usage control presented in [29], which makes the distinction between data and representations of data, in contrast to other usage control models (e.g., [31, 35, 28]). Since this model has been instantiated at different levels of abstraction, such as OS, X11, Java-bytecode, and Android mobile devices, it makes our extension suitable for all these environments.

7. DISCUSSION

We start by making the assumptions of our model explicit. First of all, data must be *quantifiable* with respect to a *data unit*. Choosing the appropriate unit for each level of abstraction is not a trivial task, especially in presence of heterogeneous representations (e.g., an OS handles both files and windows). Furthermore, converting a type of units into another type is usually not possible. For example, two records of a database, containing ten characters each, may have the same size in bytes as a single record containing twenty characters. How could a monitor using bytes as data units distinguish between the two cases?

Additionally, we also assume the absence of *data compression*. Otherwise, an event may transfer more data than the size of the event itself or transferring a single data unit from one representation could mean transferring the whole data. This assumption can be removed if an upper bound for the compression ratio of each event is known. An event of compression ratio r $(0 < r \leq 1)$ that transfers t units of data will be considered of size $s = t/r$.

Next, we assume a *fixed size for the initial representation*. If such size changes over time, a policy like *no more than 5% of customer data can be stored in an unencrypted file* can be interpreted in different ways: 5% of the amount of customer data when the policy is activated (static), 5% according to the current amount of customer data (current), or 5% according to the total amount of different customers (max). For example, if at time t_1 the customer database contains 1000 records, at time t_2 it contains 200 records and at time t_3 it contains 2000 records, the maximal amount of data that can be stored in an unencrypted file at time t_3 without violating the property can be either: 50 (static), 100 (current), or 140 (max). To handle these various interpretations, the roots of each provenance graph need some adjustments whenever the size of an initial representation changes. We implemented all these different interpretations but do not include them in this paper due to space limitations.

A common problem in data-flow tracking solutions is the *over-tainting* issue. Quantitative estimations can be used as declassification criterion to mitigate the problem by constraining the application of a policy to only those containers containing at least a certain amount of sensitive data (cf. Figure 2). However, if a small amount of data is allowed for disclosure, we also need to make sure that such disclosures cannot be channeled to the same recipient via unknown ways. For example, in our scenario, we need to make sure that the same supplier cannot receive mails on more than one account. Otherwise she could receive 10KB of different data on each email address, and with enough addresses (and Alice cooperation) she could possibly obtain all the phone specifications and test results via mail. For this reason, usage control usually requires a fully controlled environment.

Our QDFT maintains a flow graph per data item. Assuming D sensitive data items and at most V representations per item, linked by at most E edges, the memory overhead is in $\mathcal{O}(D(V+E)) = \mathcal{O}(DE)$. We use a maxflow algorithm [2] which gives the best performance on graphs with less than 10^6 nodes [17]. The augmenting paths algorithm is based on a residual graph that is reused for subsequent maxflow computations. It is hence particularly suitable for our problem because the provenance graph is built incrementally.

The existence of one independent provenance graph per data item introduces an overapproximation. Consider a container c_a containing units of data items d_1 and d_2. A transfer of size l from c_a to c_b would be modeled as a transfer of l units from c_a to c_b in the provenance graph of d_1, and as a transfer of l units from c_a to c_b in the graph of d_2. However, in a single transfer of size l, it is not possible to transfer both, l units of data d_1 and l units of data d_2. This overapproximation can be reduced by modifying the analysis, but the task must be done carefully, in order to preserve soundness (e.g., multiple data may share capacity via coding [1]).

As a final remark, quantitative measurements, in themselves, do not always reflect the usefulness of data. For example, consider a group picture: removing every second pixel or taking only the bottom half of the picture reduces in both cases the size of 50%. But the reduction in terms of information content may be very different in the two cases for someone who wants to identify people in the picture. In general, quantitative measurements make sense only if combined with the interpretation of data.

8. CONCLUSION

In this paper, we presented a model for quantitatively tracking flows of sensitive data. This model works in a dynamic context of runtime monitors. We proved the correctness of this model and assessed its precision and performance by embedding it in a usage control infrastructure.

Depending on the specific actions performed by a user, our model's precision can be as good as a fine-grained approach, which knows exactly how many different data units are in every container, or as bad as a coarse-grained approach, which knows only if a data item could be in a container but not exactly how many data units are in it. Usually, our model precision lies somewhere in between because it knows that not every data unit in a container is sensitive but over-approximates the exact amount of sensitive units.

Our model is embodied in a usage control infrastructure, similarly to what has been done in [29], and used to enforce, preventively or detectively, usage control policies. A valuable use of our system is to support the notion of acceptable exceptions, i.e. quantitative policies defined *a posteriori*. For example, in case of a data leakage, analyzing the logs of the actions with our model may establish that only 0.1% of sensitive data have been leaked. At this point, an auditor may decide that this violation is still acceptable, whereas it is unlikely that a policy defines a leakage of 0.1% to be acceptable in advance, i.e., before the leakage happens.

In terms of future work, we want to extend our model to cope with compression, model sources of data of variable sizes and investigate the problem of measuring data across different layers of abstraction (i.e., converting units of data).

Acknowledgments This work was done as part of the DFG's Priority Program SPP 1496 "Reliably Secure Software Systems," ref. numbers PR-1266/1-2. Fabio Massacci suggested to present the results as done in Figure 2.

9. REFERENCES

[1] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network information flow. *IEEE Transactions on information theory*, pages 1204–1216, 2000.

[2] Y. Boykov and V. Kolmogorov. An experimental comparison of min-cut/max-flow algorithms for energy minimization in vision. *TPAMI*, pages 1124–1137, 2004.

[3] P. Buneman, S. Khanna, and T. Wang-Chiew. Why and Where: A Characterization of Data Provenance. In *ICDT*, pages 316–330, 2001.

[4] L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis. In *DIMVA*, pages 143–163, 2008.

[5] W. Cheng, Q. Zhao, B. Yu, and S. Hiroshige. Tainttrace: Efficient flow tracing with dynamic binary rewriting. In *ISCC*, pages 749–754, 2006.

[6] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum. Understanding Data Lifetime via Whole System Simulation. In *SSYM*, pages 321–336, 2004.

[7] D. Clark, S. Hunt, and P. Malacaria. Quantitative Analysis of the Leakage of Confidential Data. *ENTCS*, 59:238–251, 2002.

[8] M. Clarkson, A. Myers, and F. Schneider. Belief in information flow. In *CSFW*, pages 31–45, 2005.

[9] J. A. Clause, W. Li, and A. Orso. Dytan: a generic dynamic taint analysis framework. In *ISSTA*, pages 196–206, 2007.

[10] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worm epidemics. *TOCS*, pages 1–68, 2008.

[11] J. R. Crandall and F. T. Chong. Minos: Control Data Attack Prevention Orthogonal to Memory Model. In *MICRO-37*, pages 221–232. IEEE, 2004.

[12] B. Demsky. Cross-application data provenance and policy enforcement. *TISSEC*, pages 1–22, 2011.

[13] D. E. Denning. A lattice model of secure information flow. *Commun. ACM*, pages 236–243, May 1976.

[14] R. Denning and D. Elizabeth. *Cryptography and data security.* Addison-Wesley, 1982.

[15] P. Efstathopoulos and E. Kohler. Manageable fine-grained information flow. In *SIGOPS*, pages 301–313, 2008.

[16] J. S. Fenton. Memoryless subsystems. *The Computer Journal*, 17(2):143–147, Feb. 1974.

[17] B. Fishbain, D. S. Hochbaum, and S. Mueller. Competitive analysis of min-cut max-flow algorithms in vision problems. Technical report, UC Berkeley, 2010.

[18] J. W. Gray. Toward a mathematical foundation for information flow security. In *SP*, pages 21–34, 1991.

[19] M. Harvan and A. Pretschner. State-based Usage Control Enforcement with Data Flow Tracking using System Call Interposition. In *NSS*, pages 373–380, 2009.

[20] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. *EuroSys 06*, 40(4):29, 2006.

[21] M. G. Kang, S. McCamant, P. Poosankam, and D. Song. DTA++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[22] M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *SOSP*, pages 321–334, 2007.

[23] S. McCamant and M. D. Ernst. Quantitative information flow as network flow capacity. In *PLDI*, pages 193–205, 2008.

[24] J. Mccullough, M. Vrable, A. C. Snoeren, G. M. Voelker, and S. Savage. Neon: system support for derived data management. *VEE*, pages 63–74, 2010.

[25] J. K. Millen. Covert channel capacity. In *SP*, pages 60–66, 1987.

[26] R. Neisse, A. Pretschner, and V. D. Giacomo. A trustworthy usage control enforcement framework. In *ARES*, pages 230–235. IEEE, 2011.

[27] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 42(6):89–100, 2007.

[28] J. Park and R. Sandhu. The ucon abc usage control model. *TISSEC*, pages 128–174, 2004.

[29] A. Pretschner, E. Lovat, and M. Büchler. Representation-independent data usage control. In *Proc. SETOP/DPM*, pages 122–140, 2011.

[30] N. Provos. Improving host security with system call policies. In *Proc. SSYM*, pages 257–272, 2003.

[31] E. Rissanen. Extensible access control markup language v3.0, 2010.

[32] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

[33] A. Slowinska and H. Bos. Pointless tainting?: evaluating the practicality of pointer tainting. In *Proc. EuroSys '09*, pages 61–74. ACM, 2009.

[34] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, pages 85–96, 2004.

[35] K. Twidle, N. Dulay, E. Lupu, and M. Sloman. Ponder2: A Policy System for Autonomous Pervasive Environments. In *ICAS*, ICAS, pages 330–335, 2009.

[36] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. Reis, M. Vachharajani, and D. August. Rifle: An architectural framework for user-centric information-flow security. *MICRO 37*, 2004.

[37] T. Wang, M. Srivatsa, D. Agrawal, and L. Liu. Modeling data flow in socio-information networks: a risk estimation approach. In *SACMAT*, pages 113–122, 2011.

[38] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. *ACM CCS*, 2007.

[39] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazi. Making Information Flow Explicit in HiStar. In *OSDI*, pages 263–278, 2006.

[40] X. Zhang, J. Park, F. Parisi-Presicce, and R. S. Sandhu. A logical specification for usage control. In *SACMAT*, pages 1–10, 2004.

APPENDIX

The two correctness proofs assume that the label of edges corresponding to C2 steps are ∞ rather than $\kappa((c_2, t'))$ for the source container c_2 and a time $t' \in \mathbb{N}$. Since $\kappa((c_2, t')) < \infty$, this does not impact correctness. Precision also is not impacted because the edge can, in any case, not transfer more than $\kappa((c_2, t'))$ units. The reason for not using ∞ in the definition of the C2 step is that, throughout the paper, this would have obscured the dual nature of edge labels as capacities and actual flows — actual flows are never infinite.

A. CORRECTNESS OF TRACKING

To show correctness of the model in the sense of § 3, we need to prove $\forall\, t \in \mathbb{N}\, \forall\, c \in C : \varphi(c, t) \leq \kappa((c, t))$.

The proof is by induction. It is trivial for the empty provenance graph (PG) of the base case. In terms of the inductive step, we assume that correctness has been established for all $\mathcal{G}_{t'}$ with $t' < t$: The induction hypothesis $(*)$ is $\forall\, c \in C\, \forall\, t' \in \mathbb{N} : t' < t \Rightarrow \varphi(c, t') \leq \kappa((c, t'))$.

I step: The only edge from $(S_d, 0)$ to (c_i, t) is the newly created one with label m. Consequently, $\kappa((c_i, t)) = m$. There are exactly m different units in c_i by assumption, and hence $\varphi(c_i, t) = m \leq \kappa((c_i, t))$.

T step: A container c (most recent node with time stamp $t' < t$) is reduced to size m. If $m \geq \kappa((c, t'))$, the PG is not modified, thus $\mathcal{G}_t = \mathcal{G}_{t-1}$. This implies $\kappa((c, t)) = \kappa((c, t'))$ which, by induction hypothesis $(*)$ was a correct estimation for $\varphi(c, t')$. Since truncating actions remove data from a container, the amount of sensitive data in c can be either the same of or less than before. Therefore, $\kappa((c, t'))$ is still a correct estimation, and we have $\varphi(c, t) \leq \kappa((c, t))$.

If, instead, $m < \kappa((c, t'))$, then node (c, t) is added as a successor of (c, t') with an edge labeled m. Because (c, t') is the only predecessor of (c, t), the min cut will now contain exactly this edge, hence $\kappa((c, t)) = m$. This estimation is also correct, because if the new size of c is m, then c cannot contain more than m different units of sensitive data. We have $\varphi(c, t) \leq \kappa((c, t))$.

C1 step: ℓ units of data are to be copied from c_A to a c_B that does not exist at time $t - 1$. We introduce a new edge e_ℓ with label ℓ from (c_A, t') to the newly created (c_B, t) where t' is the most recent timestamp for c_A. Because c_B cannot contain more different units of sensitive data than c_A, we know $\varphi(c_B, t) \leq \min(\ell, \varphi(c_A, t'))$. Let $\alpha = \kappa((c_A, t'))$. By induction hypothesis $(*)$, $\varphi(c_A, t') \leq \alpha$. Hence $\varphi(c_B, t) \leq \min(\ell, \alpha)$. If $\ell < \alpha$, $\{e_\ell\}$ is the only edge crossing the minimal cut, and we have $\kappa((c_B, t)) = \ell$. If $\ell \geq \alpha$, then $\kappa((c_B, t)) = \kappa((c_A, t')) = \alpha$. In both cases $\varphi(c_B, t) \leq \kappa((c_B, t))$.

C1 step immediately followed by a C2 step: Assume we copy ℓ units of data from container c_A to the existing container c_B. By construction of the PG, there are $t' < t$ and $t'' < t$ and nodes $(c_A, t'), (c_B, t'') \in \mathcal{N}_{t-1}$, a new node $(c_B, t) \in \mathcal{N}_t$, and two edges $e_\ell = ((c_A, t'), \ell, (c_B, t))$ and $e_k = \big((c_B, t''), \infty, (c_B, t)\big) \in \mathcal{E}_t$.

We note that by construction in every $\mathcal{E}_\tau$ $(\tau < t)$, all labels corresponding to I, C1, and T steps measure *actual* flows of data (sensitive and non-sensitive units) in-between containers and are determined by the action corresponding to the step. We can interpret these edge labels as upper bounds for the flow of *sensitive* units: For each edge (action), it is impossible that more sensitive than overall units flow.

We then observe that all edges $\big((c, t'''), \infty, (c, \tau)\big) \in \mathcal{E}_\tau$ for all $\tau < t$ corresponding to C2 steps also provide (trivial) upper bounds for the flow of sensitive units. By induction hypothesis $(*)$, $\varphi(c_A, t') \leq \kappa((c_A, t'))$ and $\varphi(c_B, t'') \leq \kappa((c_B, t''))$. The same argumentation as for the edge labels in $\mathcal{E}_\tau$ applies to the newly created labels in $\mathcal{E}_t$: The label ℓ of e_ℓ is an upper bound for the flow of sensitive units from (c_A, t') because it is impossible that more sensitive units flow than overall units flow. The label ∞ is an upper bound for the number of sensitive units in (c_B, t''), $\varphi(c_B, t'')$, which by induction hypothesis is bounded from above by $\kappa((c_B, t''))$.

In sum, all edge labels in $\mathcal{E}_t$ are upper bounds for the flow of *sensitive* units for all kinds of steps. Edge labels can hence be seen as capacities for flows of sensitive data units.

Then, every cut of the PG between $(S_d, 0)$ and (c_B, t) partitions the nodes in two sets: the source set containing $(S_d, 0)$ and the destination set containing (c_B, t). Since labels are capacities for flows of different sensitive units, the size of every cut between $(S_d, 0)$ and (c_B, t) is an upper bound for the amount of different sensitive units that flowed to any node in the destination set. Specifically, for every cut between $(S_d, 0)$ and (c_B, t), $\varphi(c_B, t)$ can't exceed the size of this cut.

Finally, by definition, $\kappa((c_B, t))$ is the maximum flow between $(S_d, 0)$ and (c_B, t). By the max-flow/min-cut theorem, this is equivalent to the value of a minimal cut that separates (c_B, t) from $(S_d, 0)$. $\kappa((c_B, t))$ hence necessarily is an upper bound for $\varphi(c_B, t)$. $\square$

Note that the proof's structure also shows that the max-flow must be computed explicitly for combined C1&C2 steps only—for the others, κ values can directly be determined.

B. CORRECTNESS OF OPTIMIZATIONS

We prove equivalence $\sim$ of a graph $\mathcal{G}_t$ with its optimized version $\mathcal{G}'_t$ according to the rules presented in § 3.

Let $I(a, q)$ denote actions that initialize a container a with q units of sensitive data, $C(a_1, q_1, a_2, q_2, \ldots, b)$ actions that transfer q_1 units and q_2 units from container a_1 and a_2 to b, respectively, and $T(a, q)$ truncations of container a to q units. C is the set of containers. While function *maxflow* returns a number, let *mincut* returns the cut.

Removal of Truncation. For a sequence of events $tr = (\varepsilon_0, \ldots, \varepsilon_k, \ldots, \varepsilon_t)$ such that $\exists\, k \in \mathbb{N}\, \forall\, c_a, c_b, c \in C, q, x \in \mathbb{N}, i \in [k+1..t-1], j \in [0..k-1] : \varepsilon_k = C(c_a, q, c_b) \wedge \varepsilon_t = T(c_b, m) \wedge \varepsilon_j \neq I(c_b, x) \wedge \varepsilon_j \neq C(c, x, c_b) \wedge \varepsilon_i \neq T(c_b, x) \wedge \varepsilon_i \neq C(c, x, c_b) \wedge \kappa((c_b, k)) > m \wedge s_Y + m \leq q$ where $s_Y = \sum_{\varepsilon_i = C(c_b, x, c) \in tr} x$, we construct a new sequence $tr' = (\varepsilon_0, \ldots, \varepsilon_{k-1}, C(c_a, m, c_b), \varepsilon'_{k+1} \ldots, \varepsilon'_{t-1})$ with $\varepsilon'_i = C(c_a, q_i, c_i)$ if $\varepsilon_i = C(c_b, q_i, c_i)$ or $\varepsilon'_i = \varepsilon_i$ otherwise, for every $i \in [k+1..t-1]$. tr' results in a simpler (i.e. *smaller*), yet equivalent graph. Let (c_a, u) be the source node of the copy action at time k and, because there is no truncation or transfer to c_b in between times $k+1$ and $t-1$, (c_b, k) the source node of the truncation action at time t. We define $\mathcal{N}'_t = \mathcal{N}_t \setminus \{(c_b, t)\}$ and $\mathcal{E}'_t = \Big(\mathcal{E}_t \setminus \big(\{((c_b, k), m, (c_b, t)), ((c_a, u), q, (c_b, k))\} \cup \bigcup_{\varepsilon_i = C(c_b, q_i, c_i)} \{((c_b, k), q_i, (c_i, i))\}\big)\Big) \cup \big(\{((c_a, u), m, (c_b, k))\} \cup \bigcup_{\varepsilon_i = C(c_b, q_i, c_i)} \{((c_a, u), q_i, (c_i, i))\}\big)$. Let $X \subseteq C$ be a set of containers. $\kappa_{t,X}(d) \neq \kappa'_{t,X}(d) \Rightarrow ((c_a, u), q, (c_b, k)) \in mincut_{\mathcal{G}_{t,X}}((S, 0), d)$ because any other edge has the same value in both graphs, but this is impossible (i.e. it is not a minimum cut) since $q \geq m + s_Y$. Hence $\mathcal{G}'_t \sim \mathcal{G}_t$.

Removal of Copy (case 1). For a sequence of events $tr = (\varepsilon_0, \ldots, \varepsilon_k, \ldots, \varepsilon_t)$ such that $\exists\, k \in \mathbb{N}\, \forall\, c, c_a \in C, q \in \mathbb{N}, i \in [k+1..t-1], x \in \mathbb{N} : \varepsilon_k = C(c_a, s_1, c_b) \wedge \varepsilon_t = C(c_c, s_2, c_b) \wedge \varepsilon_i \neq T(c_b, x) \wedge \varepsilon_i \neq C(c, x, c_b) \wedge \varepsilon_i \neq C(c_b, x, c) \wedge \varepsilon_i \neq T(c_a, x) \wedge \varepsilon_i \neq C(c, x, c_a)$, we construct a new sequence $tr' = (\varepsilon_0, \ldots, \varepsilon_{k-1}, C(c_a, s_1, c_c, s_2, c_b), \ldots, \varepsilon_{t-1})$ that results in a simpler, yet equivalent graph. Let (c_a, u) be the source node of the copy action at time k, (c_b, v) the current node for c_b (if any) at time $k-1$, and (c_c, w) the source node of the copy action at time t. We define $\mathcal{N}'_t = \mathcal{N}_t \setminus \{(c_b, t)\}$ and $\mathcal{E}'_t = (\mathcal{E}_t \setminus \{((c_b, k), \infty, (c_b, t)), ((c_c, w), s_2, (c_b, t))\}) \cup \{((c_c, w), s_2, (c_b, k))\}$. Let $X \subseteq C$ be a set of containers. If $c_b \notin X$, then $\kappa_{t,X}(d) = \kappa'_{t,X}(d)$ because no flow would go through the modified part. If $c_b \in X$, $\kappa_{t,X}(d) \neq \kappa'_{t,X}(d) \Rightarrow \big(((c_c, w), s_2, (c_b, k)) \in mincut_{\mathcal{G}'_{t,X}}((S, 0), d) \wedge ((c_c, w), s_2, (c_b, t)) \notin mincut_{\mathcal{G}_{t,X}}((S, 0), d)\big)$. This is impossible since the node (c_b, t) must be separated from $(S, 0)$ by the cut and $s_2 < \infty$. Therefore, having $((c_c, w), s_2, (c_b, k))$ in the cut in $\mathcal{G}'_t$ implies that $((c_c, w), s_2, (c_b, t))$ is in the cut in $\mathcal{G}_t$. Hence $\mathcal{G}'_t \sim \mathcal{G}_t$.

Removal of Copy (case 2). For a sequence of events $tr = (\varepsilon_0, \ldots, \varepsilon_k, \ldots, \varepsilon_t)$ such that $\exists\, k \in \mathbb{N}\, \forall\, c_a, c \in C, q, x \in \mathbb{N}, i \in [k+1..t-1], j \in [0..k-1] : \varepsilon_k = C(c_a, s_1, c_b) \wedge \varepsilon_t = C(c_a, s_2, c_b) \wedge \varepsilon_j \neq I(c_b, x) \wedge \varepsilon_j \neq C(c, x, c_b) \wedge \varepsilon_i \neq T(c_b, x) \wedge \varepsilon_i \neq C(c, x, c_b) \wedge \varepsilon_i \neq T(c_a, x) \wedge \varepsilon_i \neq C(c, x, c_a) \wedge s_1 > \sum_{\varepsilon_i = C(c_b, x, c)} x$, we construct a new sequence $tr' = (\varepsilon_0, \ldots, \varepsilon_{k-1}, C(c_a, s_1 + s_2, c_b), \ldots, \varepsilon_{t-1})$ that results in a simpler, yet equivalent graph. Let (c_a, u) be the source node of the copy action at time k and $Y = \{((c_b, k), q_i, (c_i, i)) \mid i \in [k+1..t-1], c_i \in \mathcal{C}, q_i \in \mathbb{N}, \varepsilon_i = C(c_b, q_i, c_i) \in tr\}$ the destination nodes of the copy steps from c_b between time $k+1$ and time $t-1$, and s_Y the sum of those edges' capacities. We define $\mathcal{N}'_t = \mathcal{N}_t \setminus \{(c_b, t)\}$ and $\mathcal{E}'_t = (\mathcal{E}_t \setminus \{((c_b, k), \infty, (c_b, t)), ((c_a, u), s_1, (c_b, k)), ((c_a, u), s_2, (c_b, t))\}) \cup \{((c_a, u), s_1 + s_2, (c_b, k))\}$. Let $X \subseteq C$ be a set of containers. $\kappa_{t,X}(d) \neq \kappa'_{t,X}(d) \Rightarrow Y \cap mincut_{\mathcal{G}_{t,X}}((S, 0), d) \neq \varnothing$. Let s_X be the sum of those edges' capacities. Since $\kappa_{t,X}(d) \neq \kappa'_{t,X}(d)$, it means $s_1 < s_X$. This is impossible since $s_X \leq s_Y \leq s_1$. Hence $\mathcal{G}'_t \sim \mathcal{G}_t$.